Advanced RenderMan 3: Render Harder

SIGGRAPH 2001 Course 48

course organizer: Larry Gritz, Exluna, Inc.

Lecturers:

Tony Apodaca, Pixar Animation Studios
Larry Gritz, Exluna, Inc.
Matt Pharr, Exluna, Inc.
Christophe Hery, Industrial Light + Magic
Kevin Bjorke, Square USA
Lawrence Treweek, Sony Pictures Imageworks

August 2001

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, 1999, 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.

> 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 is a co-founder and VP of Exluna, Inc., where he is the head of R&D for Entropy and BMRT. Larry was previously the head of the rendering research group at Pixar Animation Studios and chief architect of the RenderMan Interface Specification, as well as serving as a technical director on several film projects. His film credits include *Toy Story*, *Geri's Game*, *A Bug's Life*, *Toy Story 2*, and *Monsters, Inc.* Larry has a BS from Cornell University and MS and PhD degrees from the George Washington University. Larry co-wrote the book Advanced RenderMan: Creating CGI for Motion Pictures" (Morgan-Kaufmann, 1999), and has spoken at four previous SIGGRAPH courses on RenderMan (as course organizer for three of them).

Matt Pharr is a co-founder and VP of Interactive Graphics at Exluna, Inc., where he has also contributed substantially to the design and implementation of the Entropy rendering architecture. He is expected to complete his Ph.D. in Computer Science at Stanford University in 2001. Matt has published three papers at SIGGRAPH in the past four years on topics of light scattering and rendering of large scenes. He was a part-time member of Pixar's Rendering R&D group from 1998-2000, contributing to development of PhotoRealistic RenderMan and next generation rendering architectures. He has a B.S degree in Computer Science from Yale University and an M.S. from Stanford; his movie credits include *A Bug's Life*, *Toy Story 2*, and *Monsters, Inc.*

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.

Christophe Hery is an Associate Visual Effects Supervisor at Industrial Light + Magic (ILM). Christophe joined ILM in 1993 as a senior technical director on The Flinstones. Prior to that, he was working for Acteurs Auteurs Associes (AAA) studio in France setting up a CG facility devoted to feature film production. Christophe majored in architecture and electrical engineering at ESTP, where he received his degree in 1989. While a student in Paris he freelanced as a technical director at BUF, as well as Thomson Digital Images, where he worked on 3-D software development in his spare time. After graduation, Christophe took a job as director of research and development at Label 35, a Parisian cartoon studio. Christophe is currently working as a CG supervisor on *Jurassic Park 3*. His numerous projects at ILM include *Star Wars: Episode I: The Phantom Menace*, *Mission to Mars*, *Jumanji*, *Casper*, *The Mask*, and many other films.

Kevin Bjorke was the Lighting and Shading supervisor on *Final Fantasy: The Spirits Within.* Prior to *Final Fantasy*, he worked at a variety of studios in California, New York, and Europe, including Pixar and Digital Productions. He has collected three gold Clios for computer animation, and was one of the original off-site testers of Renderman, running it on an RM-1 board and rendering TV commercials on ChapReyes.

Laurence Treweek is a senior technical director at Sony Pictures Imageworks. He has worked on many films including *Hollow Man*, *Stuart Little*, *Stuart Little 2*, and *Contact*.

Brian Steiner is a senior technical director at Sony Imageworks, joining the company to work on *Hollow Man* and currently working on *Stuart Little 2*. Brian majored in computer animation at

Ringling School of Art & Design, and in 1994 came to Los Angeles to work for RezN.8 Productions. Since then he has worked at Boss Films Studios, Warner Bros., and Pacific Title. Some of Brian's movie credits include *Outbreak, Species, Batman Forever, Mars Attacks,* and *Duck Dodgers in the 21 Century*.

Douglas Sutton is a Lead Technical Director at Industrial Light + Magic(ILM). During his 4+ years at ILM he has worked on a variety of film projects including *The Lost World*, *Flubber*, *Deep Impact*, *Star Wars: Episode I: The Phantom Menace*, *Galaxy Quest*, *Mission to Mars* and *A Perfect Storm*. Previous to his work at ILM Doug worked at Will Vinton Studios in Portland Oregon and at Lamb and Co. in Minneapolis, Minnesota. He has a B.S. degree in Mechanical Engineering from the University of Minnesota and studied computer graphics as a graduate student at the U of M as well.

Schedule

Welcome and Introduction Larry Gritz	8:30 AM
Interpreting *PhotoRealistic RenderMan* Statistics Tony Apodaca	8:45 AM
A Practical Guide to Global Illumination Larry Gritz	9:45 AM
Break	10:00 AM
Global Illumination (continued)	10:15 AM
Layered Media for Surface Shaders Matt Pharr	11:00 AM
Lunch	12:00 PM
A Tutorial on Procedural Primitives Christophe Hery	1:30 PM
Using Maya with RenderMan on *Final Fantasy* Kevin Bjorke	2:30 PM
Break	3:00 PM
Maya+RenderMan on *Final Fantasy* (continued)	3:15 PM
Volumetric Shaders Used In The Production Of *Hollow Man* Lawrence Treweek	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 3: Render Harder*. (Okay, I was getting a little punchy when the submission deadline rolled around.) This course covers the theory and practice of using RenderMan and similar renderers to do the highest quality computer graphics animation production. This is the eighth 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 1998/1999 and 2000 courses (also titled "Advanced RenderMan") 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 interpretation of *PRMan*'s output statistics (and possible optimizations from what's learned there), the ins and outs of both local and global illumination and how that applies to RenderMan-compatible renderers, advanced aspects of rendering humans (with and without skin), crowds, and other essential production tasks. Not a single topic is repeated from previous SIGGRAPH course. A couple of speakers are veterans, but four 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.

Larry Gritz
April, 2001

Chapter 1

Interpreting *PRMan* Statistics

Tony Apodaca,
Pixar
aaa@pixar.com

1.1 Introduction

Starting from *PhotoRealistic RenderMan 3.8*, it is possible to gather statistics about the renderer, and with the 3.9 release this output has been expanded. This Application Note describes how to interpret these statistics.

1.2 Enabling Statistics Output

The printing of statistics at the end of the frame is controlled by the "statistics" Option:

```
Option "statistics" "endofframe" [level]
```

When enabled, statistics output will be printed after the frame has completed rendering; they will also be printed if the render is aborted mid-frame with a SIGHUP, which can be useful when analyzing memory usage. The value of `level` determines how verbose the statistics are. At level 0, nothing is printed. At level 1, information about resource usage and memory footprint is printed. At level 2, detailed statistics about geometric primitives, grids, micropolygons, and visible points are added. At level 3, statistics about the texturing system are added.

1.3 Level 1 Statistics

1.3.1 Resource Usage Summary

The output of level 1 statistics begins with a resource usage summary:

```
Rendering at 640x480 pixels, 3x3 samples
        "test.tif" (mode = rgba, type = tiff)
       Memory:        48.87 MB
       Current mem:   42.66 MB (36.71 sbrk + 5.95 mmap)
```

```
Real time:     06:14
User time:     05:04
Sys time:      00:06
Max resident mem: 52.40 MB
Page faults: 0,  Page reclaims: 63
```

Memory : The maximum amount of memory allocated on the heap over the lifetime of the render. Note that this does not include stack memory or the code and data segments, so the renderer will typically need more memory than this to operate effectively.

Current mem : The amount of memory allocated on the heap at the time the statistics were generated.

Real time : The amount of "wall clock" time elapsed since the beginning of the frame render.

User time : The amount of time spent executing user instructions on behalf of the renderer.

Sys time : The total amount of time spent in the operating system kernel executing instructions on behalf of the renderer. The sum of this time and the user time is a good indication of the total CPU time.

Max resident mem : The maximum resident set size used by the renderer, measured in kilobytes. Basically, this is the maximum amount of physical memory which the operating system sets aside for the renderer.

Page faults : The number of "hard" page faults - the number of times that a page of virtual memory in use by the renderer had to be swapped out to disk.

Page reclaims : The number of page faults serviced without any I/O activity (i.e. having to swap to disk); here I/O activity is avoided by "reclaiming" a page frame from the list of pages awaiting reallocation.

This resource usage summary is useful for timing renders as well as monitoring memory usage. For example, when the system memory is low the operating system may undergo excessive virtual memory paging ("thrashing"). This is something that should be avoided, as shown by the wall clock time in the summary below generated for the same scene above, rendered in a low memory situation. Note the number of page faults, and compare the maximum resident memory with the memory allocated by the renderer.

```
Memory:        50.41 MB
Current mem:   44.05 MB (38.10 sbrk + 5.95 mmap)
Real time:     40:04
User time:     13:45
Sys time:      00:29
Max resident mem: 35.94 MB
Page faults: 24609,  Page reclaims: 47207
```

1.3.2 Memory Breakdown

The memory breakdown table, added in PRMan 3.9, gives detailed information on the current and peak amount of memory allocated on the heap by different aspects of the renderer:

```
Memory breakdown:    Current MB      Peak MB
  Parsing:             0.52 ( 1.2%)     0.52
  GPrim:               0.00 ( 0.0%)     0.27
  Trim:                0.00 ( 0.0%)     0.11
```

```
Vertex vectors:      0.00 ( 0.0%)     1.26
Grids:               0.00 ( 0.0%)     0.04
Stitch gprims:       0.00 ( 0.0%)     0.02
Stitch edges:        0.00 ( 0.0%)     0.06
Stitch edata:        0.00 ( 0.0%)     0.08
Shader:             13.32 (31.2%)    15.73
Texture:            21.62 (50.7%)    21.62
Micropolygons:       7.34 (17.2%)     7.34
Visible points:      0.28 ( 0.7%)     0.28
Free memory:         4.51 (10.6%)
Unaccounted:         0.00 ( 0.0%)
Total:              42.66
```

Parsing : Memory needed by the RIB parser. This includes memory needed to maintain a stack as well as to store parameter lists.

GPrim : Memory allocated to store geometric primitive information, exclusive of vertex variable information.

Trim : Memory allocated for trimming information.

Vertex vectors : Memory for vertex vectors, which store vertex variables (including `P` and user-defined variables) associated with primitives.

Grids : Memory allocated for grids (two-dimensional arrays of micropolygons which are shaded in parallel).

Stitch : Memory allocated for stitching (used for crack elimination in single primitives).

Shader : Memory associated with the shading system.

Texture : Memory associated with the texture system. Most of this is reserved for caching textures read from disk. The maximum amount of memory in the cache can be controlled by setting the `"texturememory"` limit; for example, in this particular scene, it was set with the following RIB statement:

```
Option "limits" "texturememory" [20480]
```

See Section 4.1.5 of the *PRMan* User Manual for more details.

Micropolygons : Memory needed for allocating micropolygons.

Visible points : Memory associated with each subpixel zubffer sample. This includes color, opacity, and depth information about the micropolygons overlapping the sample, sorted by depth; these are composited together to compute the final color of the pixel.

Free memory : This usually indicates allocated memory which was formerly in use by the renderer, and has now been freed.

Unaccounted : Miscellaneous allocated memory which is not accounted for in one of the above categories.

The memory breakdown is useful when tweaking the various knobs in the renderer in order to trade off memory against speed against quality. Lowering the shading rate will improve quality at the expense of more micropolygons (and more memory needed to store them). Raising the bucket and grid sizes directly increases memory usage, but increases the renderer's efficiency, up to a point. Here is the memory breakdown for the same scene as above, but rendered with larger grid and bucket sizes; note the increased memory usage in several categories, which paid off with a slightly faster render time.

```
Memory breakdown:    Current MB      Peak MB
  Parsing:             0.52 ( 0.9%)     0.52
  GPrim:               0.00 ( 0.0%)     0.25
  Trim:                0.00 ( 0.0%)     0.11
  Vertex vectors:      0.00 ( 0.0%)     1.32
  Grids:               0.00 ( 0.0%)     0.07
  Stitch gprims:       0.00 ( 0.0%)     0.02
  Stitch edges:        0.00 ( 0.0%)     0.05
  Stitch edata:        0.00 ( 0.0%)     0.11
  Shader:             18.76 (33.7%)    21.15
  Texture:            21.62 (38.8%)    21.62
  Micropolygons:       9.16 (16.5%)     9.16
  Visible points:      8.21 (14.8%)     8.21
  Free memory:         5.10 ( 9.2%)
  Unaccounted:         0.00 ( 0.0%)
  Total:              55.66
```

The *PRMan* User Manual “Appnote #3” provides more information on these tradeoffs.

1.4 Level 2 Statistics

The level 2 statistics are very detailed, and are generally less useful than the level 1 statistics. In general, this information is probably only useful to Pixar engineers debugging renderer internals.

Most of these fields are self explanatory; explanations for some of the more obscure sections follow the example output.

```
Geometry:
  Gprims: created 7933, currently 0, peak 1257 (278K)
  Trim curves: created 1366, currently 0, peak 889 (108K)
  Trim data: created 878K, currently 0K, peak 67K
  NURBS: 0 failed flat tests due to uniform variables.
         531 failed flat tests due to repeated knots.
         314 sick derivatives fixed.
         15 entire grids trimmed.
         184 grids backface culled.
  Frustum box check: 3009 calls, 314 culled, 320 inside
  Vertex vectors: created 9281, currently 0K, peak 1293K
  Light lists: created 14, currently 0, peak 13 (1K)
  Lights per light list: average 6.3, peak 9
Crack elimination:
  Gprims: created 7933, currently 0, peak 1257 (24K)
  Edges: created 19227, currently 0, peak 2963 (57K)
  Edge data: created 3229K, currently 0K, peak 86K
Rendering:
  Gprims: 21639 processed, 10638 pushed forward
  ------------------------------------------------------------------
  Gprim Fates:          Total  Eye   Frust  Back   Hidden  Split  Diced
                               Split Culled Facing
  ------------------------------------------------------------------
  Initial model:          121   2.5%   7.4%         19.8%  70.2%
   (after splits)       12677   0.3%   2.5%   3.7%   6.4%  47.3%  39.4%
  In procedurals:          3                               100%
   (after splits)        2247          4.8%          0.5%  49.9%  44.8%
  All gprims:           14924   0.3%   2.8%   3.1%   5.5%  47.7%  40.2%
  ------------------------------------------------------------------
```

```
  Diced grids: 4985 total, 152.2 average points/grid
               0 extreme displacements succeeded, 0 rejected
Histogram of the number of points in each diced grid:
+------+------+------+------+------+------+------+------+
|<=  4 |<=  8 |<= 16 |<= 32 |<= 64 |<=128 |<=256 | >256 |
+------+------+------+------+------+------+------+------+
|  0.0%|  0.5%|  6.9%| 10.9%|  8.6%|  6.1%| 57.5%|  9.4%|
+------+------+------+------+------+------+------+------+
Shading:  758864 points, 4985 grids
  Memory per grid: average 743.9k, max 5985.1k
  Points per grid: average 152.2, min 6, max 289
  Shader instances (sices): reused 106, created 88, total 194
  Vertex instances (GSBs): reused 0, created 103 (+ one per sice)
  Executable instances (instances): reused 40949, created 107
Shader storage (proc -1):
  def storage: created 33, currently 33, peak 33 (7647K)
  sice storage: created 88, currently 1, peak 60 (1K)
  GSB storage: created 297, currently 1, peak 143 (16K)
  arg storage: created 674, currently 67, peak 329 (459K)
  tag storage: created 330, currently 34, peak 172 (3K)
  instance storage: created 107, currently 0, peak 55 (2443K)
  BOP storage: created 28717K, currently 5994K, peak 5994K
Texture filter width histogram (max of s and t widths):
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|   1 |   2 |   3 |   4 |   5 |   6 |   7 |   8 |   9 |  10 |  11 |>=12 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 0.0%|73.3%|10.4%| 4.1%| 2.8%| 3.2%| 1.8%| 0.8%| 0.6%| 0.5%| 0.4%| 2.2%|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

Trim curves / Trim data : Trim curves store the trim definition as defined with `RiTrimCurve` at parsing time. Trim data is generated at render time to determine which micropolygons are will be affected by trim curves.

Frustum box check : Geometric primitives undergo checks against the viewing frustum. Ones that lie completely outside the frustum are culled, while ones that are completely inside the frustum can be optimized against further checks.

Gprims processed and pushed forward :

In PRMan, geometric primitives are placed in a list attached to the first bucket where it will matter, based on which buckets its bounding box overlaps. The ”Gprims processed” number indicates the number of times a geometric primitive is taken out of a bucket list. If the primitive which is taken out of the list for a bucket can be determined to be invisible in that bucket, it is ”pushed forward” to a future bucket for future processing; the pushed forward count indicates the number of times this occurs.

Gprim Fates : This table summarizes what happens to geometric primitives created either in the initial model (created by a direct call to a function such as `RiPatchMesh`) or generated by a procedure (`RiProcedural`). The first line indicates what happens to these primitives before bucketing and splitting, while the ”after splits” line indicates what happens to primitives created by splitting other primitives. Primitives can end up being removed because of eye splitting (behind the camera), frustum culling (outside the viewing frustum), being hidden by other primitives, or because they’re backfacing. If they pass these tests, after bucketing they can be split further, or diced into micropolygons.

Shader (def, sice, gsb, arg, tag, instance) storage : These categories refer to storage needed for the internal data structures of the shader system.

Shader BOP storage : This refers to the memory allocated to hold all the variables and temporary registers for the shaders' run-time execution. The size is proportional to both the complexity of the shaders used, as well as the size of the largest grid.

1.5 Level 3 Statistics

Level 3 statistics add a detailed breakdown of the texture system as well as extra information about the hider.

1.5.1 Texture Statistics

PRMan uses a very efficient texturing system which brings in texture data from disk only when necessary. Texture data is brought in as tiles which are stored in a cache, and organized to optimize cache hit rates. Detailed information about the operation of the cache is printed in a table similar to the following:

```
Texture 0: texture0.tx
Texture 1: texture1.tx
Texture 2: texture2.tx
...
CPU TFILE CH     REFS    WIDE   SAME SEARCH  PROBE  FAULTS    RATE TILES PEAK
 -1 [   0 0]   195959       0   89.1  10.93   1.02      48   0.024  13.7   26
 -1 [   0 1]   195959       0   89.1  10.93   1.02      48   0.024  13.7   26
 -1 [   0 2]   195959       0   89.1  10.93   1.02      48   0.024  13.7   26
 -1 [   0 3]   195959       0   89.1  10.93   1.02      48   0.024  13.7   26
 -1 [   1 0]     4299K      0   78.4  21.56   1.01    1273   0.029 259.6  376
 -1 [   2 0]     4574K      0   83.3  16.74   1.01     452   0.010 100.3  167
 -1 [   3 0]     8994K      0   78.8  21.19   1.00    1411   0.015 322.3  465
 -1 [   4 0]     4808K      0   78.5  21.52   1.00    1182   0.024 249.7  376
 -1 [   5 0]     2155K      0   94.0   6.01   1.04    1460   0.066 283.7  399
 -1 [   6 0]     4018K      0   83.6  16.43   1.01     414   0.010  93.1  145
 -1 [   7 0]    35034       0   70.5  29.48   1.23     390   1.113 110.0  168
 -1 [   8 3]    21559    1248   75.9  24.09   1.12     105   0.487  25.8   47
...
Grand Total:    83740K 199853   80.3  19.74   1.02   24844   0.029 178.0  913
```

CPU : The processor ID. Only relevant in multiprocessor versions of the renderer (which at the moment do not exist).

TFILE : The ID of the texture file. The filename associated with the ID is printed before the table.

CH : The channel of the texture file.

REFS : The number of texture accesses made.

WIDE : The number of texture accesses made where the `s` or `t` width was "large".

SAME : The percentage of all pixel accesses which were found in the "current" (most recently accessed) texture tile.

SEARCH : The percentage of pixels which were not found in the currently loaded texture tile, thus requiring going to the cache. (Note: SAME + SEARCH = 100

PROBE : The average number of hash probes needed to find a texture tile in the cache, per search.

FAULTS : The number of cache faults (the needed texture tile was not in the cache during a pixel access).

RATE : The percentage of all pixel access that result in a cache fault.

TILES : The average number of tiles used by this texture channel (the average number of tiles in memory when a cache fault occurs).

PEAK : The maximum number of tiles in memory for this texture channel.

1.5.2 Hider Statistics

```
Hiding:
  Occlusion culling: 806 gprims, 10708 mps, 6904739 vps
  Micropolygons:
    505126 created (667118 base + 223 chopped - 162215 culled)
    Culled: 81636 frustum, 77756 backface, 2823 trim
    0 current (0 KB), 48909 peak (6674 KB)
    55360 allocated (7518 KB), 55360 free
    Trims: keep 6033, cull 2823, trim 740
    Pushes: 88419 forward (17.50%), 48986 down (9.70%)
            245980 far down (48.70%) (current 4164080 KB, peak 0 KB)
    Hittest success rate: 47.57% (8271590 of 17387889 samples)
   Trimmed Mps: 1110 hits, 545 misses
  Visible points:
    8271590 created
    0 current, 5565 peak, 5632 allocated (286 KB), 5632 free
    166779 opacity culled (2.04%)
Depth complexity histogram:
+------+------+------+------+------+------+------+------+------+------+
|<=  1 |    2 |    3 |    4 |    5 |    6 |    7 |    8 |    9 | >  9 |
+------+------+------+------+------+------+------+------+------+------+
| 98.1%|  1.7%|  0.2%|  0.0%|  0.0%|  0.0%|  0.0%|  0.0%|  0.0%|  0.0%|
+------+------+------+------+------+------+------+------+------+------+
Visible point list insertion histogram:
+------+------+------+------+------+------+------+------+------+------+
| 98.0%|  2.0%|  0.0%|  0.0%|  0.0%|  0.0%|  0.0%|  0.0%|  0.0%|  0.0%|
+------+------+------+------+------+------+------+------+------+------+
Visible point list depth histogram:
+------+------+------+------+------+------+------+------+------+------+
|<=  1 |    2 |    3 |    4 |    5 |    6 |    7 |    8 |    9 | >  9 |
+------+------+------+------+------+------+------+------+------+------+
|100.0%|  0.0%|  0.0%|  0.0%|  0.0%|  0.0%|  0.0%|  0.0%|  0.0%|  0.0%|
+------+------+------+------+------+------+------+------+------+------+
```

Micropolygon pushes : After primitives are diced and shaded, micropolygons are sometimes generated which do not belong in the current bucket; this can happen due to displacement or large motion blur. These are kept around and ”pushed” to the appropriate bucket in order to prevent the micropolygons from being reshaded. Pushing ”forward” means the micropolygon is stored in buckets on the right, while pushing ”down” moves the MP to a bucket immediately below. Pushing ”far down” means that the micropolygon was pushed down more than one row.

An excessive amount of micropolygon pushing can eat up a lot of memory. In the case of displacement, this can be alleviated by reducing excessively large displacement bounds which

causes the geometry to be diced far too soon. Alternatively, the `"extremedisplacement"` option (see Section 4.16 of the *PRMan* User Manual) can be used to save memory, albeit at the time expense of running the shader twice per primitive.

Hittest success rate : After micropolygons are shaded, they are tested against pixel subsamples to determine visible point information. "Trimmed Mps" count the number of micropolygons tested and culled against trim curves.

Depth complexity histogram : A visible point stores color, opacity, and depth information about micropolygons which overlap a single subpixel sample. Usually these lists are small (1 or so) after backface culling, although if there is transparency in the scene, the lists can be larger. The number of micropolygons per visible point is graphed in the histogram, before opacity culling takes place.

Depth complexity histogram :

A visible point stores color, opacity, and depth information calculated from a sampled point on a micropolygon. Each subpixel sample keeps track of a list of visible points, since a subpixel sample can overlap multiple micropolygons. Usually these lists are small (1 or so) after backface culling, although if there is transparency in the scene, the lists can be larger - these visible points will need to be composited together for the final subpixel value. The number of visible points per subpixel sample is graphed in the histogram, before opacity culling takes place.

Visible point list insertion histogram : After backface culling, visible point lists need to be sorted in depth before opacity culling can take place. This histogram measures the number of out-of-order visible points per subpixel sample which need to be placed back in the correct sorted order.

Visible point list depth histogram : This histogram graphs the number of visible points per subpixel sample, after opacity culling has taken place.

Chapter 2

A Practical Guide to Global Illumination

Larry Gritz,
Exluna, Inc.
lg@exluna.com

Abstract

This chapter describes several types of "global" lighting effects, and practical ways to either achieve or fake them using various RenderMan-compatible renderers.

2.1 Introduction: What is global illumination?

2.1.1 Nomenclature

People tend to get pretty pedantic in discussions about global illumination. So let me first explain some terms that I'll use in this chapter, and what I think they mean.[1]

Local illumination refers how light interacts with surfaces: given the light impinging right on the surface, what's the color of the light just leaving in a particular direction? A *local illumination model*, also known as a *bidirectional reflection distribution function* (or *BRDF*) is a mathematical formula describing this. **Surface shaders implement local illumination models** — they take the energy and direction of incoming light (`Cl` and `L`, per light) as input, and compute the amount of light (`Ci`) leaving in a particular direction (`-I`).[2]

Global illumination, simply put, is the set of lighting effects that require arbitrary knowledge of the scene. Surface shaders do not know or care (and for the most part cannot influence) *how* the light got to the surface, or what happens to it after it leaves. That's up to the renderer.

All renderers consider *direct lighting*: the light that travels in a straight line, unimpeded, from a light source to a surface point. In Heckbert's notation, these are `L(S|D)E` paths. Most renderers also have a way to compute *reflections* (the "coherent images" of the scene you see on shiny surfaces)

[1] Please don't email me to tell me that I've got terminology wrong. I've read all the papers, too. I've purposely written this chapter using terminology as it might be used by a TD, not somebody defending their dissertation on the topic.

[2] Chapter 9 of *Advanced RenderMan* has a good description of how this works in shaders, as well as a few specific useful local illumination functions and their implementation in SL. Matt Pharr's chapter in these course notes are all about particular kinds of local illumination models, specifically those involving subsurface scattering.

and *shadows* (blockage of direct lighting by other objects). These are global effects, of course, but they're basic and easily faked, so in colloquial usage "GI" usually refers to what I will call *indirect lighting*: effects of light bouncing between multiple surfaces. When those surfaces are highly specular (either reflective or refractive), they can form bright *caustics* — places where the light focuses.

Radiosity[3] is a method for compute GI that uses finite element techniques to solve for the steady-state energy balance of the scene. Because radiosity techniques were the only widely known GI techniques for many years, lots of people use the term "radiosity" when they should be saying "GI" or "indirect lighting." The actual finite element techniques are falling out of favor because (1) they generally have a hard time with specular light transport (in other words, they are only for `LD*E` paths); and (2) they require meshing of the entire scene and therefore don't scale well and are hard to use with certain types of geometry (like hairs or procedural models).

Most GI work these days uses *Monte Carlo* techniques. MC is a fancy way of saying that you're interested in computing something that's very difficult, but you know that the value is a weighted average of a function that you can easily compute at individual points. By carefully selecting your samples, you can come up with a good estimate for the function. In rendering, this usually means sampling a variety of directions to try to find out about the (very complex, continuous) pattern of light arriving at a particular surface. MC techniques don't require meshing of the scene, work well with oddball geometry (hair, LOD, procedurals, etc.), and scale with complexity much better than FE methods. In particular, they do not need the $O(n^2)$ (or $O(n \log n)$ if you do hierarchical clustering) interactions between surface patches to be considered, as do radiosity solutions.

Lately there has been quite a bit of interest in *image based lighting* (IBL), which allows you to supply detailed irradiance information to your local illumination functions without actually computing it (for example, by storing it in a texture map). It turns out that the results are rather lackluster with ordinary 8-bit texture maps; to really take advantage of IBL, you need *high dynamic range* (i.e., floating point) texture maps.

2.1.2 Nerdy Math

I'm not going to fill this chapter with equations. This chapter is meant for users of renderers. Everybody knows what I mean by "shadows," "reflections," "indirect illumination," and "caustics," so I'm just going to plainly explain how to achieve these effects. If you want to *implement* a renderer that supports these techniques, the end of the chapter will list references that contain all the gory details.

2.1.3 Why is global illumination so expensive?

It is widely known that *PRMan*'s (and other renderers') efficiency, both in terms of speed and frugal use of memory, is largely due to its aggressive pursuit of locality in its computations. In short, it's careful to "expand" (by reading, splitting, dicing, and shading) geometry only if and when it is needed, ruthlessly cull geometry that is off-screen or occluded, and reclaim memory as soon as it's done working on the area of the image where a piece of geometry overlaps. This is why *PRMan* must rely on texture mapping techniques such as shadow and environment maps, cannot ray trace, and makes no attempt to calculate any interreflected light.

It's very hard to write a ray tracer (let alone a true GI renderer) that's anywhere near as efficient, because the very nature of reflections, cast shadows, and interreflections preclude the aggressive culling of geometry not visible to the camera. Furthermore, in a ray tracer it's very hard to know when it's safe to throw away geometry – a reflection or shadow ray could be cast in any direction at any time, interacting with geometry in arbitrary parts of the scene.

[3] Actually, radiosity is a specific radiometric quantity, but here I use it to denote the technique that computes it.

The renderers that I've personally worked on cover all three choices for resolving this dilemma: *PRMan* supports no global effects whatsoever; BMRT supports lots of global effects by ray tracing everything, but is significantly slower and uses gobs more memory as a result; Entropy tries to compromise, using scanline methods when it can, and selectively ray tracing. If no ray tracing or GI is used, Entropy performance is extremely good, but the more global techniques that are used in a scene, the more expensive it will be.

2.1.4 Bias Disclosure

In previous course notes, I've tried to be exceptionally nonpartisan, but by the very nature of this topic – exploring effects that are right at the edge or even beyond traditional RenderMan capabilities – I think it's unavoidable to concentrate on renderers that support these features. I still try where possible to present solutions (or at least hacks) that should work on most other RenderMan-compliant renderers, but often there's no avoiding the need for a specific non-standard extension or special facility (like ray tracing).

In many cases I emphasize the way things are done in Entropy. This is not intended to slight PRMan, BMRT, or any other renderers. Rather, I have written extensively about those renderers before, and have little to add. Also, many of Entropy's new features and semantics more accurately reflect my current thinking about shading language design, so I present some new idioms that reflect this, rather than ugly code strewn with `#ifdef`'s that tries to be all things for all renderers.

I will also admit right from the start that this chapter is going to be quite light on the formality, instead concentrating on the dirty practicalities of achieving certain kinds of lighting effects in RenderMan-compliant renderers.

2.2 Shadows

Everybody thinks of shadows as being pretty basic, but they are actually global effects because they must consider other objects in the scene. For the purposes of this section, I'll be talking about shadows specifically for objects that occlude the direct illumination in the scene. The same techniques can be used for occlusion of indirect illumination.

2.2.1 Shadow Maps

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 2.1.

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 2.1 shows a simple scene with and without shadows, as well as the depth map that was used to produce the shadows.

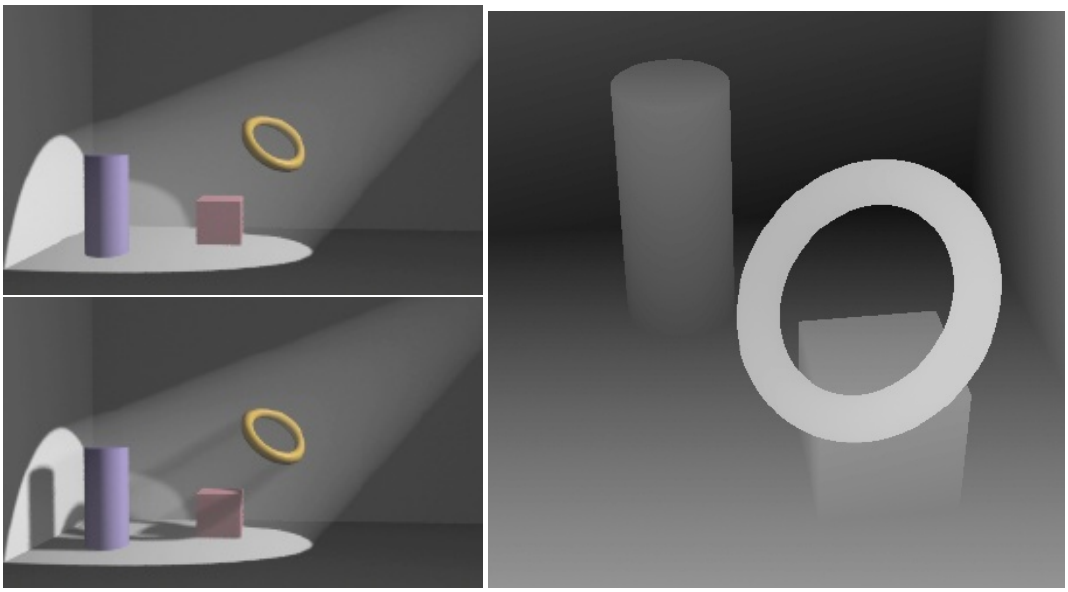


Figure 2.1: Shadow depth maps. A simple scene with and without shadows (left). The shadow map is just a depth image rendered from the point of view of the light source (right). To visualize the map, we assign white to near depths, black to far depths.

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 2.2). 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 2.3 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"`.

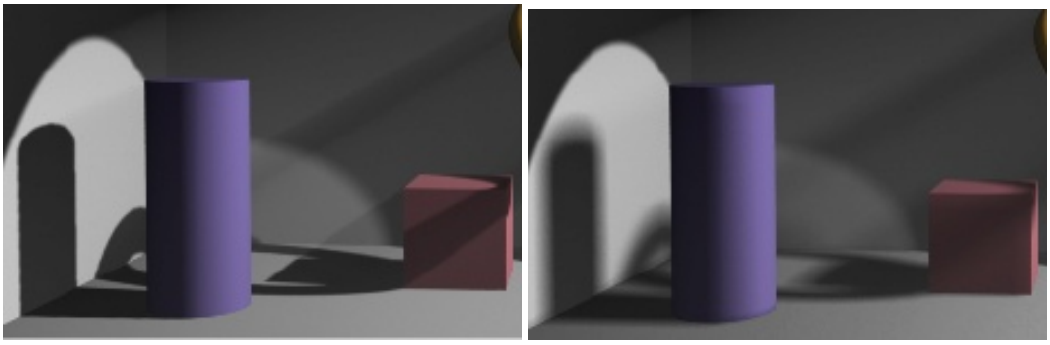


Figure 2.2: Adding blur to shadow map lookups can give a penumbra effect.

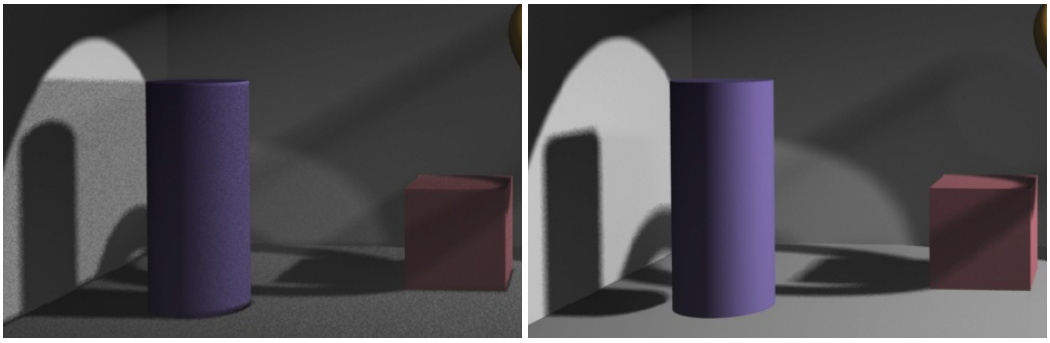


Figure 2.3: Selecting shadow bias. Too small a bias value will result in incorrect self-shadowing (left). Notice the darker, dirtier look compared to Figures 2.2 or 2.1. Too much bias can also introduce artifacts, such as the appearance of "floating objects" or the detached shadow at the bottom of the 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 2.1 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:

Listing 2.1 `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);
        Cl = atten * intensity * lightcolor;
        if (shadowname != "") {
            Cl *= 1 - shadow (shadowname, Ps, "samples", samples,
                              "blur", blur, "bias", bias);
        }
    }
}
```

- 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`. The equivalent command for either Entropy or BMRT is:

  ```
  mkmip -shadow shadow.z shadow.sm
  ```

2.2.2 Ray traced Shadows – BMRT

Some renderers may provide other shadow algorithms, which almost certainly will be specified in implementation-dependent ways. BMRT had two methods for specifying ray-traced shadows.

The older method was to have a special attribute that you'd set when you declared a particular light:

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

Somehow, the renderer would know to use ray casting to probe occlusion information in order attenuate the `Cl` generated by the light shaders, just before it returned control to the surface shader. I decided that this was not powerful enough (meaning that there weren't enough ways to hack it in the shader), so I added a non-standard built-in SL function:

```
color visibility ( point P0, P1 )
```

You could use this call inside your light source shader to determine the amount of occlusion between two points (presumably `P` and `Ps`), and modify `Cl` accordingly. Unfortunately, if you were writing a shader that had to work with both BMRT and PRMan (or using the "ray server"), this led to a rather ugly section of code:

```
#if (defined(BMRT) || defined(RAYSERVER))
    Cl *= visibility (P, Ps);
#else
    if (shadowname != "") {
        Cl *= 1 - shadow (shadowname, Ps, "samples", samples,
                          "blur", blur, "bias", bias);
    }
#endif
```

This worked fine but was not aesthetically pleasing, it required modification of any light shader that you might want to work with both shadow maps and ray tracing, and it also depended on another non-standard supposition — that `P` was set to the light position, even for an area light.

Luckily, we realized that a cleaner paradigm had been sitting right under our noses the whole time.

2.2.3 Combined Paradigm – Entropy

Entropy makes two minor semantic changes to the `shadow()` function:

- `shadow()` may return either a color or a float (much like `texture()` or `environment()`).
- If the texture name is `"shadow"`, it returns the occlusion as computed by ray tracing.

Thus, `shadowspot.sl` (Listing 2.1) requires *no modification* to make it either ray trace or use shadow maps. To make it ray trace, you just need to pass `"shadow"` as the map name, and everything else happens automagically. Even the optional parameters (such as `"bias"`) do approximately the analogous thing when ray tracing as when using a shadow map. (Those optional parameters with no obvious ray tracing analogue are simply ignored.)

We think that you'll find that with this minor semantic change allows almost any light shader that was written to work with shadow maps to be used with ray tracing as well, without modification, and maintaining perfect back-compatibility.

Entropy users should also be aware that by default, objects do *not* appear in ray traced reflections and shadows. You must specifically set attributes to tell the renderer that geometry must be retained and tested against rays:

```
Attribute "visibility" "integer shadow" [1]
Attribute "visibility" "integer reflection" [1]
```

2.3 Reflections

Many surfaces are polished to a sufficient degree that one can see coherent reflections of the surrounding environment. People often assume that mirror-like reflections require ray tracing. But not all renderers support ray tracing, and even those that do are often more efficient when using other techniques. 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, the 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.

2.3.1 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. The environment map can be completely synthetic (from six rendered images, or a painting), or captured from a real environment. An example of an "unwrapped" environment map is shown in Figure 2.4.

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:

- 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).

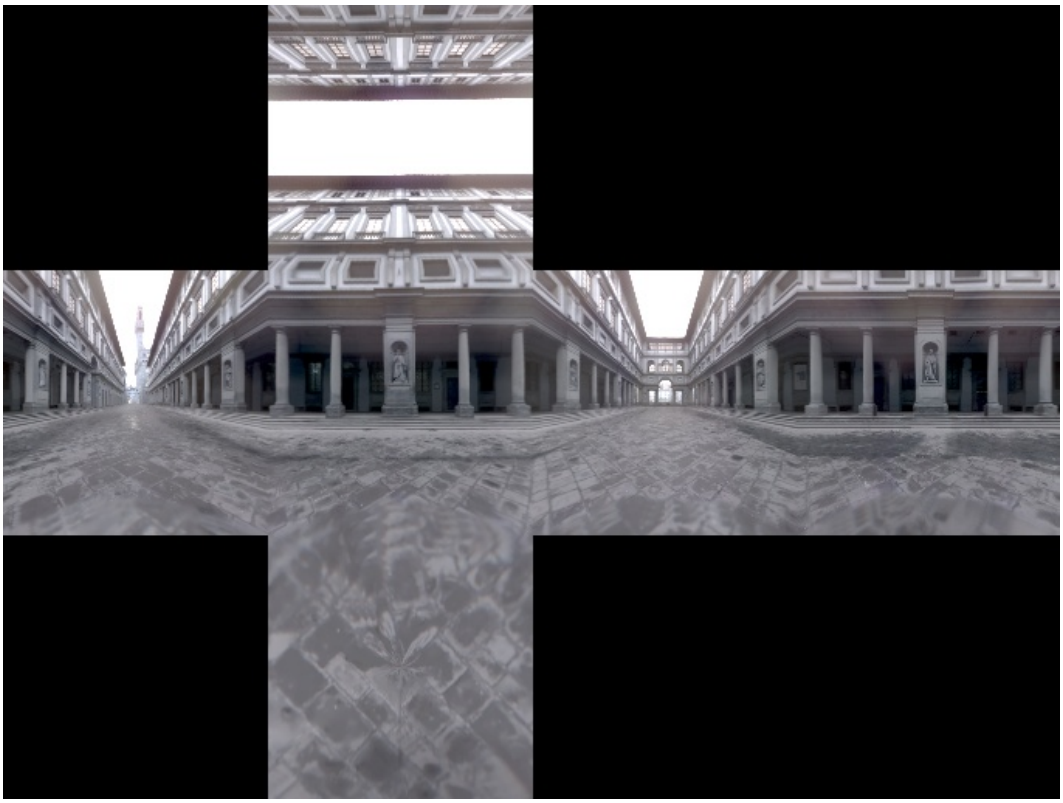


Figure 2.4: An example environment map generated from a "light probe" created by Paul Debevec (`www.debevec.org`).

- 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 `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 (see Listing 2.2).

2.3.2 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 2.5.

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:

```
/* Transform to the space of the environment map */
point Pndc = transform ("NDC", P);
float x = xcomp(Pndc), y = ycomp(Pndc);
Ct = color texture (reflname, x, y);
```

Listing 2.2 The `raysphere` function intersects ray `(E,I)` with a sphere of radius `r`.

```
/* 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;
}
```

2.3.3 Ray traced reflections

Shading Language has always had a `trace()` function:

```
color trace (point p, vector d)
```

In renderers that supported ray tracing (such as BMRT and its "ray server" mode), this function returns the color seen when looking in direction `d` from point `p`. In *PRMan* and other renderers that don't support ray tracing, the `trace()` function just returns 0.

2.3.4 Combined Paradigm

We would prefer to write our shaders so that we may use any of reflection maps, environment maps, or ray tracing, without needing to know which at the time we write the shader. I have given functions that do this in previous course notes and in the *Advanced RenderMan* book. After giving it a bit more thought, we would like to recommend a new idiom, shown in Listing 2.3.

It's easy to see how the `Environment()` routine deals with both environment and reflection mapping. What about ray tracing? The newest part of the idiom relies on a slightly changed semantic of the `environment()` call itself as we have implemented it in Entropy: if passed `"reflection"` as the name of the map, it does ray tracing. Furthermore, most of the optional environment map parameters (such as `"bias"`) are implemented analogously when performing ray tracing.

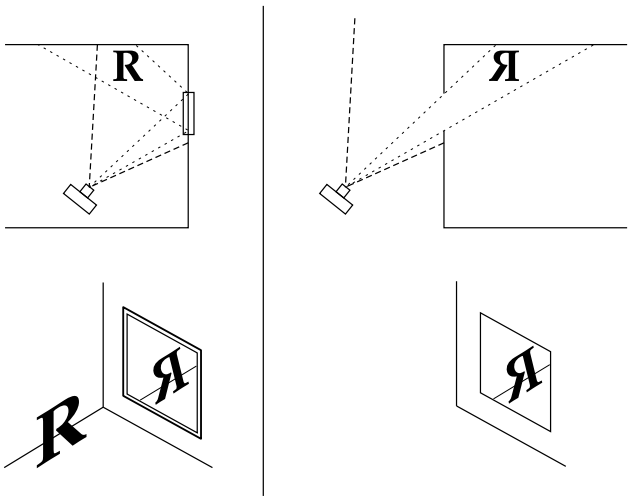


Figure 2.5: Creating a mirrored scene for generating a reflection map. On the top left, a camera views a scene that includes a mirror. Below is the image it produces. On the top right, the camera instead views the *mirror scene*. Notice that the image it produces (below) contains the required reflection.

The most useful optional ray tracing parameters are `"blur"` and `"samples"`. If you want blurry ray-traced reflections, there's no reason to write a loop with carefully selected samples (as we did when we presented the technique in *Advanced RenderMan*). Rather, just call `environment()` with an appropriate `"blur"` amount, exactly as you would if you were using an environment map file. The renderer does the rest. Remember, though, that they ray tracing will produce strange results if `envspace` is not `"current"` space!

Listing 2.3 `Environment()` combines environment mapping, reflection mapping, and ray tracing (using Entropy's semantic extensions) into one simple call.

```
/* Environment() - A does-all replacement for environment() lookups
 * (including parallax, if desired), flat reflection maps, and even
 * ray tracing (if supported).
 *
 * 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
 *
 * In ordinary circumstances, we just do environment mapping, assuming
 * that the environment map was formed in the named coordinate system
 * given by envspace.  If envrad != 0, we do "fake ray tracing" against
 * a sphere of that radius, in order to simulate parallax.
 *
 * If flat reflection mapping is desired, envspace must be "NDC" and
 * envname must be an ordinary texture map that's an image made with
 * the camera on "the other side" of the flat mirror.
 *
 * In the case of Entropy, we can also use this for ray tracing,
 * which just requires that envname=="reflection", envspace=="current",
 * and envrad==0.
 *
 * Warning -  the environment call itself takes derivatives, causing
 * trouble if called inside a loop or varying conditional!  Be cautious.
 */
color Environment ( string envname;  vector R;
                    string envspace;  uniform float envrad;
                    float blur, samples, Kr)
{
    extern point P;
    color C = 0;
    if (envspace == "NDC") {
        /* envspace "NDC" signifies the special case of a flat refl map */
        point Pndc = transform ("NDC", P);
        float x = xcomp(Pndc), y = 1-ycomp(Pndc);
        C = color texture (envname, x, y, "blur", blur);
    } else if (envspace != "") {
        vector Rsp = normalize (vtransform (envspace, R));
        if (envrad != 0) {
            /* Transform to the space of the environment map */
            point Psp = transform (envspace, P);
            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);
        }
        if (Kr > 0.0001)
            C = color environment (envname, Rsp,
                                   "blur", blur, "samples", samples);
    }
    return Kr * C;
}
```

2.4 Indirect illumination

It's very hard to be renderer-agnostic here. I just don't know a good way to compute indirect illumination with *PRMan*. As much as I'd like to present methods that will work with any RenderMan-compliant renderer, this is one area that is utterly dependent upon implementation-specific extensions. Thus, I will concentrate on BMRT and Entropy.

2.4.1 FE Radiosity in BMRT

For many years BMRT has had an implementation of radiosity with non-hierarchical finite-element meshing, progressive refinement, and ray-cast form factors. Finite element radiosity subdivides all of your geometric primitives into patches, then subdivides the patches into "elements." In a series of progressive steps, the patch with the most energy "shoots" its energy at all of the vertices of all of the elements. This distributes the energy around the scene. The more steps you run, and the smaller your patches and elements are, the more accurate your image will be (but the longer it will take to render). The radiosity steps are all computed up front, before the first pixel is actually rendered.

Finite element radiosity has some big drawbacks, almost all of which are related to the fact that it has to pre-mesh the entire scene. The technique actually worked reasonably well when it was first implemented in BMRT in 1992. But as the RenderMan world grew more sophisticated and gradually changed its tricks of the trade, the FE radiosity got a bit long in the tooth. Among the specific problems that began to crop up were the following:

- Our method would create patch and element hierarchies adaptively based on object size, but it would not "cluster." This meant, in particular, that objects made from many small polygons would make many too-small patches and elements, making the radiosity computations very expensive in both time and memory. More generally, taking time and storage proportional to the number of gprims seemed like it was unlikely to scale well with scene complexity.

- Back in 1992 when all RenderMan primitives were warped quads (such as `NuPatch` and `PatchMesh` primitives and quadrics), meshing for radiosity was simple. It was unclear how the FE radiosity would be extended to the various new primitives such as `SubdivisionMesh`, `Curves`, `Points`, and `Blobby`.

- I never did get the FE radiosity to work particularly well with CSG or trim curves, and it was hard to imagine it working well with level-of-detail models. The entire purpose of `RiProcedural`'s is defeated by FE radiosity's need to unpack and mesh all objects at the start of computation.

As a result, we now consider this technique to be depricated. It is still present in BMRT 2.6 but will almost certainly be removed for the next major release, and we left it out of Entropy entirely, in favor or MC methods.

2.4.2 MC Radiance Methods in Entropy & BMRT

Both Entropy and BMRT 2.6 and later support a new method for computing indirect illumination. It's a Monte Carlo technique based on Greg Ward's irradiance-based methods. Rather than enmeshing the scene and solving the light transport up front, the MC approach is "pay as you go." As it's rendering, when it needs information about indirect illumination, it will do a bunch of extra ray tracing to figure out the irradiance. It will save those irradiance values, and try to reuse them for nearby points.

The irradiance method can take much longer than FE radiosity for small scenes, but it scales better and should be cheaper for large scenes. It also addresses most of the concerns we had with

FE techniques, due mainly to the fact that the irradiance data is stored in a spatial data structure, decoupled from the explicit representation of the gprims themselves. This has several desirable consequences:

- The size of the irradiance database is proportional to the resolution and desired quality of the image, and is fairly independent of the number and type of gprims used in the scene.
- It has no trouble with primitives that would be tricky to mesh (such as `Blobby`'s or `Curves`), and easily handles trim curves, level of detail, and procedural primitives.
- The irradiance cache can be written and read from disk, potentially saving a huge amount of computation when rerendering a frame or when rendering a sequence of images (assuming no objects have moved). This was nearly impossible with the FE methods, since RenderMan frames are independent and it would be exceptionally difficult to match one frame's radiosity mesh with another frame's gprims.
- Though it has not been done at the time of this writing, the technique is much more easily extensible to handling light transmission through volumes.

To use the Monte Carlo irradiance calculations for global illumination, you need to follow the following steps:

1. Add a light source to the scene using the `"indirect"` light shader. If there are any objects that you specifically do not want indirect illumination to fall on, you can just use `Illuminate` to turn the light off, and subsequent surfaces won't get indirect illumination.
2. There are several options that control the behavior of the computations. See the list below for their description. You may need to adjust several of them on a per-shot basis, based on time/memory/quality tradeoffs.

Many of the options are related to the fact that it's ridiculously expensive to recompute the indirect illumination at every pixel. So it's only done periodically, and results from the sparse sampling are interpolated or extrapolated. Many options relate to how often it's done. Most of the settings are attributes so that they can be varied on a per-object basis. They are shown here with their default values as examples:

```
Attribute "indirect" "float maxerror" [0.25]
```

A maximum error metric. Smaller numbers cause recomputation to happen more often. Larger numbers render faster, but you will see artifacts in the form of obvious "splotches" in the neighborhood of each sample. Values between 0.1-0.25 work reasonably well, but you should experiment. But in any case, this is a fairly straightforward time/quality knob.

```
Attribute "indirect" "float maxpixeldist" [20]
```

Forces recomputation based roughly on (raster space) distance. The above line basically says to recompute the indirect illumination when no previous sample is within roughly 20 pixels, even if the estimated error is below the allowable maxerror threshold.

```
Attribute "indirect" "integer nsamples" [256]
```

How many rays to cast in order to estimate irradiance, when generating new samples. Larger is less noise, but more time. Should be obvious how this is used. Use as low a number as you can stand the appearance, as rendering time is directly proportional to this.

There are also two options that make it possible to store and re-use indirect lighting computations from previous renderings.

```
Option "indirect" "string savefile" ["indirect.dat"]
```

If you specify this option, when rendering is done the contents of the irradiance data cache will be written out to disk in a file with the name you specify. This is useful mainly if the next time you render the scene, you use the following option:

```
Option "indirect" "string seedfile" ["indirect.dat"]
```

This option causes the irradiance data cache to start out with all the irradiance data in the file specified. Without this, it starts with nothing and must sample for all values it needs. If you read a data file to start with, it will still sample for points that aren't sufficiently close or have too much error. But it can greatly save computation by using the samples that were computed and saved from the prior run.

You shouldn't use the `"seedfile"` option if the objects have moved around. But if the objects are static, either because you have only moved the camera, or because you are rerendering the same frame, the combination of `"seedfile"` and `"savefile"` can *tremendously* speed up computation.

Here's another way they can be used. Say you can't afford to set the other quality options as nice as you would like, because it would take too long to render each frame. So you could render the environment from several typical viewpoints, with only the static objects, and save all the results to a single shared seed file. Then for main frames, always read this seed file (but don't save!) and most of the sampling is already done for you, though it will redo sampling on the objects that move around. Does this make sense?

2.4.3 Illustrative Example

There was a big hubbub on the net this past year about a certain new renderer and the stunning images it made. It seemed to me that mainly these images were fairly straightforward global illumination on a completely diffuse scene with a single hemispherical light. For some reason, people seemed to think that support for such a lighting setup was exceptionally unique. So as an illustrative example, let's see how well we can reproduce the effect.

Chris Bond of Frantic Films sent me a tank model to use for such trials. Figure 2.6 showd what the model looks like with no lighting, just the RenderMan `defaultsurface` shader. The tank has 44483 polygons, the ground is a single bilinear patch, and the sky is a 1/2 sphere.

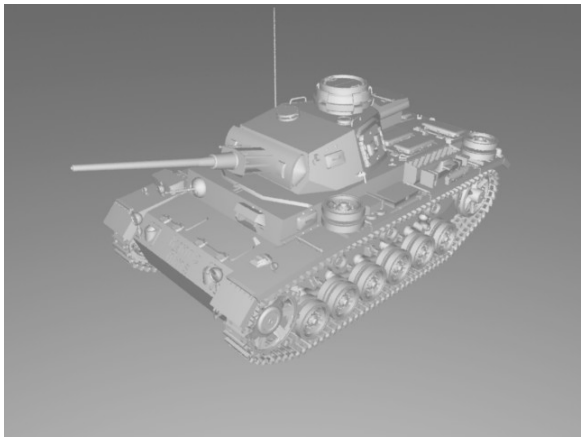


Figure 2.6: Tank, with `defaultsurface`. (Model courtesy of Chris Bond, Frantic Films.

For comparison, Figure 2.7 shows the scene rendered with a single light with ray cast shadows, and using a shadow map.

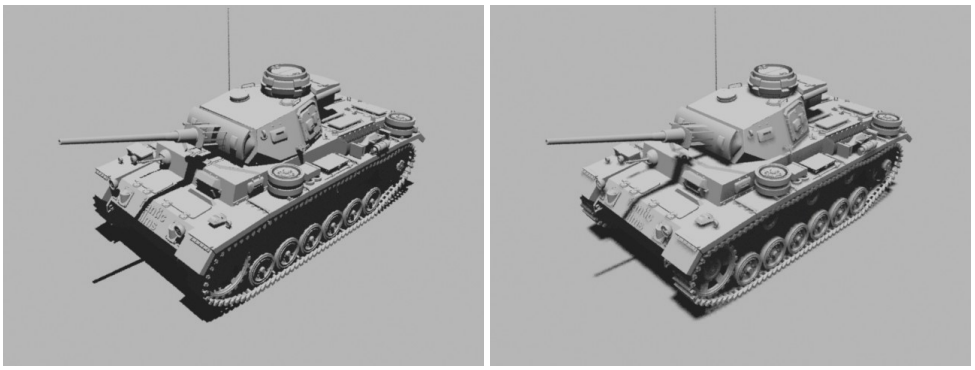


Figure 2.7: Tank with ray traced shadows (left) and shadow map shadows (right). (Model courtesy of Chris Bond, Frantic Films.

I rendered another image with the simple shadow map file (no radiosity, no area lights, no indirect illumination), but with a bigger blur on the shadow lookup. The results, shown in Figure 2.8 are surprisingly close to the higher quality methods, at a fraction of the cost, and could easily be done with *PRMan*.

Figure 2.8: Tank with very blurry shadow mapped shadows (and blue-tinted light). (Model courtesy of Chris Bond, Frantic Films.

I tried two more exotic methods for achieving the effect that everybody's been talking about. First, I made the sky an area light source, using

```
Attribute "light" "samples" [64]
```

The results are shown in Figure 2.9. I think that's pretty close to the look and feel of the highly-touted images, don't you?

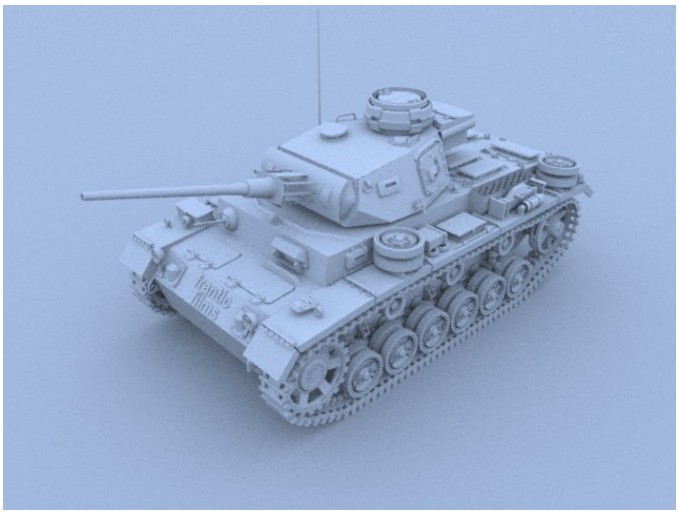


Figure 2.9: Tank lit by a hemispherical area light. (Model courtesy of Chris Bond, Frantic Films.

But wanting to even more faithfully reproduce the look and get the full effect of indirect illumination, I tried using the radiance. Instead of making the sky a big light, I made the sky just be constant colored, then used the new irradiance calculations with the following parameters:

```
Attribute "indirect" "float maxerror" [0.05]
Attribute "indirect" "float maxpixeldist" [5]
Attribute "indirect" "integer minvalid" [3]
Attribute "indirect" "integer nsamples" [256]
```

Figure 2.10 shows the final results.

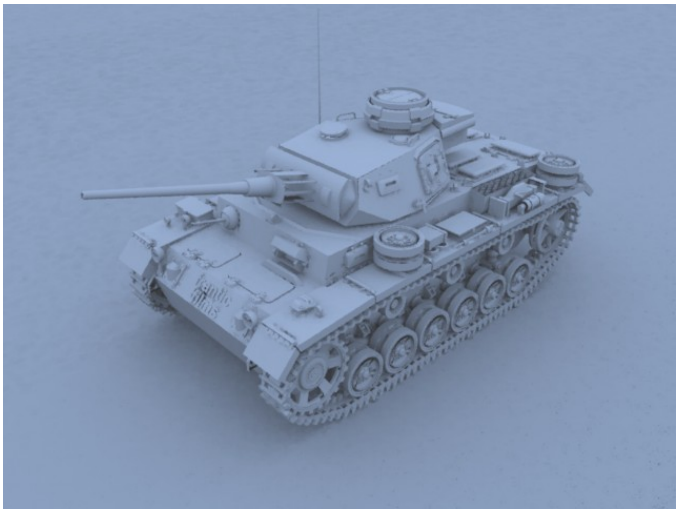


Figure 2.10: Tank with true indirect lighting and an emissive hemispherical dome. (Model courtesy of Chris Bond, Frantic Films.

2.5 Caustics

Caustics are bright spots caused by the focusing of light that is refracted or specularly reflected, particularly by curved objects. I will discuss two ways to generate caustics — one very fake method that will work with any renderer, and also the computation of true caustics by GI renderers.

2.5.1 Fake caustics

Perhaps even more so than refractions or shadows, it is very hard for the viewer to reason about the correct appearance of caustics, and therefore potentially easy to fake. A rather cartoonish understanding of caustics might be as follows: in the interior where a cast shadow would ordinarily be, we see a bright spot. Can we mimic the appearance without simulating the phenomenon?

Figure 2.11: A plastic vase (left) and a glass vase (right).

Let's start with a concrete example. Figure 2.11 shows a vase made of plastic, and then changing the material properties to that of glass. We can see that when the vase is made of glass, the shadow seems too "dense" – we expect some of the light to make it through the class anyway, but distorted and perhaps concentrated in the center.

Observe that in our `shadowspot` shader (Listing 2.1) we blocked light by using a shadow map lookup:

```
Cl *= 1 - shadow (shadowname, Ps, "samples", samples,
                  "blur", blur, "bias", bias);
```

If `1 - shadow()` is the amount of light that gets through, then a light that multiplied by `shadow()` instead would illuminate only the occluded region. To restrict ourselves to the interior of the occluded region, we can blur the boundaries of the shadow map lookup and threshold the results:

```
float caustic = shadow (shadowname, Ps, "samples", samples,
                        "blur", blur, "bias", bias);
caustic = smoothstep (threshold, 1, caustic);
Cl *= caustic;
```

We apply the caustic light only to the floor, using the very same shadow map that our ordinary spotlight was using to create the shadows. The results can be seen in Figure 2.12.

Figure 2.12: Our first attempt at fake caustics, using a special light that only shines in the interior of occluded regions.

We can also add some noise to spruce it up, yielding the final image in Figure 2.13. The shader for the light is given in Listing 2.4.

2.5.2 Real caustics – Entropy/BMRT

BMRT and Entropy support "real" caustics, computed by the renderer by simulating the action of light with *photon mapping*. This involves several steps:

1. Declare a light source with the `"caustic"` shader (like `"indirect"`, it really a hook into various renderer internal magic). You should use `RiIlluminate` to turn the caustic light on for objects that receive caustics, and turn it off for objects that are known to not receive caustics. Illuminating just the objects that are known to receive caustics can save lots of rendering time.

2. For any light sources that should reflect or refract from specular object, thereby causing caustics, you will need to set the number of photons with:

   ```
   Attribute "light" "integer nphotons"
   ```

   This sets the number of photons to shoot from this light source in order to calculate caustics. The default is 0, which means that the light does not try to calculate caustic paths. Any nonzero number will turn caustics on for that light, and higher numbers result in more accurate images (but more expensive render times).

3. The algorithm for caustics doesn't understand shaders particularly well, so it's important to give it a few hints about which objects actually specularly reflect or refract lights. This is done with

   ```
   Attribute "radiosity" "specularcolor"
   ```

Listing 2.4 `causticlight.sl`: a light source that shines caustics.

```
light
causticlight ( 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;
               float  threshold = 0.5;
               float  noiseamp = 0, noisefreq = 1, noisepow = 1;
    )
{
    uniform vector axis = normalize(to-from);

    illuminate (from, axis, coneangle) {
        float cosangle = (L . axis) / length(L);
        float atten = pow (cosangle, beamdistribution) / (L . L);
        atten *= smoothstep (cos(coneangle), cos(coneangle-conedeltaangle),
                             cosangle);
        Cl = atten * intensity * lightcolor;
        if (shadowname != "") {
            float caustic = shadow (shadowname, Ps, "samples", samples,
                                    "blur", blur, "bias", bias);
            caustic = smoothstep (threshold, 1, caustic);
            if (noiseamp != 0) {
                point PL = transform ("shader", Ps);
                caustic *= noiseamp * pow (noise(PL*noisefreq), noisepow);
            }
            Cl *= caustic;
        }
    }
}
```

```
Attribute "radiosity" "refractioncolor"
Attribute "radiosity" "refractionindex"
```

4. Finally, you may want to adjust several global options that control basic time/quality tradeoffs. These are also described below.

`Attribute "caustic" "float maxpixeldist" [16]`

Limits the distance (in raster space) over which it will consider caustic information. The larger this number, the fewer total photons will need to be traced, which results in your caustics being calculated faster. The appearance of the caustics will also be smoother. If the maxpixeldist is too large, the caustics will appear too blurry. As the number gets smaller, your caustics will be more finely focused, but may get noisy if you don't use enough total photons.

`Attribute "caustic" "integer ngather" [75]`

Sets the minimum number of photons to gather in order to estimate the caustic at a point. Increasing this number will give a more accurate caustic, but will be more expensive.

There's also an attribute that can be set per light, to indicate how many photons to trace in order to calculate caustics:

`Attribute "light" "integer nphotons" [0]`

Sets the number of photons to shoot from this light source in order to calculate caustics. The default is 0, which means that the light does not try to calculate caustic paths. Any nonzero number will turn caustics on for that light, and higher numbers result in more accurate images (but more expensive render times). A good guess to start might be 50,000 photons per light source.

The algorithm for caustics doesn't understand shaders particularly well, so it's important to give it a few hints about which objects actually specularly reflect or refract lights. These are controlled by the following attributes:

```
Attribute "caustic" "color specularcolor" [0 0 0]
```

Sets the reflective specularity of subsequent primitives. The default is `[0 0 0]`, which means that the object is not specularly reflective (for the purpose of calculating caustics; it can, of course, still look shiny depending on its surface shader).

```
Attribute "caustic" "color refractioncolor" [0 0 0]
Attribute "caustic" "float refractionindex" [1]
```

Sets the refractive specularity and index of refraction for subsequent primitives. The default for `refractioncolor` is `[0 0 0]`, which means that the object is not specularly refractive at all (for the purpose of calculating caustics; it can, of course, still look like it refracts light depending on its surface shader).

So for the case of the vase image, I modified the scene in the following way:

1. I added a "caustic" light to the scene:

   ```
   LightSource "caustic" 99
   ```

   The light is turned on (via `Illuminate`) for the floor.

2. I marked the vase as specularly refractive (i.e., causing caustics):

   ```
   Attribute "caustic" "color refractioncolor" [1 1 1]
   Attribute "caustic" "float refractionindex" [2]
   ```

   I've fudged a bit here — the true physical values would be for the refraction color to be less than 1, and the refraction index should really be about 1.5. I adjusted the numbers to get a more pleasing appearance.

3. The spot light is marked with an attribute that marks it as generating photons that potentially cause caustics:

   ```
   Attribute "light" "integer nphotons" [40000]
   ```

The final image using this technique is shown in Figure 2.14. You can be the judge of whether or not the results are more pleasing than the "fake" method. I think that in this example, I like the fake technique better. But for many scenes more complex than this one, the fake method would be impossible to do convincingly.

Figure 2.13: Fake caustics, including noise.

Figure 2.14: “Real” caustics.

2.6 HDRI / Environment lighting

Many people have been interested in *image-based lighting*, which means using a computed or captured environment map that contains the irradiance information for a scene. This kind of technique is particularly handy for inserting synthetic objects into real scenes or lighting environments. To have a remotely accurate representation of a scene's lighting, integer values of 0-255 will not do. Instead, to achieve a reasonable quality level, a *high dynamic range* representation is needed, which really means that you need some kind of floating point environment map.

Much of these techniques were developed by Paul Debevec. Please see the references at the end of this chapter, as well as his web site `www.debevec.org` for more thorough information.

2.6.1 Acquiring "Light Probes"

Just to show how easy it is to acquire HDRI light probes, I tried it in Exluna's offices. Here is the procedure I followed:

1. Figure 2.15 shows my naively simple setup: a polished metal "gazing ball" that I ordered from a garden catalog, a cardboard mailing tube, a Nikon Coolpix 880 digital camera (street price: $650), and a tripod. I set up the gazing ball on its stand in the middle of the room, with the camera pointed at it on a tripod.

Figure 2.15: The Nikon CP 880 camera and LG making a light probe at Exluna, Inc.'s Worldwide Headquarters. Photographed by Doug Epps.

2. I took four images of the reflective sphere at different exposures: 1/1000, 1/250, 1/60, and 1/15 seconds. All four exposures used the same aperture: f/4. I converted the images to linear space (the Nikon 880 writes 8-bit JPEG images with a gamma of 2.2), then registered and cropped the four images by hand using `gimp`. Figure 2.16 shows the four images.

Figure 2.16: Linearized, registered exposures of 1/1000, 1/250, 1/60, and 1/15 seconds.

3. I used Paul Debevec's `mkhdr` program (freely available — see `www.debevec.org`) to convert the four exposures into a single high dynamic range image, saved as a floating-point TIFF file. It doesn't do much good to print that image here – its very nature as a high dynamic range image precludes its faithful reproduction in print or on a computer screen.

4. Using Entropy's `mkmip` texture map creation program, I converted the "image of a sphere" floating-point TIFF into a cube-face environment map:

```
mkmip -lightprobe exlunahdr.tif exluna.env
```

   The resulting environment map is shown in Figure 2.17.

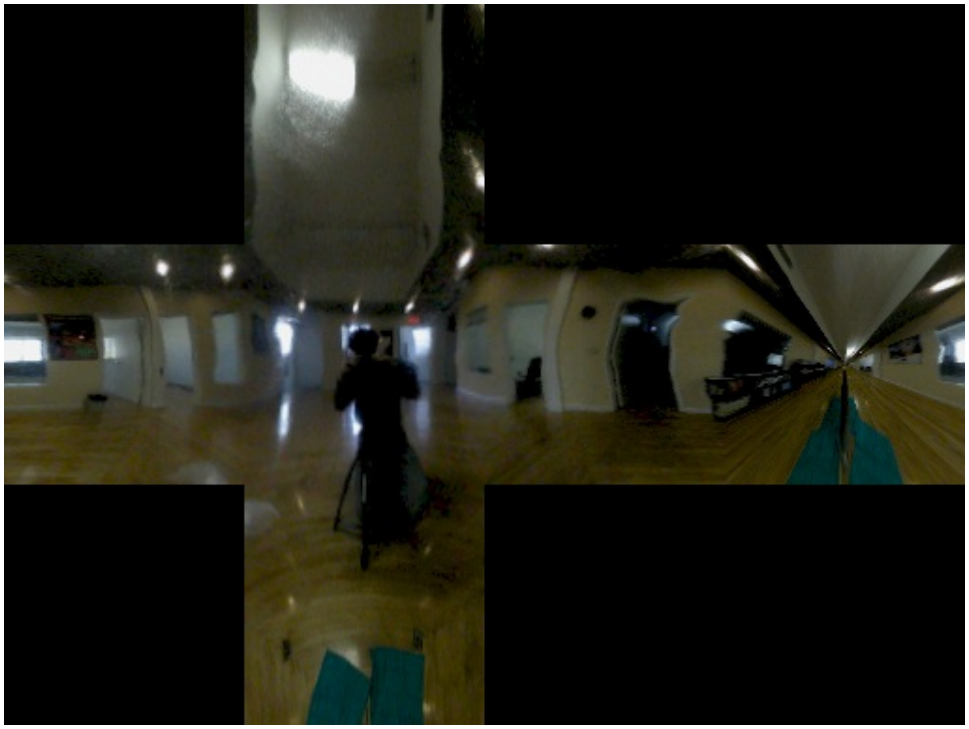


Figure 2.17: Our HDR spherical image converted to a cube-face environment map.

The gist of the multiple-image technique is that any one image that is captured by a digital camera or scanned stores its values in the 0-255 range. Pixels near the middle of the range are

accurate, but pixels at the bottom of the range are probably noisy and degraded by quantization, while pixels at the high end of the range may be saturated. The `mkhdr` program combines several images at different exposures, hoping that any particular pixel will be properly exposed for at least one of them. The combined image should be an accurate linear HDR image.

Paul Debevec's light probes have many advantages over mine: they are carefully constructed so as not to contain an image of the photographer, and his silver ball is obviously less dented than mine (judging by the warping in my environment map). Bear in mind that I spent a total of about 3 hours and $50 figuring out how to create light probes. I'm sure that with somewhat better equipment, a little practice, and some custom software, this could be much more of a science. In any case, having proven that I could create a light probe if I needed to, for subsequent experiments I used Paul Debevec's light probe of the Uffizi gallery in Florence (seen in Figure 2.4).

2.6.2 Using HDRI

I could think of a number of ways to try using HDRI-based lighting in our renderers of choice.

Method 1 – Broad Solar Lights

In most renderers, including *PRMan*, you could do an extremely "poor man's HDRI" using a light with a "broad solar" statement:

```
light hdri1 (string envname = "")
{
    solar () {
        if (envname != "")
            Cl = environment (envname, -L, "blur", 0.5);
        else Cl = 0;
    }
}
```

The `solar` statement, when given no arguments, will choose its own `L` in a "good" direction. Generally, this means in the reflection direction. So what we're doing here is making a light that simply does a very blurry environment map lookup in order to find its color. This is better than nothing, but it's quite inaccurate. Unfortunately, without area lights, GI, or explicit support for HDRI, that's approximately the best we can do. The biggest problem is that there's no way to take occlusion into account — shadow maps can only shadow from a point (not from sampled points on a hemisphere), and *PRMan* doesn't ray trace.

Method 2 – Brute Force

If you have a renderer that has no ray tracing or area lights, but you have cycles to burn, you could make a huge number of lights, distributed over the hemisphere. Each light could be a simple `distantlight`, pointing in toward the center and with its color set by looking up its position in the HDRI map. Each light could have its own shadow map.

I didn't try this myself, as it seemed that you'd probably need dozens or hundreds of lights. It would almost certainly be more expensive than using a ray tracer. But it should give the desired effect.

Method 3 – Area Light Sources

One method that I could think of, that would work in any renderer supporting true area lights, would be to make a hemispherical area light source with a shader that used `-L` to index into the HDRI environment map. Such a shader is below:

```
light
hdri2 (float intensity = 1;
       color lightcolor = 1;
       string envname = "";
       string envspace = "world";
       float envblur = 0.5;
       string shadowname = "";
       float shadowblur = 0.001;
       float shadowbias = 0.01;
       float shadowsamples = 1;)
{
    vector Lenv = vtransform (envspace, P-Ps);
    illuminate (P) {
        if (envname != "")
            Cl = environment (envname, Lenv, "blur", envblur);
        else Cl = 0;
        if (shadowname != "")
            Cl *= 1 - color shadow (shadowname, Ps, "samples", shadowsamples,
                                    "blur", shadowblur, "bias", shadowbias);
    }
}
```

Results were exceptionally noisy when I used “low” sampling rates for the area light (for example, 16 area light samples).

Presumably, the horrible noise is because the HDRI image has much of its energy concentrated in a very small area, and thus uniform sampling of the area light hemisphere is prone to noise since so few samples will tend to fall into the bright region. Images with a direct image of the sun (which subtends only 1/2 degree) would be especially noisy. Increasing the number of area light samples to 256 made the image much smoother, but at a huge expense.

Method 4 – Emissive Dome for GI

We can observe that the irradiance from the HDRI map is of very low spatial frequency on screen — in other words, it doesn’t change much spatially. For BMRT or Entropy (or presumably any other renderer that supports its methods of indirect lighting), we can take advantage of the irradiance caching. Instead of an area light, we make a scene with no lights. The scene is surrounded by dome that’s emissive – in other words, its color is taken by looking up from the environment map, without consulting any light sources (just like we did for the dome lighting earlier). Here’s the shader I used:

```
surface envsurf (string envname = "", envspace = "world")
{
    if (envname != "")
        Ci = environment (envname, normalize(vtransform(envspace, I)));
    else Ci = Cs;
}
```

We make sure to render with indirect illumination turned on. The main advantage here is that because the renderer calculates the indirect illumination sparsely, we can afford to have a huge number of samples (thus reducing the noise). In other words, with an area light, 16 light samples are too noisy; 256 light samples looks good, but is too expensive to compute for every shading sample. But with the irradiance method, we can afford to use 256 or more samples, because we’re only recalculating it, say, every 100 pixels.

Method 5 – Entropy HDRI light

Shameless plug: If you're using Entropy, there's a "built-in" `environmentlight` that *just works* (much like the built-in `caustic` and `indirect` lights do specific tasks that would be difficult to do in shaders). It's sort of like an area light, but it needs no geometry. You still must choose the right number of samples, but it's significantly lower than a "dumb" area light. Basically, it analyzes the environment map to assure that it is sampled in the right directions, thus helping to ameliorate the sampling problems resulting from the extremely uneven distribution of energy across the hemisphere.

Figure 2.18: Teapots lit only with a single HDRI map.

2.7 References

Many of you will be curious about the math behind these techniques, and how they can be implemented in various renderers. Unfortunately, the gory details are beyond the scope of these course notes, and too detailed and extensive to fit into a 1-hour talk. But for those of you who are curious, following are some good references:

The classic papers for finite-element radiosity are: [Goral et al., 1984], [Cohen et al., 1988], [Wallace et al., 1989], [Hanrahan and Salzman, 1989], [Smits et al., 1994].

The Monte Carlo techniques used in BMRT and Entropy to compute indirect illumination is a variant of those described in [Ward et al., 1988], [Ward and Heckbert, 1992], and [Ward, 1994].

Photon mapping, used by BMRT and Entropy for caustics are described in [Jensen, 1996] and [Jensen and Christensen, 1998]. There was also an excellent SIGGRAPH 2000 course notes on implementing photon mapping.

HDRI and environment lighting are described in: [Debevec and Malik, 1997] and [Debevec, 1998]. Paul Debevec is organizing a course this year on the topic, and I advise purchasing the course notes, which I'm sure go into much more detail on how to create and use light probe images.

Acknowledgements

Chris Bond of Frantic Films provided me with the tank geometry.

Paul Debevec graciously allowed me to use his light probe image of the Uffizi. His light probe collection can be found at `www.debevec.org`.

I'd like to thank the rest of the Entropy development team: Craig Kolb, Doug Epps, and especially Matt Pharr, who along with myself co-implemented much of the indirect and caustic code in both BMRT and Entropy.

Finally, I'd like to thank Tony Apodaca, my coauthor, former boss, and sounding-board for interface issues. Significantly more than any other single individual, Tony was responsible for the adoption of the RenderMan interface and widespread use of *PRMan* throughout the industry. We all owe you one, Tony.

Chapter 3

Layered Media for Surface Shaders

Matt Pharr
Exluna, Inc.
mmp@exluna.com

This section of the course will describe techniques for modeling the appearance of surfaces with layered structure in shaders. This kind of layered structure is often present in real-world objects– examples that we will discuss include paint, where the color of the base-coat may affect the overall color in spite of differently-colored coats painted on top of it; skin, which can be modeled as a stack of interacting scattering layers; and weathering effects, where the appearance of base material changes over time as dust and dirt are deposited on it. More generally, we'll describe ways of decomposing complex surfaces into parts that can be described independently and ways of combining these parts together to compute realistic shading.

An important distinction can be made between the *texturing* part of a surface shader and the *reflection* part of it. By texturing, we mean computing values that describe surface parameters at a point on a surface. For example, determining what color paint is present at a surface point, how much oil an oiliness map says is on the skin, etc. These texturing values may come from 2D texture maps, 3D paint, noise functions, etc. These notes will focus some problems from on the other half of surface shading: reflection, or how to take that texturing information and write a shader so that the surface you're describing reflects light in a way that it really looks like skin, paint, etc. A number of methods can be used to describe these complex scattering effects in surface shaders, with varying levels of computational complexity. We'll describe a number of them, drawn from both recent and classic work in the research literature.

The higher-level goal of this chapter is to start to provide a set of shading language building blocks that can be used for describing all sorts of layered surfaces. Just as operations like `noise()`, `texture()`, and `smoothstep()` are part of the standard toolbox for describing material properties ("what color is the wood at this point", "is there dirt on the surface at this position", etc.), I hope that the toolbox for modeling complex surface reflection with layers that this chapter provides will also be generally handy for a wide variety of surfaces.

3.1 The Kubelka Munk Model

An early approach to the problem of modeling the appearance of layered surfaces was developed by Kubelka and Munk starting in the 1930s [Kubelka and Munk, 1931, Kubelka, 1948, Kubelka, 1954]. They considered the problem of determining the resulting overall color from a set of layers of differently colored paints. For example, consider a white wall with a coat of blue paint on top of it. For

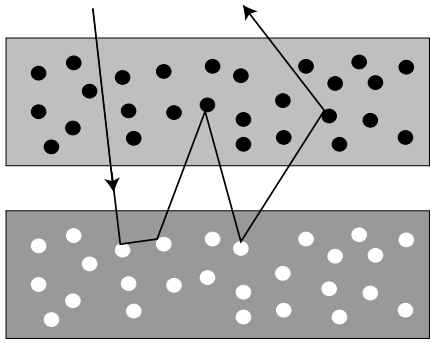


Figure 3.1: Light scattering within layers of colored paint. As light enters the top layer, it interacts with particles in that layer, causing some of it to be absorbed and some to be scattered. Some of the remaining light is then scattered out of the top of the layer and the rest is transmitted through to the bottom layer, where it interacts with particles there. This process continues until the incident light has either been absorbed or scattered back out of the medium. In order to compute the final distribution and color of light that leaves the top back toward the viewer, it's necessary to model all of these light interactions.

thin layers of blue paint, the resulting color will be a very light blue, but as the thickness increases, the resulting color will approach to the color of the blue paint. As such, the color of the underlying wall starts out having a large contribution to the overall color for thin added layers of blue paint, but given a sufficiently thick layer on top of it, the base color effectively has no more impact.

Media like paint can be modeled volumetrically as collections of small colored particles. As light passes through the medium, some of it misses all of the particles, and exits with reduced intensity due to absorption in the medium. The rest of the light interacts with one or more particles in the medium, and is scattered in a new direction. The interactions with the particles cause some wavelengths of the light to be absorbed, so that the particles' colors change the color of the light that hits it. Because there are so many particles, these interactions are often modeled statistically, rather than as individual interactions. (See Gondek et al's paper for a more detailed, simulation-based approach to modeling paint reflection [**?**].)

In order to model these situations accurately, it's necessary to both model the scattering of light within each layer as well as the scattering between the layers–see Figure 3.1. In its most general setting, solving this problem is analogous to solving the rendering equation to model global light transport in a geometric scene; all the same issues of modeling multiple interactions of light come into play. In Section 3.3, we'll come back to this theme with a more detailed simulation.

Kubelka and Munk, however, made a number of assumptions about the details of these interactions in order to develop a model that could be solved analytically:

- They assumed that the composition of each layer was *homogeneous*; i.e. having uniform color and scattering properties throughout.
- They ignored the directional distribution of light: specifically they assumed that the top layer was irradiated by incident light uniformly from all directions, and that light exiting each layer after scattering was also uniformly distributed.

The homogeneity assumption is only somewhat limiting–the composition rules described below in Section 3.1.4 help sidestep it when it's not an accurate assumption. However, the assumptions about light distribution are more serious. Uniform incident illumination is rare in practice (except for outdoors on an overcast day, for example); a uniform exiting distribution is not physically plausible but is not too far off the mark for objects that are not very glossy.

By making the above assumptions, Kubelka and Munk were able to compute an analytic solution to this limited multiple scattering problem between all of the layers. The resulting model has been found to be useful in a variety of scientific fields and is widely used due to its predictive power. It can be extended to model scattering from more complex objects, and some of the ideas behind it provide a basis for the more sophisticated approaches described in Section 3.3.

Hasse and Meyer first introduced their model to computer graphics. They demonstrated that it was a much better model for pigmented materials that both reflect and transmit light than previous models based on additive (RGB) or subtractive (CMYK) color spaces [Haase and Meyer, 1992]. The model was later used by Dorsey and Hanrahan to model light scattering from weathered surfaces [Dorsey and Hanrahan, 1996]. See also Glassner for an overview and derivation of some of the equations [Glassner, 1995].

3.1.1 Why not use compositing?

The most common approach to modeling layered surfaces in shaders is to use alpha blending, as described by Porter and Duff in the context of image composition [Porter and Duff, 1984]. Consider simulating tarnish on a copper surface: standard illumination models would be used to compute a color for pure tarnish and for the original copper surface, and the two are blended according to the thickness of the tarnish. Taking advantage of `material.h` from ARM, one might write:

```
...
color base = MaterialRoughMetal(Nf, copperColor, Ka, Kd, Ks, roughness);
color tarnish = MaterialMatte(Nf, tarnishColor, Ka, Kd);
Ci = mix(base, tarnish, tarnishLevel);
```

where `tarnishLevel` is determined empirically from some notion of how thick the tarnish is at the point being shaded.

This approach often works perfectly well and can create convincing images. However, it has a number of drawbacks grounded in its lack of physical accuracy:

- It doesn't accurately model the effect of varying thickness. For example, doubling the layer of the tarnish layer doesn't double its influence on the final color computed. Rather, the effect of increasing layer thickness is non-linear, as once it reaches sufficient thickness the base layer is no longer visible and additional thickness has no effect. While this particular drawback can be fixed by clamping the maximum `tarnishLevel` to one; it's a symptom of a deeper problem.

- It doesn't account for what happens to light as it passes through the tarnish. For example, the tarnish will likely cause the light passing through it to become diffused; light entering the top layer is likely to exit the bottom with a random direction (see Figure 3.2). The result of this is that specular highlights from the bottom layer should be gradually washed out or eliminated, since much less light would be coming in from the directions that caused specular highlights to be generated.

In short, because it completely sidesteps the problem of how light scatters between the two layers, the results of varying the parameters aren't intuitive.

3.1.2 The Parameters

Kubelka and Munk assumed that three parameters that describe the low-level scattering properties of each layer are known. These are:

- d, the physical thickness (or depth) of the layer

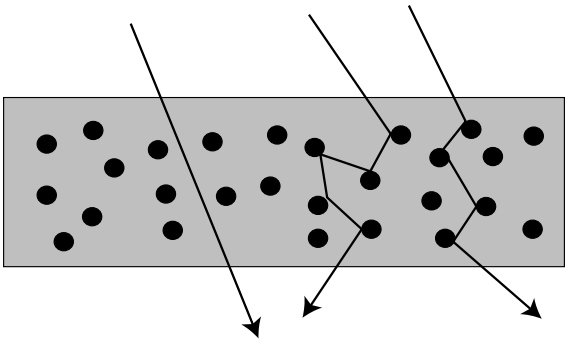


Figure 3.2: Light that passes through a transmissive layer tends to lose its directional distribution due to random scattering as it interacts with particles in the layer. As a result, the incident light arriving on the bottom layer tends to have a much more uniform distribution. This causes, for example, specular highlights to wash out and disappear.

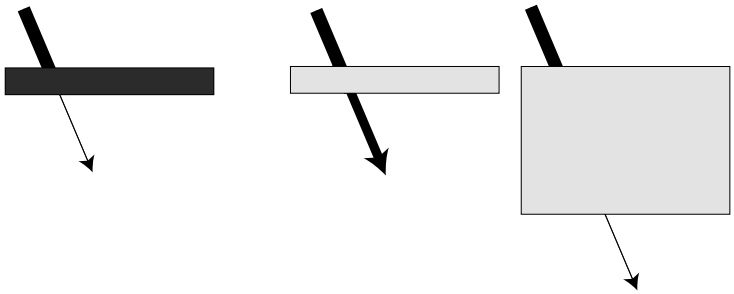


Figure 3.3: Attenuation through layers of varying thicknesses and attenuation coefficients. Left: a thin layer with a high attenuation coefficient transmits very little light, while (middle) a similarly thin layer with low attenuation coefficient transmits almost all of the incident light. Right: a very thick layer with low attenuation coefficient may transmit as little light as a thin layer with a high attenuation coefficient.

- σ_a, the attenuation coefficient of the layer,[1] and
- σ_s, the scattering coefficient of the layer

The meaning of the depth parameter is obvious; the other two are only slightly little more complicated.

The attenuation coefficient describes how much light is absorbed by the layer as it passes through. Specifically, the attenuation coefficient of a volumetric medium describes the differential attenuation per distance traveled in the medium; as such it's necessary to know both the coefficient and the thickness of the object in order to compute how much light it attenuates. If light travels a total distance x between entering and exiting the medium, the fraction of it that was attenuated is $1 - e^{-\sigma_a x}$.

The scattering coefficient describes how much light is scattered back out by the object per unit distance of travel through it (Figure 3.4). As light travels through the medium, it will interact with the particles inside of it. After such an interaction, it then may scatter in a different direction than it was originally traveling; some of this light eventually leaves the medium, having taken on the color of the particles it interacted with.

[1] The symbols K and S are often used for σ_a and σ_s, respectively, in the Kubelka Munk literature.

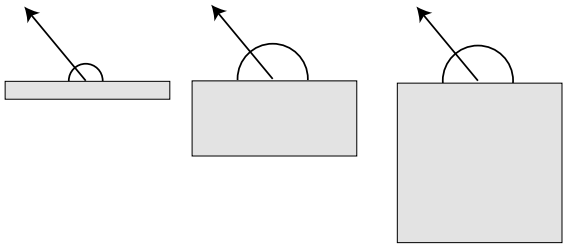


Figure 3.4: Scattering from layers: for a fixed scattering coefficient, increasing the thickness of a layer will tend to increase the amount of light scattered back by it. Given a sufficiently thick layer, however, increased thickness doesn't increase overall scattering (think of painting a wall white; after a certain number of coats of paint, adding another coat doesn't make it any whiter.)

In many applications, it's difficult to determine the values of the σ_a and σ_s parameters for the medium being modeled. However, this doesn't need to stop us from using the Kubelka Munk model. We can estimate reasonable values for the parameters if we know the reflectance of an infinitely thick sample of the layer [Haase and Meyer, 1992], R_∞. Fortunately, R_∞ is an easy and intuitive parameter to start from, since the problems we typically use Kubelka Munk for are along the lines of, "I know that if there's enough tarnish on this object, I won't see the base surface and it'll be a particular shade of green; what color will it be with less tarnish?"

The Kubelka Munk equations (given in the following section) can be inverted to show that the ratio of the attenuation and scattering coefficients can be written in terms of R_∞.

$$\frac{\sigma_a}{\sigma_s} = \frac{(1 - R_\infty)^2}{2R_\infty}$$

We can then select a generally plausible value for σ_a, 1, and compute σ_s. In shader code, we have:

Listing 3.1 This function estimates the coefficients σ_a and σ_s for the Kubelka Munk shading model, given the reflectivity of an infinitely thick sample of the layer under consideration.

```
void KMEstimateCoeffs(color Rinf; output color sigma_a, sigma_s;) {
    if (Rinf == color 0.) {
        sigma_a = color 1.;
        sigma_s = color 0.;
    }
    else {
        color a_over_s = (1. - Rinf) * (1. - Rinf) / (2. * Rinf);
        sigma_a = color 1.;
        sigma_s = sigma_a / a_over_s;
    }
}
```

3.1.3 Reflection and transmission from a single layer

Once that we've got the scattering and attenuation parameters for a medium that we're trying to model, and given the layer thickness, we can start to apply the KM machinery to make pictures. The Kubelka Munk equations give values for R and T, the reflection and transmission coefficients of the layer. More specifically, these give us the fraction of light that is reflected and the fraction that is transmitted through a particular layer.

The Kubelka Munk equations are:

$$R = \frac{\sinh t}{a \sinh t + b \cosh t} \tag{3.1}$$

$$T = \frac{b}{a \sinh t + b \cosh t} \tag{3.2}$$

where $a = (\sigma_s + \sigma_a)/\sigma_s$, $b = \sqrt{a^2 - 1}$, and $t = b\,\sigma_s\,d$.

The translation of this into shading language is straightforward (see Listing 3.2). Although hyperbolic sine and cosine are not available in shading language, they can be computed using the identities $\sinh x = (e^x - e^{-x})/2$ and $\cosh x = (e^x + e^{-x})/2$.

Given the R value, we might then use it in place of the diffuse coefficient `Kd` in a standard light reflection model (e.g. `diffuse()`, or the Oren-Nayar model.) For a thin translucent object, we might use both of them together and gather incident light from both sides of the object:

```
normal Nf = faceforward (normalize(N), I);
color R, T;
KM(sigma_a, sigma_s, thickness, R, T);
Ci = R * diffuse(Nf) + T * diffuse(-Nf);
```

Alternatively, if the layer is on top of an opaque object, we can use the composition equations in Section 3.1.4 below to compute the result of putting one layer on top of another.

Listing 3.2 Implementation of the Kubelka Munk formulas for computing diffuse reflection and transmission from a layer of some thickness. The `sinhcosh()` function (not shown) computes both the hyperbolic sine and cosine of the value passed in.

```
void KM(color sigma_a, sigma_s; float thickness;
        output color R, T;) {
    float i;
    for (i = 0; i < ncomps; i += 1) {
        float s = comp(sigma_s, i), a = comp(sigma_a, i);
        float aa = (s+a)/s;
        float b = sqrt(max(aa*aa - 1., 0.));

        float sh, ch;
        sinhcosh(b*s*thickness, sh, ch);

        setcomp(R, i, sh / (aa * sh + b * ch));
        setcomp(T, i, b  / (aa * sh + b * ch));
    }
}
```

To give a sense of how increasing the thickness of a layer affects its reflection and transmission properties, see the graph in Figure 3.5. For fixed σ_a and σ_s values, we have graphed the values of R and T as we increase the thickness. We have also generated some images to show this effect. We rendered a series of bilinear patches with the light placed at the viewer. As the thickness of the layer increases, it reflects more light; see Figure 3.6.

3.1.4 Composition of layers

The Kubelka Munk model really becomes useful when we start to work with multiple layers. We'd like to compute the resulting reflectance (and possibly transmittance) from putting all of them on top of each other. The resulting values should account for all of the light inter-reflection between all of the layers.

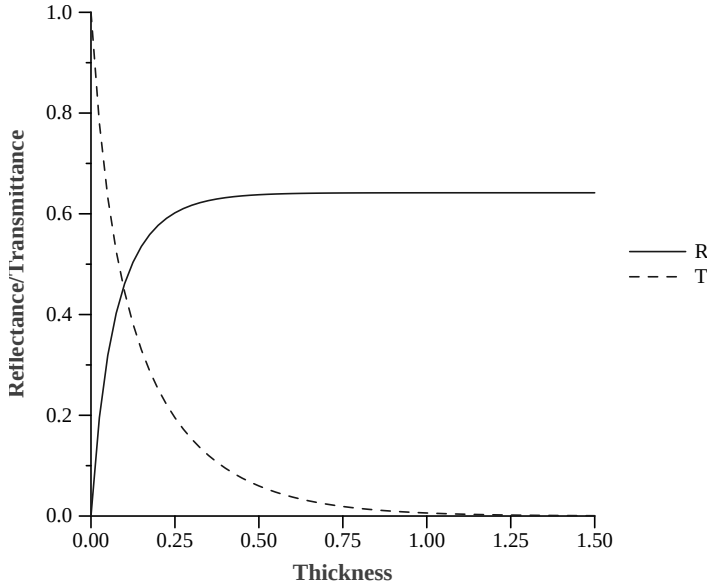


Figure 3.5: Reflection and transmission coefficients computed by the Kubelka Munk model for a layer of increasing thickness with fixed σ_a of 1 and σ_s of 10. Note how as the layer becomes progressively thicker, the addition of extra thickness has less and less influence on the reflection and transmission values.

Figure 3.6: Applying the KM model to compute reflection from a bilinear patch that is illuminated from the eye. As we increase the thickness of the scattering layer from left to right, the amount of light reflected increases.

Consider the case of two layers with known R and T values, R_1, T_1, R_2, and T_2 (see Figure 3.7). Assume for now that the bottom layer is just opaque–e.g. that we're modeling the result of tarnish on a dull old metal surface. Thus, R_2 might be the diffuse reflection coefficient of the surface without tarnish, `Kd`, and T_2 would just be zero.

We can compute a new R value for the amount of light reflected from the new composite surface by considering what happens to the light that passes through the top layer. For example, some will be reflected from the top layer, without even reaching the metal surface (R_1 of it). Some will be transmitted through the top layer (T_1 of it, in fact), reflected by the base surface (R_2), and transmitted back out through the top (T_1). As we consider all of the possible interactions of this form, we find the infinite series:

$$R' = R_1 + T_1 R_2 T_1 + T_1 R_2 R_1 R_2 T_1 + \cdots$$

or

$$R' = R_1 + T_1 R_2 T_1 / (1 - R_1 R_2) \tag{3.3}$$

Similarly, if the base object isn't opaque ($T_2 \neq 0$), we can compute the transmission coefficient for light passing between the two layers together by

$$T' = T_1 T_2 / (1 - R_1 R_2) \tag{3.4}$$

Listing 3.3 Computing the reflectivity of an infinitely-thin layer from the KM scattering and attenuation parameters. (This isn't as useful for shaders, since we're normally starting out with R_∞ and using it to derive σ_a and σ_s.)

```
color KMInfinite(color sigma_a, sigma_s;) {
    float i;
    color R = 0;
    for (i = 0; i < ncomps; i += 1) {
        float a = comp(sigma_a, i) / comp(sigma_s, i);
        float r = 1. + a - sqrt(a*a + 2.*a);
        setcomp(R, i, r);
    }
    return R;
}
```

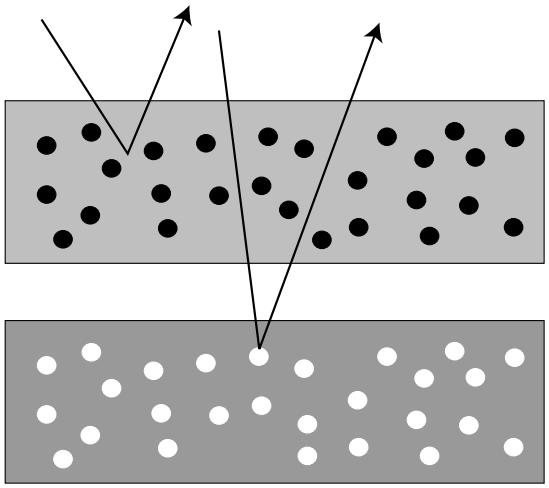


Figure 3.7: KM composition setting: As we consider all of the possible scattering interactions between the layers, the composition equations 3.3 and 3.4 can be derived.

One important thing to notice about these equations is that the R and T values completely characterize the scattering behavior of the layer; once we have computed them, we no longer need to worry about the details of scattering and attenuation coefficients, layer thicknesses, etc. Again, the translation of these equations into shading language is straightforward–see Listing 3.4.

Listing 3.4 Implementation of the Kubelka Munk composition equations in shading language. Given reflectances and transmittances of two layers, compute the new aggregate reflectance and transmittance of them together.

```
color KMComposeR(color R1, T1, R2, T2) {
    return R1 + T1*R2*T1 / (color 1. - R1*R2);
}

color KMComposeT(color R1, T1, R2, T2) {
    return T1*T2 / (color 1. - R1*R2);
}
```

Now we're cooking. Figure 3.8 shows the result of using the Kubelka Munk equations to com-

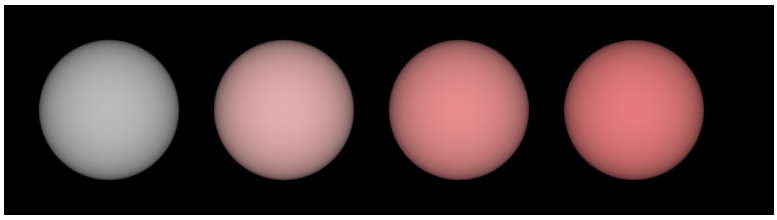


Figure 3.8: Series showing the effect of adding a red layer on top of a grey diffuse object. From left to right, the layer thicknesses are 0, 0.1, 0.5, and 100.

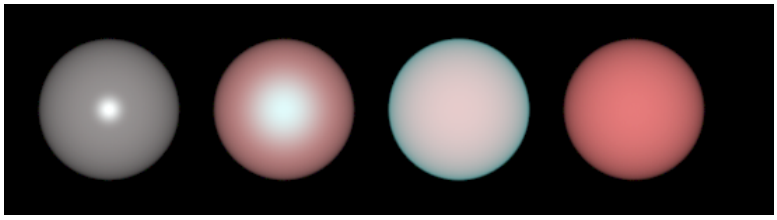


Figure 3.9: Series showing increasing thickness of an added layer on a diffuse/glossy ball.

pute reflection and transmission values for an added red layer on a grey surface and then using the composition equations to compute a final diffuse reflectivity.

3.1.5 Application to non-diffuse objects

In principle, all that we've done so far is only appropriate for matte objects, since the diffuse scattering assumption is a cornerstone of the KM derivation. However, we can play it a little bit fast and loose with that restriction and still see the advantages of the KM framework. Dorsey and Hanrahan did just this for layering diffuse tarnish over glossy metallic surfaces for their weathering model [Dorsey and Hanrahan, 1996].

We take a similar approach, with a few refinements. Assume a basic specular and diffuse shading model for the bottom surface, such that it's described by `Kd`, `Ks`, and `roughness` parameters. We have a layer that we want to put on top of it, with some thickness and computed R and T values. Our goal is to compute reasonable values for new `Kd`, `Ks`, and `roughness` parameters for the composite surface that account for the addition of the layer on top of it. We can treat the diffuse component of it as above, using the KM model and the composition equations (see Listing 3.5).

We then need to estimate the effect of the added layer on the specular part of the original shading model. We update the specular reflection weight `Ks` by treating it similarly to a diffuse R coefficient, however we set $R1$ for the top layer to be zero for this commputation, since we are assuming that the added layer has no specular contribution. This is not physically correct, since we are directly violating the KM assumption of diffuse light distributions, but it's a reasonable approximation to make if we're going to be using this framework.

Finally, we need estimate a new `roughness` value. Because the added layer will tend to diffuse out sharp specular highlights (recall Figure 3.2), we increase the roughness based on a the ratio of our new `Ks` value to its original value. This is completely arbitrary, but seems to work reasonably well in practice. See Figure 3.9 for example images.

Listing 3.5 SL function that computes shading for a composite surface where a diffuse layer has been placed on top of a surface with a standard diffuse-glossy specular shading model. We compute the Kubelka Munk reflection and transmission coefficients given σ_a, σ_s, and the thickness of the added layer, compute a new diffuse reflection coefficient `Kdnew`, and then compute plausible values for the specular coefficients `Ksnew` and `roughnew` that model the effect of the added layer on the light passing through it.

```
color KMOverGlossy(normal Nf; color sigma_a, sigma_s, Kd, Ks;
                   float thickness, roughness;) {
    color R1, T1;
    KM(sigma_a, sigma_s, thickness, R1, T1);

    color Kdnew = KMComposeR(R1, T1, Kd, 0.);
    color Ksnew = KMComposeR(0., T1, Ks, 0.);

    float KsRatio = comp(Ks, 0) / comp(Ksnew, 0);
    float roughnew = roughness * KsRatio;

    extern vector I;
    return Kdnew*diffuse(Nf) + Ksnew*specular(Nf, -normalize(I), roughnew);
}
```

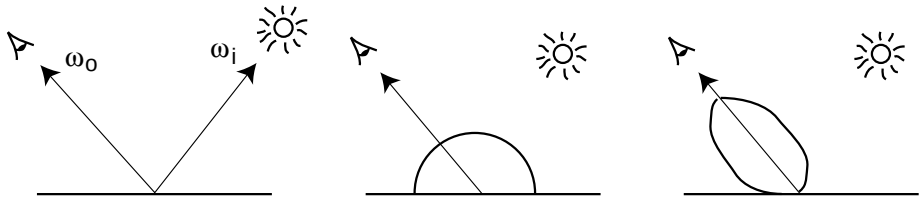


Figure 3.10: Basic setting for the BRDF. Left: given incident (ω_i) and outgoing (ω_o) directions, the BRDF describes how much light is scattered from one to the other. Middle: the BRDF for a diffuse surface equally scatters light in all directions. Right: a glossy surface scatters most light in the mirror direction to the incident.

3.2 The BRDF and Shading Language

Another component of accurate reflection modeling is developing more accurate BRDFs for our surfaces. The BRDF ("bidirectional reflectance distribution function") is just a way of describing how incoming light at a surface is reflected back toward the viewer.[2] (Related to the BRDF is the BTDF ("bidirectional transmission distribution function"). The BTDF describes the distribution of light transmitted through a transparent or translucent surface.) Functions like `diffuse()`, or the Phong reflection model implement *phenomenological* BRDFs–basically ad-hoc functions that were chosen to match observed behavior.

An important characteristic of the BRDF to keep in mind is that it hides the details of how light scatters at a surface. The widely-used Cook–Torrance model for scattering from metallic surfaces has some fairly complex mathematics behind its derivation, but once it was derived, the users of the model didn't need to worry about the physical theory—they can just use the resulting formulas to make realistic pictures. The BRDF models we will discuss can similarly be approached in two ways: with theoretical interest, for understanding methods for modeling light scattering at surfaces; and with practical interest, for gaining new tools and segments of code for writing shaders. If you're not interested in the theory, it can be skimmed rather than read carefully, though it's helpful to understand in order to make the most of the toolbox.

The BRDF is a function of two directions: given an incident direction that light is coming from, ω_i, and an outgoing direction to the viewer, ω_o, the BRDF basically describes how much light the surface reflects from one direction to the other (see Figure 3.10). Given a surface's BRDF f_{r}, and information about the distribution light arriving at the surface, L_i, the reflected light to the viewer L_o is given by integrating the incoming light over the hemisphere Ω above the point being shaded:

$$\mathrm{L}_o(\omega_o) = \int_{\Omega} f_{\mathrm{r}}(\omega_i \rightarrow \omega_o)\, \mathrm{L}_i(\omega_i)\, (\omega_i \cdot \mathrm{N})\, \mathrm{d}\omega_i,$$

The $(\omega_i \cdot \mathrm{N})$ term in this equation is the cosine of the angle between ω_i and the surface normal N. This is *Lambert's law*, which describes the fact that a light source pointing down a surface's normal direction delivers the most light to the surface, and as it rotates around to the surface's horizon, the amount of incoming light decreases according to the cosine of the incoming angle with the normal.

A typical case for evaluating Equation 3.2 is with a fixed set of point light sources and ignoring global illumination (i.e. light that arrives at a surface after scattering from objects that aren't light

[2] McCluney's book on radiometry is a reasonably gentle introduction to the description of surface reflection and light transport [McCluney, 1994]. Alternatively, for graphics-grounded approaches to the theory of light reflection see Hall [Hall, 1989] or Glassner [Glassner, 1995]. For the really-interested, see Nicodemus et al's monograph for the first formalization of the BRDF [Nicodemus et al., 1977].

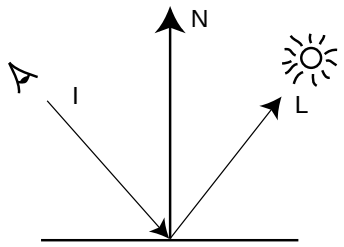


Figure 3.11: Basic setting for `illuminance` in shading language. Note that the incident direction is inward-pointing, in contrast to the BRDF. We will correct for this difference when implementing BRDFs in SL.

sources themselves). In that case, the integral turns into a sum over all of the light sources j:

$$\mathrm{L}_o(\omega_o) = \pi \sum_j f_\mathrm{r}(\omega_j \rightarrow \omega_o)\, \mathrm{L}_i(\omega_j)\, (\omega_j \cdot \mathrm{N}),$$

where ω_j is the direction to the jth light source[3]

3.2.1 Illuminance

Fortunately, shading language has a construct that lets us process all of the light sources in turn– `illuminance`. Illuminance generally takes the position being shaded `P`, a gather axis (typically the surface normal `N`), and a cone angle in radians, which describes the extent of the hemisphere centered around `N` in which to look for light sources. We will always use the usual value of $\pi/2$ for the cone angle, which gives the entire upper hemisphere. The `illuminance` loop is executed once for each light source sample taken (i.e. once for each point or distant light, possibly multiple times for each area light.)

```
...
normal Nf = faceforward (normalize(N), I);
illuminance (P, Nf, PI/2) {
    // process another light source sample
    ...
}
```

Inside an `illuminance` loop, two magic variables are made available: `L`, which gives the (unnormalized) vector from the point being shaded to the position of the light source, and `Cl`, which the light source shader sets to be the amount of light arriving from the light source at `P` (see Figure 3.11).[4] These are just the pieces we need to evaluate a BRDF with the scene's lights.

If we have a SL function `color brdf(vector in, out; normal Nf)` that implements a BRDF model, this `illuminance` loop feeds all of the light source samples through the BRDF and computes the final color.

Note that we normalize `Ln` and normalize and negate `I` before passing it into `brdf()`; this gives us two outgoing vectors for the BRDF function (recall the convention of Figure 3.10) and allows us to assume that they are normalized inside the `brdf()` function, which can save computation there.

In shading language `diffuse()` implements a basic diffuse BRDF; such a surface reflects light equally in all directions. Given a diffuse reflectivity `Kd`, the diffuse BRDF function is:

[3]Hey–where'd that π come from? It's just an artifact from turning the integral over the hemisphere–to a sum. Happens all the time.

[4]For a more extensive refresher on `illuminance`, see also ARM [Apodaca and Gritz, 1999, Section 9.3]).

Listing 3.6 Basic structure for computing light reflected from a surface with `illuminance`, given a function `brdf()` that computes the surface's BRDF for a given pair of directions.

```
vector In = normalize(I);
normal Nf = faceforward (normalize(N), I);
illuminance (P, Nf, PI/2) {
    vector Ln = normalize(L);
    Ci += PI * brdf(-In, Ln, Nf) * Cl * (Nf . Ln);
}
Oi = Os;
Ci *= Oi;
```

```
color brdf_diffuse(vector in, out; normal Nf; color albedo) {
    return albedo / PI;
}
```

When used with the `illuminance` loop above and where `Kd * Cs` is passed in for the `albedo` parameter, this gives the same results as the SL builtin `diffuse()` or the supplied matte shader[5], which is reassuring.

For most of the remainder of this section, we'll be describing ways of computing BRDF values for a pair of directions. We assume that an `illuminance` loop like the one above is being used to compute the final `Ci` value that the shader needs to compute.

3.2.2 Environment and friends

`illuminance` is a passive way to gather incoming light from light sources in the scene; the shader doesn't have much say over what the incoming `L` directions are. In contrast, the SL functions `trace()` and `environment()` let shaders actively ask the renderer about light coming from a particular direction.

For surfaces where the BRDF is strongly directional (e.g. a mirror), these functions must be used. If the BRDF only reacts to light from a small set of incident directions, considering the light sources, which will almost certainly come from directions that are not important, isn't worth the trouble. Conversely, if the surface has a BRDF that is non-zero for most directions, `illuminance` is more appropriate, since it only gives us light from the directions where a lot of light is coming from.

For most of the reflection models we'll consider in the rest of this chapter, `illuminance` is the more appropriate way to gather incident light in the shader. However, keep in mind the `illuminance` and `environment()` are different and complementary ways to gather incoming light in a shader.

[5] Assuming that there are no ambient lights

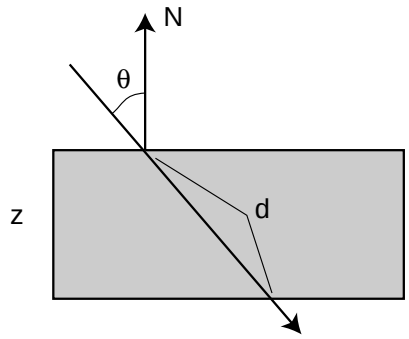


Figure 3.12: Attenuation of light through a layer of thickness z. The total distance d through the layer that the ray passes through is $z/|\cos\theta|$, where θ is the angle of the ray with the surface normal N. If the layer has a uniform attenuation coefficient σ_a, the fraction of light that exits the bottom of the layer is $e^{-\sigma_a z/|\cos\theta|}$.

3.3 Subsurface Scattering

Some of the basic assumptions that are the foundation of the KM model often don't match well with realistic lighting conditions and reflection functions. For example, most of the incident light at a point usually comes from just a few directions (e.g. a set of point light sources), rather than being uniform over the incident hemisphere, and light generally isn't reflected with a uniform distribution. In this section, we'll discuss more sophisticated models of light transport in volumetric media and show how they can be applied to subsurface scattering problems. By modeling light interactions in a layer by following rays through the layer that sample the layer's aggregate scattering behavior, we'll be able to compute solutions to more general scattering problems than the KM model is able to.

In approaching this problem, we will make a distinction between two types of scattering from surfaces. The first is subsurface scattering, where most of the light enters the object's surface layer (which can be modeled as a volume with particles suspended in it), interacts with those particles, and then exits the layer. The second is surface scattering, where most of the light is reflected at the air/surface boundary. Surfaces like paint, skin, and marble are generally modeled best with subsurface scattering, while things like metal and mirrors are better modeled with surface scattering models.

Westin et al were among the first to apply sophisticated light transport algorithms to model surface scattering from complex surfaces [**?**]. They modeled low-level surface geometry (e.g. microfibers in velvet) and ray-traced that geometry to compute tables of BRDF values, which they then represented with spherical harmonics for use at render-time. Hanrahan and Krueger then developed efficient approximations for subsurface scattering from volumetric media, particularly biological objects like skin or leaves [Hanrahan and Krueger, 1993]. We'll focus on the subsurface case in this section.

3.3.1 Attenuation and Scattering

First, a little bit of background on more realistic ways of modeling light in volumes. As with the KM model, both attenuation and scattering need to be considered. Recall that the attenuation coefficient σ_a describes how quickly light is absorbed as it passes through a medium. Specifically, after traveling some distance d, $e^{-\sigma_a d}$ of the light remains. Given a ray entering a volume of thickness z, $e^{-\sigma_a z/|\cos\theta|}$ of the light it is carrying remains when it exits–see Figure 3.12.

When modeling scattering within the layer, we can use *phase functions* to describe the result of light interacting with particles in the layer. A phase function describes the scattered distribution

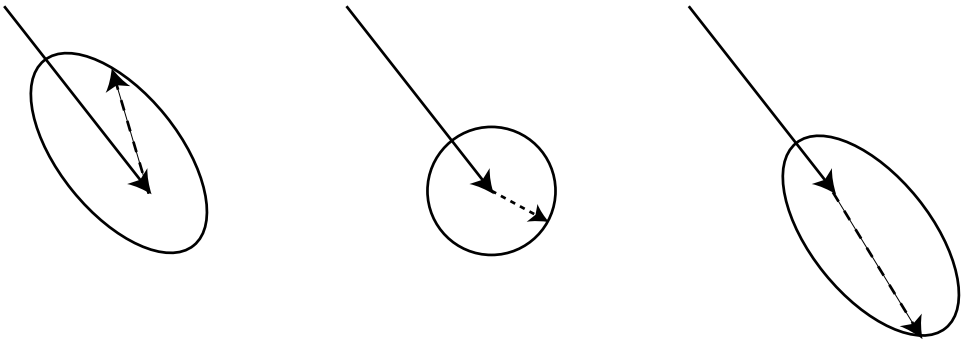


Figure 3.13: Negative values of the asymmetry parameter g for the Henyey-Greenstein phase function correspond to increased backscattering (left), a zero value corresponds to isotropic scattering (middle), and positive values correspond to forward scattering (right).

of light after a ray hits a particle in the layer. Phase functions are in general four-dimensional functions $p(\omega_i, \omega_o)$ of the two directions. However, with the exception of media like ice, that have an oriented structure to them, particles in most media are randomly oriented. Under this situation, the phase function can be reduced to a function of the phase angle between the two directions, $p(\cos\theta) = p(\omega_i \cdot \omega_o)$.

We will use the Henyey-Greenstein phase function, which is a commonly-used parameterized phase function. It takes an asymmetry parameter g, that ranges from -1 to 1, which spans the range of strong retro-reflection to strong forward scattering (see Figure 3.13.)[6]

The Henyey-Greenstein phase function is

$$p(\cos\theta) = \frac{1 - g^2}{(1 + g^2 - 2g\cos\theta)^{3/2}}.$$

Translation into shader code is straightforward:

Listing 3.7 Evaluate the Henyey-Greenstein phase function for two vectors with an asymmetry value `g`. `v1` and `v2` should be normalized and `g` should be in the range (-1, 1). Negative values of `g` correspond to more back-scattering and positive values correspond to more forward scattering. (See Figure 3.13.)

```
float phase(vector v1, v2; float g) {
    float costheta = -v1 . v2;
    return (1. - g*g) / (pow(1. + g*g - 2.*g*costheta, 1.5));
}
```

To compute overall scattering from a layer with known phase function and attenuation and scattering coefficients, it's in general necessary to simulate all of the multiple scattering within the layer. In the general case, where we aren't making the limiting assumptions that Kubelka and Munk did, there is no closed-form solution to this problem. One approach to the problem is to use a numerical integration method, such as Monte Carlo, to compute a solution. Another approach is to make some approximations to make the problem easier to solve.

[6]There's an interesting bit of recursiveness going on here; BRDFs like Cook-Torrance, Oren-Nayar, etc., are generally derived by statistically modeling light scattering from surfaces and thus deriving a simple parameterized model. Here, we're computing a simulation of light scattering on the fly, thus avoiding pre-determined BRDFs entirely. However, one of our building blocks is that the lower-level scattering interactions between light and particles in the medium are themselves modeled with statistically-based phase functions. Thus, the problem has in some sense just been pushed one level down a hierarchy of detail/accuracy.

One frequently used approximation is the *single scattering* assumption. If we only consider a single scattering event with particles in the medium for each incident light ray, we can derive a closed form expression that describes that scattering. This approximation is reasonable if the layer is relatively thin, or if the medium attenuates more light than it scatters (in both cases, it can be shown that single scattering describes the majority of the overall scattering.)

It can be shown [Kokhanovsky, 1999, p. 84] that the BRDF that describes single scattering from a medium is

$$f_{\mathrm{r}}(\omega_i \rightarrow \omega_o) = \frac{a\, p(\omega_i, \omega_o)\, e^{-d\left(\frac{1}{\omega_i \cdot N} + \frac{1}{\omega_o \cdot N}\right)}}{((\omega_i \cdot N) + (\omega_o \cdot N))}$$

where a is the *scattering albedo*, σ_s/σ_a and d is the layer thickness.

Listing 3.8 Compute a the single-scattering approximation to scattering from a one-dimensional volumetric surface. Given incident and outgoing directions ω_i and ω_o, surface normal N, asymmetry value g (see above), scattering albedo (between 0 and 1 for physically-valid volumes), and the thickness of the volume, use the closed-form single-scattering equation to approximate scattering within the layer.

```
float singleScatter(vector wi, wo; normal n; float g, albedo, thickness) {
    float win = abs(wi . n);
    float won = abs(wo . n);

    return albedo * phase(wo, wi, g) / (win + won) *
                (1. - exp(-(1/win + 1/won) * thickness));
}
```

Though we won't use them in examples below, the single scattering transmission approximation is

$$f_{\mathrm{t}}(\omega_i \rightarrow \omega_o) = \frac{a\, p(\omega_i, \omega_o)\, \left(e^{-\frac{d}{\omega_o \cdot N}} - -e^{-\frac{d}{\omega_i \cdot N}}\right)}{(\omega_o \cdot N) - (\omega_i \cdot N)}$$

or, if $\omega_i = \omega_o$,

$$f_{\mathrm{t}}(\omega_i \rightarrow \omega_o) = \frac{a\, p(\omega_i, \omega_o)\, e^{-\frac{d}{\omega_i \cdot N}}}{\omega_i \cdot N}$$

3.3.2 Fresnel Reflection

Another important effect to account for is Fresnel reflection. At the boundary between two objects with different indices of refraction (e.g. air and a scattering volume), some incident light is reflected back from the boundary and some of it is transmitted, depending on the angle of the incident light ray and the indices of refraction of the surfaces. The Fresnel formula describes this effect. (Recall that the index of refraction describes how much more slowly light travels in a medium compared to a vacuum. Air has an index of refraction that is a small amount greater than one, while glass has an index of refraction closer to 1.33.)

At the interface between media that don't conduct electricity (*dielectrics*), the Fresnel reflectance, or fraction of light that is reflected at the boundary and doesn't enter the medium is

$$F_r = \frac{1}{2}(r_{\parallel}^2 + r_{\perp}^2)$$

where

$$r_{\parallel} = \frac{\eta_t(N \cdot \omega_i) + \eta_i(N \cdot T)}{\eta_t(N \cdot \omega_i) - \eta_i(N \cdot T)}$$

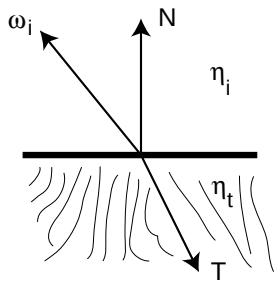


Figure 3.14: Basic setting for computing Fresnel reflection and transmission at the boundary between a medium with index of refraction η_i and a medium with an index of refraction η_t. An incident ray ω_i is entering the layer with i.o.r η_t; due to refraction at the boundary, the new ray direction is T. The Fresnel formulas give the fraction of light reflected and transmitted at the boundary.

and

$$r_\perp = \frac{\eta_i(N \cdot \omega_i) + \eta_t(N \cdot T)}{\eta_i(N \cdot \omega_i) - \eta_t(N \cdot T)}$$

The specular transmission direction T is computed with Snell's law, using ω_i, N, η_t and η_i (See Figure 3.14). The Fresnel transmittance, F_t, is the fraction of light that is transmitted into the medium.

The effect of Fresnel reflection is that as the incident angle heads toward grazing the surface, more light is reflected and less is transmitted. See Figure 3.15 for a visualization of this effect. When light is traveling into a medium with a lower index of refraction than the medium that it's currently in, there is an angle beyond which no light at all is transmitted.

Fortunately, there is an SL function that computes the Fresnel reflectance and transmittance, as well as the directions of the reflected and transmitted rays, R, and T. This function is `fresnel()`. Unfortunately, there is disagreement in the Fresnel transmittance values, F_t, returned by various renderers that support shading language.[7] The correct value of F_t in terms of F_r is

$$F_t = (1 - F_r)\left(\frac{\eta_t}{\eta_i}\right)^2$$

We can use the Fresnel coefficients to more accurately model light entering and exiting surfaces. First, not all of the incident light will actually enter the object; the F_t coefficient describes how much of the incident light we should feed through our subsurface scattering model. Similarly, not all of the light that is exiting the layered medium will make its way out to air; again, the F_t for the exiting ray should scale the amount of exiting light. We also should model the fate incident light that doesn't enter the surface, but is reflected at the boundary, e.g. with a standard surface scattering model (we will do just this in the following section.)

One interesting thing to note is that if we are interested in considering light that enters the medium, interacts with particles inside of it, and then exits, the ratio of η terms in F_t cancels out between the entering and exiting events. The overall reflection due to subsurface scattering is given by the fraction transmitted into the medium times the amount that is scattered inside the medium, times the fraction that exits the medium,

$$F_t^{\text{enter}}\, S\, F_t^{\text{exit}},$$

where S represents the scattering inside the layer (all of these are implicitly functions of ω_i, ω_i, etc.) Because the roles of η_i and η_t are swapped between the entering event and the exiting event, the ratios cancel, so we can just proceed with $F_t = 1 - F_r$ for each of them.

[7] This is a polite way of saying that some of them have it wrong.

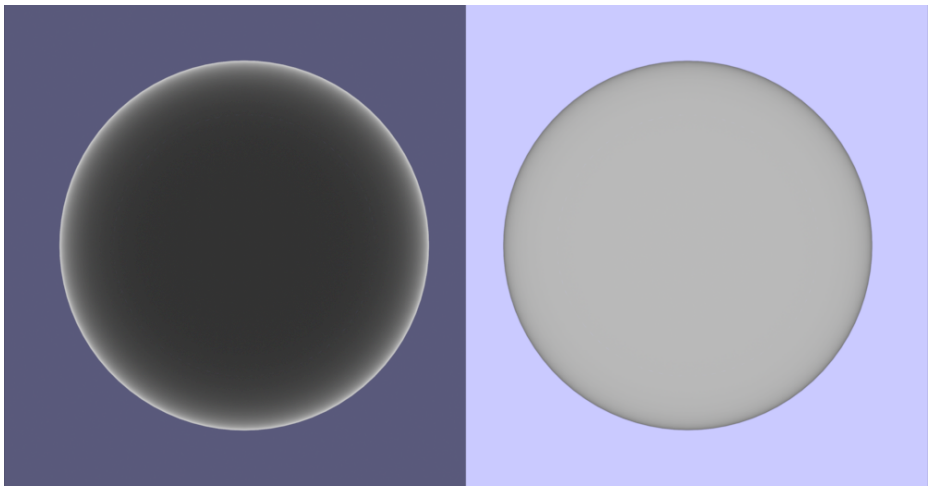


Figure 3.15: Visualizations of Fresnel reflection (left) and transmission (right) coefficient values over spheres. The background colors are adjusted for each so that behavior at the boundaries is easier to see. Note that reflection increases at grazing angles but is quite low at angles closer to the normal. Conversely, most of the light coming in near the normal direction is transmitted, while little of the grazing light is transmitted.

3.3.3 Skin

We can put all of these pieces together to develop a realistic model of reflection from skin. This model is based on the Hanrahan and Krueger subsurface model [Hanrahan and Krueger, 1993], with a few simplifications and a few not-physically-valid tweaks for better visual results.

Because skin has an index of refraction different than air (in fact, the index of refraction of skin layers is around 1.4), we need to consider the Fresnel effect at the boundary. We will write our own little `efresnel` function ("enhanced fresnel") to make this process easier (Listing 3.9. Our function does a few things differently:

- It returns the transmitted direction directly, rather than via an `output` variable; this just makes things a little cleaner.
- It overrides the value of `Kt` returned by the built-in `fresnel` function, so that the renderers that return the wrong value give the right results in the end anyway.
- It increases the value of the Fresnel reflection coefficient by running it through `smoothstep`. This is completely physically unjustified, but the resulting images are a bit nicer by having the Fresnel edge effects exaggerated.

We can now use the Fresnel function to mix in a model for subsurface scattering beneath the surface (See Listing 3.10). The `subsurfaceSkin` function first computes the Fresnel reflection coefficient given the viewing direction as well as a transmitted ray `T` going into the surface (note that `T != I`, since the change in index of refraction bends the incident ray as it enters th medium.)

We then loop over all of the light sources. We mix in some glossy and diffuse reflection, using the standard Blinn model, but weighted according to the Fresnel reflection coefficient; this increases its effect along the edges, and helps model light that enters the backside of the thin edge and exits toward the front.

For each light, we then compute the transmitted direction of the `L` ray as it enters the surface and compute the Fresnel coefficients for that direction. The pair of transmitted rays gives us ω_i and ω_o directions for the single scattering approximation in the layer.

Listing 3.9 Our implementation of the `fresnel` function, with corrections for misbehaving renderers and a bumped-up value of `Kr`.

```
vector efresnel(vector II; normal NN; float eta; output float Kr, Kt;) {
    vector R, T;
    fresnel(II, NN, eta, Kr, Kt, R, T);
    Kr = smoothstep(0., .5, Kr);
    Kt = 1. - Kr;
    return normalize(T);
}
```

We add together three single scattering approximation terms, with g values of $.8$, $.3$ and 0. The $.8$ term comes from measured skin scattering data, as reported in the Hanrahan and Krueger paper. The other two terms are a phenomenological model of the effect of multiple scattering in the layer. Recall that as light scatters in a layer, its distribution becomes more uniform; as such, using phase function g coefficients that are increasingly isotropic is a plausible model of this effect.

Listing 3.10 Implements overall skin subsurface shading model. Takes viewing and surface normal information, the base color of the skin, a color for an oily surface sheen, the ratio of the indices of refraction of the incoming ray (typically 1 for air) to the index of refraction for the transmitted ray (say something like 1.4 for skin), and the overall thickness of the skin layer. Then loops over light sources with illuminance() and computes the reflected skin color.

```
color subsurfaceSkin(vector Vf; normal Nn; color skinColor, sheenColor;
                     float eta, thickness) {
    extern point P;
    float Kr, Kt, Kr2, Kt2;
    color C = 0;

    vector T = efresnel(-Vf, Nn, eta, Kr, Kt);

    illuminance(P, Nn, PI/2) {
        vector Ln = normalize(L);

        // Add glossy/diffuse reflection at edges
        vector H = normalize(Ln + Vf);
        if (H . Nn > 0)
            C += Kr * sheenColor * Cl * (Ln . Nn) * pow(H . Nn, 4.);
        C += Kr * sheenColor * Cl * (Ln . Nn) * .2;

        // Simulate subsurface scattering beneath the skin; three
        // single scattering terms, each progressively becoming
        // more isotropic, is an ad-hoc approximation to the distribution
        // of multiply-scattered light.
        vector T2 = efresnel(-Ln, Nn, eta, Kr2, Kt2);
        C += skinColor * Cl * (Ln . Nn) * Kt * Kt2 *
            (singleScatter(T, T2, Nn, .8, .8, thickness) +
             singleScatter(T, T2, Nn, .3, .5, thickness) +
             singleScatter(T, T2, Nn, 0., .4, thickness));
    }
    return C;
}
```

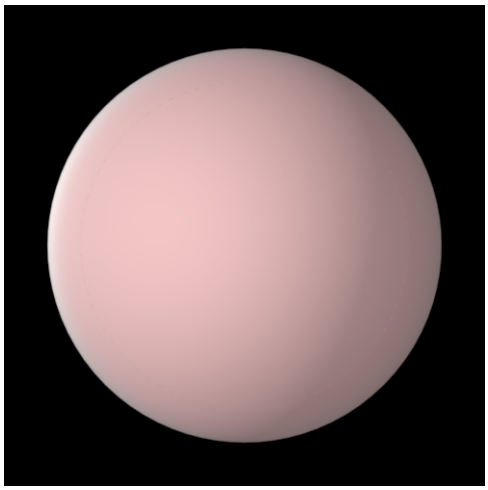


Figure 3.16: Skin shader applied to a sphere.

We can put this together with a simple surface shader that just calls out to the `subsurfaceSkin` function. Here we just use the object's `Cs` color as the skin color; more generally, we might want to use a texture map or procedural function to compute varying skin color. The RGB values of $(1, .6, .6)$ give reasonably accurate Caucasian skin.

Listing 3.11 Basic surface shader that uses the skin reflection model implemented above.

```
surface skin(color Ka = .5; color sheenColor = 1.;
             float eta = 1./1.4, thickness = .5) {
    normal Nn = faceforward(normalize(N), I);
    vector Vf = -normalize(I);

    Oi = Os;
    Ci = Os * subsurfaceSkin(Vf, Nn, Cs, sheenColor, eta, thickness);
}
```

Results of using the skin shader can be seen in Figures 3.16, which shows its application to a sphere, and 3.17, which shows it used on a Cyberware scan of a human head. Note the additional reflectance along the edges of the surfaces, which adds to the overall realism, as well as a good sense of the diffuse character of skin elsewhere.

This basic model can be extended in a number of ways for greater realism (see e.g. Mitch Prater's course notes from last year, which describe the range of parameters that one might want to paint on a surface.) Maps that describe the probability of having freckles at various places on the surface in conjunction with procedural generation of freckles can add important detail. Wrinkles and small creases in skin are have an important impact on its appearance as well; the local variation of surface normal from bump or displacement mapping these effects is another component to consider. Finally, modulating the oiliness at different places on the surface (e.g. extra oil on the forehead and nose) is also important.

Figure 3.17: Skin shader applied to a Cyberware scan of a head. The skin shading model looks better on the skin than on the hair part of the model!

3.4 Summary

We have provided a set of techniques for rendering subsurface scattering from layered media in shading language. These techniques are directly applicable to more accurate rendering of a variety of effects, such as old paint, weathering effects, and biological objects like skin and leaves. I hope that the general paradigms for thinking objects in terms of their scattering layers and their interactions that this chapter tried to provide will be more widely useful for modeling reflection from a broader variety of surfaces. A number of interesting sub-issues have been glossed over; the references previously cited provide a lot more background about the history and derivation of some of these methods and are all good sources for more ideas about surface reflection modeling.

One issue that we haven't discussed at all is the more general case of subsurface reflection, where light enters the surface at one location, travels beneath the surface, and edits at another. See Dorsey et al [**?**] for some striking images that show this effect. For many applications, this level of accuracy isn't necessary; in fact, it can be shown that if the surface is uniformly illuminated over a (relatively) large enough area and if it's homogeneous in the planes that are perpendicular to the normal direction, then the general setting is unnecessary. This approximation often works well in practice. Unfortunately, it's probably not possible to handle the more general case in SL without using the `trace()` function (assuming it's available).

Acknowledgements

Steve Westin kindly read these notes and offered a number of helpful suggestions and ideas for improvement.

Chapter 4

Tutorial On Procedural Primitives

Christophe Hery and Douglas Sutton,
Industrial Light + Magic

4.1 Introduction

The use of procedural RIB generation via *PRMan*'s `"DelayedReadArchive"` (DRA), `"RunProgram"`, and `"DynamicLoad"` functions has become an important part of just about every show produced at ILM. Procedural RIB generation techniques at ILM were initially explored for the specific case of handling large crowds during *Star Wars Episode I*. However their use has expanded to include tasks such as debris instancing, simple blobbies, particle systems and hair creation.

The benefits of procedural RIB generation are fairly obvious. Procedural RIB allows users to extend the RIB language by essentially creating their own geometric primitives tailored to their specific rendering needs. Examples might include creating a tree primitive. One simple call to a DSO can create an entire RIB description for a tree.

Another advantage is that because geometries are being created on the fly, simple pawns can be used to represent what can sometimes be large amounts of heavy data when working with a scene interactively. Also because RIB data is provided on demand based on bounding information, only RIBs that create geometry visible in frame are generated and passed to the renderer. We can also use this information to control the level of complexity of the generated RIB based on the screen coverage.

Procedural RIB also replaces or supersedes the object block instanciation method. Using object blocks limited what could be instanced. For example materials are not allowed in object blocks which makes instancing complex geometries with multiple materials very difficult. Object blocks also have the downfall of actually keeping copies of geometry in memory during rendering even if they are no longer needed. Use of `"DelayedReadArchive"` and reading pre-generated RIBs from disk avoids these drawbacks and allows much more flexibility when dealing with instancing problems.

All in all, the ability to write specialized RIB generation plug-ins or to simply read pre-generated RIBs from disk only as needed makes procedural RIB generation an extremely powerful tool.

4.2 Simple DRA Example

ILM spends a lot of time making things blow up, fall apart or smash into each other (see Figure 4.1). This usually requires the creation of many little pieces of flying geometry. An easy way to get a very

Figure 4.1: Mission to Mars

complex looking field of debris is to use procedural RIB generation tied to a particle simulation. Typically we build a library of RIB files, each of which contains a single piece of debris. Each particle contains not only the position, rotation, and scale of the current piece but also an index into this library that designates which debris RIB is associated with each particle. It's then a simple matter to read the particle file and for each particle in the file place a DRA call with the appropriate transform and RIB file into a master RIB file for rendering.

The parsing of the particle file can also be written as a procedural RIB command. We often use a python script in a `"RunProgram"` call to read our particle files and create the procedural RIB call for each particle. Below is an example of a simple RIB with a procedural RunProgram call to a script which will read a particle file and create the per particle RIB which in this case is a DRA call to "baked" RIBs.

Here is the particle data file, `simple.dat`:

Listing 4.1 `simple.dat`: Simple particle data file

```
#position velocity scale cycleName cycleFrame mtlName boundingBox

-0.328339 1.059637 0.269985 -8.251540 26.693054 6.785024 1 1 1 debrisRockyA 5 Red -1 1 -1 1 -1 1
0.036857 1.047518 -0.287551 0.969217 27.674776 -7.561695 1 1 1 debrisMetalB 2 Blue -2 2 -2 2 -2 2
-0.104504 0.996650 0.268462 -2.873052 27.595234 7.380629 1 1 1 debrisRockyB 3 Green -1 1 -1 1 -1 1
-0.166141 0.988375 0.005661 -4.713767 28.286507 0.160605 1 1 1 debrisMetalA 1 Blue -2 2 -2 2 -2 2
0.140685 0.931055 0.060383 4.225995 28.305006 1.813818 1 1 1 debrisRockyA 2 Red -1 1 -1 1 -1 1
```

Here is the command file, `dra.cmd`:

```
Procedural "DelayedReadArchive" [ "$cycleName.$cycleFrame.rib" ] [ $boundingBox ]
```

Insert the following statement in your RIB file:

```
Procedural "RunProgram" ["dataread.py" "simple.dat dra.cmd"] [ -1e38
1e38 -1e38 1e38 -1e38 1e38 ]
```

Or, to test in the shell any of the programs provided (without a full render), do:

```
echo "0 simple.dat dra.cmd" | dataread.py | more
```

This results in the RIB output stream shown in `output.1` (Listing 4.3).

Using techniques like this gives amazing amounts of flexibility and control over RIB creation through the use of very simple scripts.

Listing 4.2 `dataread.py`: Python particle data parser

```
#! /usr/bin/env python
# dataread.py: Christophe Hery + Doug Sutton - ILM
# usage: Procedural "RunProgram" ["dataread.py" "particles.dat rib.cmd"] [ -1e38 1e38 -1e38 1e38 -1e38 1e38 ]
#

import sys, os, string, regsub

if __name__ == '__main__':

    args = sys.stdin.readline()

    while args:
        # get the information from the renderer
        words     = string.split(args)
        detail    = string.atof(words[0])
        datablock = words[1:]

        # get the data
        dataFile  = datablock[0]
        cmdFile   = datablock[1]

        # check for files existence
        fileok = 1
        if os.path.isfile(dataFile):
            datafile = open(dataFile, "r")
        else:
            sys.stderr.write('FATAL ERROR: Cannot find Data File %s\n' % dataFile)
            fileok = 0
        if os.path.isfile(cmdFile):
            cmdfile = open(cmdFile, "r")
            cmd = cmdfile.read()
            cmdfile.close()
        else:
            sys.stderr.write('FATAL ERROR: Cannot find Cmd File %s\n' % cmdFile)
            fileok = 0
        if fileok == 0:
            sys.exit(1)

        # skip to the data
        datafile.readline()
        datafile.readline()

        # read first line of data
        data = string.split(datafile.readline())

        # loop over data
        while (data):

            sys.stdout.write('AttributeBegin\n')

            # Insert transforms
            sys.stdout.write('   Translate %s %s %s\n' % (data[0],data[1],data[2]))
            sys.stdout.write('   Rotate %s 0 0 1\n' % (data[5]))
            sys.stdout.write('   Rotate %s 0 1 0\n' % (data[4]))
            sys.stdout.write('   Rotate %s 1 0 0\n' % (data[3]))
            sys.stdout.write('   Scale %s %s %s\n' % (data[6],data[7],data[8]))

            # Parse the data into a proper rib command
            ribCmd = regsub.gsub('$cycleName', data[9], cmd)
            ribCmd = regsub.gsub('$cycleFrame', data[10], ribCmd)
            ribCmd = regsub.gsub('$mtlName', data[11], ribCmd)
            bbox = data[12] + ' ' + data[13] + ' ' + data[14] + ' ' + data[15] + ' ' + data[16] + ' ' + data[17]
            ribCmd = regsub.gsub('$boundingBox', bbox, ribCmd)
            sys.stdout.write('   ' + ribCmd)

            sys.stdout.write('AttributeEnd\n')

            # Get next line from the data file
            data = string.split(datafile.readline())

        # clean up
        datafile.close()
        sys.stdout.write('\377')
        sys.stdout.flush()

        # read the next line
        args = sys.stdin.readline()
```

Listing 4.3 `output.1`: RIB stream output of the first example.

```
AttributeBegin
   Translate -0.328339 1.059637 0.269985
   Rotate 6.785024 0 0 1
   Rotate 26.693054 0 1 0
   Rotate -8.251540 1 0 0
   Scale 1 1 1
   Procedural "DelayedReadArchive" [ "debrisRockyA.5.rib" ] [ -1 1 -1 1 -1 1 ]
AttributeEnd
AttributeBegin
   Translate 0.036857 1.047518 -0.287551
   Rotate -7.561695 0 0 1
   Rotate 27.674776 0 1 0
   Rotate 0.969217 1 0 0
   Scale 1 1 1
   Procedural "DelayedReadArchive" [ "debrisMetalB.2.rib" ] [ -2 2 -2 2 -2 2 ]
AttributeEnd
AttributeBegin
   Translate -0.104504 0.996650 0.268462
   Rotate 7.380629 0 0 1
   Rotate 27.595234 0 1 0
   Rotate -2.873052 1 0 0
   Scale 1 1 1
   Procedural "DelayedReadArchive" [ "debrisRockyB.3.rib" ] [ -1 1 -1 1 -1 1 ]
AttributeEnd
AttributeBegin
   Translate -0.166141 0.988375 0.005661
   Rotate 0.160605 0 0 1
   Rotate 28.286507 0 1 0
   Rotate -4.713767 1 0 0
   Scale 1 1 1
   Procedural "DelayedReadArchive" [ "debrisMetalA.1.rib" ] [ -2 2 -2 2 -2 2 ]
AttributeEnd
AttributeBegin
   Translate 0.140685 0.931055 0.060383
   Rotate 1.813818 0 0 1
   Rotate 28.305006 0 1 0
   Rotate 4.225995 1 0 0
   Scale 1 1 1
   Procedural "DelayedReadArchive" [ "debrisRockyA.2.rib" ] [ -1 1 -1 1 -1 1 ]
AttributeEnd
```

4.3 Our original reason for using procedural RIB

Figure 4.2: *Star Wars I* — Gungan Battle

Our interest in procedural RIB first arose when faced with the task of creating two armies of completely CG creatures during *Star Wars Episode I* (see Figure 4.2). The Gungan Battle scenes required us to render large crowds of creatures with complex geometries, complex scene layouts and lots of variety in animation.

It was obvious from the beginning that standard creature by creature pipelines would be too slow. Imagine a simple creature evaluation that takes 30 seconds. Now multiply that by 500. That's more than 4 hours! This is how long it would take to get the performance converted into the proper RIB format for a typical crowd shot. At that stage, we haven't even rendered anything yet. This combined with the memory costs during the creatures parsing make it prohibitively expensive to use usual techniques for generating scenes with lots of creatures.

Our solution was to go with pre-baked RIBs and use procedural RIB generation. By generating and saving a RIB for every frame of each animation cycle we were able to create a very large library of RIBs to chose from. We effectively traded disk space for efficiency: on the Gungan Battle, the total cost of the cycle data (including all the support and original pre-evaluated files) was more than 64 GB (see the Appendix).

Each RIB file was generated as a full description of a single creature including motion blur just as if it were being rendered. We use a flattened RIB hierarchy that allows for easy modification either as a post process or during reading of the RIB via a procedural `"RunProgram"` script. This becomes especially important when trying to optimize disk space usage but still maintain the flexibility to get variety in a crowd. Section 4.3.1 illustrates this further. It is also important to note that using this technique allows us to use our standard geometry and material evaluation pipelines so no compromises are required as far as animation, model, or material detail.

A simple RIB example is shown in Listing 4.4.

Choreographing and placing all the cycles was handled using particles. Each particle contained the information about its transform, its current animation cycle and what frame in that cycle it was on. Just like the debris example we used a procedural `RunProgram` script to read the particle information and select the appropriate RIB to read via DRA.

To optimize the use of procedural RIB it is also necessary to provide an accurate bounding box of the RIB being generated. As part of our RIB pre-generation process we calculated animated bounding boxes corresponding to each frame of a given cycle. When the RunProgram script reads the particle file and selects the appropriate appropriate RIB to read for each member of the crowd, it also looks up the bounding box for that cycle and frame. This is important for both camera frustum culling and for level of detail calculations as discussed in section 4.3.4

4.3.1 Adding material variation

So now we were able to generate crowds, but each member of the crowd had the same materials as every other creature. Our solution was to split the RIBs into material and geometry components. The various material RIB files were made in such a way that they could be combined with any of the animation cycles. Each material RIB contained the same basic structure as any regular RIB file for the creature but with markers where the geometry should go. Listing 4.5 is an example of such a material file.

Listing 4.4 `cube.rib`: Simple RIB example.

```
AttributeBegin
AttributeBegin
Attribute "identifier" "name" [ "backCube" ]
MotionBegin [ 0 0.5 ]
ConcatTransform [ 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 ]
ConcatTransform [ 2 0 0 0 0 2 0 0 0 0 2 0 0 0 0 1 ]
MotionEnd
NuPatch 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 "Pw" [ -0.5 -0.5 0.5 1 -0.5 -0.166667 0.5 1
-0.5 0.166667 0.5 1 -0.5 0.5 0.5 1 -0.5 -0.5 0.166667 1 -0.5 -0.166667 0.166667 1 -0.5 0.166667 0.166667 1
-0.5 0.5 0.166667 1 -0.5 -0.5 -0.166667 1 -0.5 -0.166667 -0.166667 1 -0.5 0.166667 -0.166667 1
-0.5 0.5 -0.166667 1 -0.5 -0.5 -0.5 1 -0.5 -0.166667 -0.5 1 -0.5 0.166667 -0.5 1 -0.5 0.5 -0.5 1 ]
AttributeEnd
AttributeBegin
Attribute "identifier" "name" [ "bottomCube" ]
ConcatTransform [ 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 ]
MotionBegin [ 0 0.5 ]
NuPatch 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 "Pw" [ -0.5 -0.5 0.5 1 -0.5 -0.5 0.166667 1
-0.5 -0.5 -0.166667 1 -0.5 -0.5 -0.5 1 -0.166667 -0.5 0.5 1 -0.166667 -0.5 0.166667 1 -0.166667 -0.5 -0.166667 1
-0.166667 -0.5 -0.5 1 0.166667 -0.5 0.5 1 0.166667 -0.5 0.166667 1 0.166667 -0.5 -0.166667 1
0.166667 -0.5 -0.5 1 0.5 -0.5 0.5 1 0.5 -0.5 0.166667 1 0.5 -0.5 -0.166667 1 0.5 -0.5 -0.5 1 ]
NuPatch 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 "Pw" [ -1.0 -1.0 1.0 1 -1.0 -1.0 0.333334 1
-1.0 -1.0 -0.333334 1 -1.0 -1.0 -1.0 1 -0.333334 -1.0 1.0 1 -0.333334 -1.0 0.333334 1 -0.333334 -1.0 -0.333334 1
-0.333334 -1.0 -1.0 1 0.333334 -1.0 1.0 1 0.333334 -1.0 0.333334 1 0.333334 -1.0 -0.333334 1
0.333334 -1.0 -1.0 1 1.0 -1.0 1.0 1 1.0 -1.0 0.333334 1 1.0 -1.0 -0.333334 1 1.0 -1.0 -1.0 1 ]
MotionEnd
AttributeEnd
AttributeBegin
Attribute "identifier" "name" [ "frontCube" ]
MotionBegin [ 0 0.5 ]
ConcatTransform [ 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 ]
ConcatTransform [ 2 0 0 0 0 2 0 0 0 0 2 0 0 0 0 1 ]
MotionEnd
NuPatch 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 "Pw" [ 0.5 -0.5 0.5 1 0.5 -0.5 0.166667 1
0.5 -0.5 -0.166667 1
0.5 -0.5 -0.5 1 0.5 -0.166667 0.5 1 0.5 -0.166667 0.166667 1 0.5 -0.166667 -0.166667 1
0.5 -0.166667 -0.5 1 0.5 0.166667 0.5 1 0.5 0.166667 0.166667 1 0.5 0.166667 -0.166667 1
0.5 0.166667 -0.5 1 0.5 0.5 0.5 1 0.5 0.5 0.166667 1 0.5 0.5 -0.166667 1 0.5 0.5 -0.5 1 ]
AttributeEnd
AttributeBegin
Attribute "identifier" "name" [ "leftCube" ]
ConcatTransform [ 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 ]
MotionBegin [ 0 0.5 ]
NuPatch 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 "Pw" [ -0.5 -0.5 0.5 1 -0.166667 -0.5 0.5 1
0.166667 -0.5 0.5 1 0.5 -0.5 0.5 1 -0.5 -0.166667 0.5 1 -0.166667 -0.166667 0.5 1 0.166667 -0.166667 0.5 1
0.5 -0.166667 0.5 1 -0.5 0.166667 0.5 1 -0.166667 0.166667 0.5 1 0.166667 0.166667 0.5
1 0.5 0.166667 0.5 1 -0.5 0.5 0.5 1 -0.166667 0.5 0.5 1 0.166667 0.5 0.5 1 0.5 0.5 0.5 1 ]
NuPatch 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 "Pw" [ -1.0 -1.0 1.0 1 -0.333334 -1.0 1.0 1
0.333334 -1.0 1.0 1 1.0 -1.0 1.0 1 -1.0 -0.333334 1.0 1 -0.333334 -0.333334 1.0 1 0.333334 -0.333334 1.0 1
1.0 -0.333334 1.0 1 -1.0 0.333334 1.0 1 -0.333334 0.333334 1.0 1 0.333334 0.333334 1.0 1
1.0 0.333334 1.0 1 -1.0 1.0 1.0 1 -0.333334 1.0 1.0 1 0.333334 1.0 1.0 1 1.0 1.0 1.0 1 ]
MotionEnd
AttributeEnd
AttributeBegin
Attribute "identifier" "name" [ "rightCube" ]
MotionBegin [ 0 0.5 ]
ConcatTransform [ 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 ]
ConcatTransform [ 2 0 0 0 0 2 0 0 0 0 2 0 0 0 0 1 ]
MotionEnd
NuPatch 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 "Pw" [ -0.5 -0.5 -0.5 1 -0.5 -0.166667 -0.5 1
-0.5 0.166667 -0.5 1 -0.5 0.5 -0.5 1 -0.166667 -0.5 -0.5 1 -0.166667 -0.166667 -0.5 1 -0.166667 0.166667 -0.5 1
-0.166667 0.5 -0.5 1 0.166667 -0.5 -0.5 1 0.166667 -0.166667 -0.5 1 0.166667 0.166667 -0.5 1
0.166667 0.5 -0.5 1 0.5 -0.5 -0.5 1 0.5 -0.166667 -0.5 1 0.5 0.166667 -0.5 1 0.5 0.5 -0.5 1 ]
AttributeEnd
AttributeBegin
Attribute "identifier" "name" [ "topCube" ]
ConcatTransform [ 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 ]
MotionBegin [ 0 0.5 ]
NuPatch 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 "Pw" [ -0.5 0.5 0.5 1 -0.166667 0.5 0.5 1
0.166667 0.5 0.5 1 0.5 0.5 0.5 1 -0.5 0.5 0.166667 1 -0.166667 0.5 0.166667 1 0.166667 0.5 0.166667 1
0.5 0.5 0.166667 1 -0.5 0.5 -0.166667 1 -0.166667 0.5 -0.166667 1 0.166667 0.5 -0.166667 1
0.5 0.5 -0.166667 1 -0.5 0.5 -0.5 1 -0.166667 0.5 -0.5 1 0.166667 0.5 -0.5 1 0.5 0.5 -0.5 1 ]
NuPatch 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 4 4 [ 0 0 0 0 1 1 1 1 ] 0 1 "Pw" [ -1.0 1.0 1.0 1 -0.333334 1.0 1.0 1
0.333334 1.0 1.0 1 1.0 1.0 1.0 1 -1.0 1.0 0.333334 1 -0.333334 1.0 0.333334 1 0.333334 1.0 0.333334 1
1.0 1.0 0.333334 1 -1.0 1.0 -0.333334 1 -0.333334 1.0 -0.333334 1 0.333334 1.0 -0.333334 1
1.0 1.0 -0.333334 1 -1.0 1.0 -1.0 1 -0.333334 1.0 -1.0 1 0.333334 1.0 -1.0 1 1.0 1.0 -1.0 1 ]
MotionEnd
AttributeEnd
AttributeEnd
```

Listing 4.5 `cube.mtl` - Sample material definition

```
AttributeBegin
Declare "Ks" "constant float"
Declare "Kd" "constant float"
Declare "Ka" "constant float"
Declare "roughness" "constant float"
Declare "specularcolor" "constant color"
Surface "plastic" "Ks" [.5] "Kd" [.5] "Ka" [1] "roughness" [.1]
Color 1 0 0
AttributeBegin
#$$ ["backCube"] "specularcolor" [0 0 1]
AttributeEnd
AttributeBegin
#$$ ["topCube"] "specularcolor" [0 1 0]
AttributeEnd
AttributeBegin
#$$ ["bottomCube"] "specularcolor" [0 1 1]
AttributeEnd
Color 0 0 1
AttributeBegin
#$$ ["frontCube"] "specularcolor" [1 0 0]
AttributeEnd
AttributeBegin
#$$ ["leftCube"] "specularcolor" [0 1 0]
AttributeEnd
AttributeBegin
#$$ ["rightCube"] "specularcolor" [1 1 0]
AttributeEnd
AttributeEnd
```

Replacing our DRA calls with RunProgram calls to a program which combined a geometry RIB with a material RIB allowed us to generate the heavy geometry RIBs only one time and yet have as many material variations as we needed.

Here is the new command file to use with `dataread.py` (Listing 4.2), `dja.cmd`:

```
Procedural "RunProgram" [ "DJA.py" "$cycleName.$cycleFrame.rib $mtlName.mtl" ] [ $boundingBox ]
```

The resulting RIB stream is shown in Listing 4.6.

There are many things to keep in mind when setting up a system such as this. For instance, to quickly and easily perform RIB edits on the fly, the geometry RIBs have to be baked in a flattened hierarchy using `ConcatTransform` as opposed to `Transform` calls. Material RIB files should be organized and simplified so that the minimum number of `Surface` and `Displacement` calls are made. This reduces memory overhead associated with each shader call (in our case the cost was around 20k). Also because smaller parameter lists on shaders would incur smaller costs, it is more efficient memory-wise to push as much shader data onto the geometry stack through the use of so called primitive variables as possible. An example of this is storing texture names for individual patches on each primitive rather than re-specifying the surface call for each patch with a different texture name. Lastly, and most importantly, make sure that, if you are doing hundreds of objects or creatures, your `RunProgram` executables are very fast and optimized: even if it only takes 10 seconds to combine and return the RIB, doing so for hundreds of creatures on each frame can eat up hours of rendering time very quickly.

Listing 4.6 `output.2`: Resulting RIB stream output from the `DJA` script.

```
AttributeBegin
   Translate -0.328339 1.059637 0.269985
   Rotate 6.785024 0 0 1
   Rotate 26.693054 0 1 0
   Rotate -8.251540 1 0 0
   Scale 1 1 1
   Procedural "RunProgram" [ "DJA.py" "debrisRockyA.5.rib Red.mtl" ] [ -1 1 -1 1 -1 1 ]
AttributeEnd
AttributeBegin
   Translate 0.036857 1.047518 -0.287551
   Rotate -7.561695 0 0 1
   Rotate 27.674776 0 1 0
   Rotate 0.969217 1 0 0
   Scale 1 1 1
   Procedural "RunProgram" [ "DJA.py" "debrisMetalB.2.rib Blue.mtl" ] [ -2 2 -2 2 -2 2 ]
AttributeEnd
AttributeBegin
   Translate -0.104504 0.996650 0.268462
   Rotate 7.380629 0 0 1
   Rotate 27.595234 0 1 0
   Rotate -2.873052 1 0 0
   Scale 1 1 1
   Procedural "RunProgram" [ "DJA.py" "debrisRockyB.3.rib Green.mtl" ] [ -1 1 -1 1 -1 1 ]
AttributeEnd
AttributeBegin
   Translate -0.166141 0.988375 0.005661
   Rotate 0.160605 0 0 1
   Rotate 28.286507 0 1 0
   Rotate -4.713767 1 0 0
   Scale 1 1 1
   Procedural "RunProgram" [ "DJA.py" "debrisMetalA.1.rib Blue.mtl" ] [ -2 2 -2 2 -2 2 ]
AttributeEnd
AttributeBegin
   Translate 0.140685 0.931055 0.060383
   Rotate 1.813818 0 0 1
   Rotate 28.305006 0 1 0
   Rotate 4.225995 1 0 0
   Scale 1 1 1
   Procedural "RunProgram" [ "DJA.py" "debrisRockyA.2.rib Red.mtl" ] [ -1 1 -1 1 -1 1 ]
AttributeEnd
```

Listing 4.7: `DJA.py`: Python DelayedJoinArchive script.

```
#! /usr/bin/env python
# DJA.py (DelayedJoinArchive): Christophe Hery + David Weitzberg + Doug Sutton + Mayur Patelm - ILM
# usage: Procedural "RunProgram" ["DJA.py" "geom.rib mat.mtl"] [ bounding_box ]
#

import sys, string, os

# material tokens
mtlfields = ['declare', 'color', 'opacity', 'texturecoordinates',
             'surface', 'atmosphere', 'displacement',
             'displacementbound','shadingrate', 'shadinginterpolation',
             'matte', 'sides', 'orientation', 'reverseorientation']

# geometry tokens
geomfields = {'polygon' : 1, 'generalpolygon' : 2,
              'pointspolygons' : 3, 'pointsgeneralpolygons' : 4,
              'patch' : 2, 'patchmesh' : 6 , 'nupatch' : 11,
              'subdivisionmesh' : 8}

# geometric primitive variables
primfields = ['"P"', '"Pz"', '"Pw"', '"N"', '"Np"',
              '"Cs"', '"Os"', '"s"', '"t"' '"st"']

# begp, endp: pointers to text array

# organize geometric data
class Geometry:

    def __init__(self, inrib):

        self.f = inrib.read()
        inrib.close()
        # initialize internal variables
```

```
        self.geomBlocks = {}

        self.__makeGeomBlocks__()

    def __stripMats__(self, f):

        begp = 0
        lf = len(f)
        geomBlock = ''

        while begp < lf:

            line = f[begp:begp+22]
            parse = string.lower(string.split(line)[0])

            # first, don't include lines (materials) we don't want
            if parse in mtlfields:
                endl = string.find(f, '\012', begp)
                begp = endl + 1

            elif parse in geomfields.keys():

                doloop = 1
                wordlist = []

                while doloop and begp < lf:

                    endp = string.find(f, '[', begp)
                    endl = string.find(f, '\012', begp)

                    # if the line ends before we're in a bracketed expression
                    # (i.e., the "line" has ended but there are more to come)
                    if (endl < endp and endl >= 0) or (endp < 0):
                        wordlist = wordlist + string.split(f[begp:endl])
                        begp     = endl + 1
                        doloop   = 0
                    else:
                        # pull off the pre-bracket expressions
                        wordlist = wordlist + string.split(f[begp:endp])
                        begp     = endp

                        # find end of bracketed expression
                        endp = string.find(f, ']', begp) + 1

                        # error catch
                        if endp == -1:
                            sys.stderr.write('E01: Bad geomrib format, expecting but could not find"]"')
                            sys.exit(1)

                        # add that word to the list and continue
                        wordlist.append(f[begp:endp])
                        begp = endp

                geomBlock = geomBlock + string.join(purifyGeom(wordlist))

            else:

                endp = string.find(f, '\012', begp)
                if endp >= 0:
                    geomBlock = geomBlock + f[begp:endp] + '\n'
                    begp = endp + 1
                else:
                    geomBlock = geomBlock + '\n' + f[begp:]
                    begp = lf

        return geomBlock

    def __makeGeomBlocks__(self):

        endp = string.find(self.f, 'Attribute "identifier"')

        while endp >= 0:
            begp = string.find(self.f, '\012', endp) + 1
            line = self.f[endp:begp]
            parse = string.split(line)
            geomBlockName = parse[3]
            endp = string.find(self.f, 'AttributeEnd', begp)
            geomBlock = self.f[begp:endp-1]
            geomBlock = self.__stripMats__(geomBlock)
            begp = endp-1
            self.geomBlocks[geomBlockName] = geomBlock
            endp = string.find(self.f, 'Attribute "identifier"', begp)
```

```
# keep only "pure" geometric data
def purifyGeom(wordlist):

    firstword = string.lower(wordlist[0])
    if firstword not in geomfields.keys():
        return wordlist
    index = geomfields[firstword]
    # prefix with a virtual marker: will signal beginning of a real geometric definition
    wordlist[0] = '##$' + wordlist[0]
    while index < len(wordlist):
        if not wordlist[index] in primfields:
           del wordlist[index:index+2]
        else:
           index = index + 2
    # insert another virtual marker here: will signal where to put the material data
    wordlist.append('###')

    return wordlist

# insert corresponding material primitive properties
def addMtlVars(f, mtllist):

    endp = string.find(f, '##$')

    if endp >= 0:
        newgeom = f[0:endp]
        begp = endp + 3
        while endp >= 0:
            endp = string.find(f, '###', begp)
            # has to find it!
            newgeom = newgeom + f[begp:endp] + mtllist + " "
            begp = endp + 3
            endp = string.find(f, '##$', begp)
            if endp >= 0:
                newgeom = newgeom + f[begp:endp] + '\n'
                begp = endp + 3
        newgeom = newgeom + f[begp:] + '\n'
    else:
        newgeom = f + mtllist + '\n'

    return newgeom

# where all the magic happens!
def joinRib(geomfile, mtlfile):

    # make sure files exist before doing any work
    fileok = 1
    if not os.path.isfile(geomfile):
       sys.stderr.write('FATAL ERROR: Cannot find Geometry RIB  %s\n' % geomfile)
       fileok = 0
    if not os.path.isfile(mtlfile):
       sys.stderr.write('FATAL ERROR: Cannot find Material File %s\n' % mtlfile)
       fileok = 0
    if fileok == 0:
       sys.exit(1)

    # create Geometry Object
    G = Geometry(os.popen('catrib %s' % geomfile))

    # create Material Object
    M = os.popen('catrib %s' % mtlfile).read()

    # loop over material object inserting geometry calls
    begp = 0
    endp = string.find(M, '#$$')
    while endp >= 0:
        sys.stdout.write(M[begp:endp])
        endl = string.find(M, '\012', endp) + 1
        line = M[endp:endl]
        parse = string.split(line)
        ident = parse[1]
        del parse[0]
        del parse[0]
        addInGeomStr = string.join(parse)
        sys.stdout.write(addMtlVars(G.geomBlocks[ident],addInGeomStr))
        # no real savings to do this I guess
        # del G.geomBlocks[ident]
        begp = endl
        endp = string.find(M, '#$$', begp)
```

```
    sys.stdout.write(M[begp:])
    sys.stdout.write('\377')
    sys.stdout.flush()

#
# MAIN PROGRAM
#
if __name__ == '__main__':

    args = sys.stdin.readline()

    while args:
        # get the information from the renderer
        words     = string.split(args)
        detail    = string.atof(words[0])
        datablock = words[1:]

        gfile = datablock[0]
        mfile = datablock[1]

        # call our main engine
        joinRib(gfile, mfile)

        # read the next line
        args = sys.stdin.readline()
```

Combining our simple geometry `cube.rib` with our simple material `cube.mtl` generates the output stream shown in Listing 4.8.

Listing 4.8: `output.4`: Result of combining `cube.rib` and `cube.mtl`.

```
##RenderMan RIB
version 3.03
AttributeBegin
Declare "Ks" "constant float"
Declare "Kd" "constant float"
Declare "Ka" "constant float"
Declare "roughness" "constant float"
Declare "specularcolor" "constant color"
Surface "plastic" "Ks" [0.5]"Kd" [0.5]"Ka" [1]"roughness" [0.1]
Color [1 0 0]
AttributeBegin
MotionBegin [0 0.5]
ConcatTransform [1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]
ConcatTransform [2 0 0 0 0 2 0 0 0 0 2 0 0 0 0 1]
MotionEnd
NuPatch 4 4 [0 0 0 0 1 1 1 1] 0 1 4 4 [0 0 0 0 1 1 1 1] 0 1 "Pw" [-0.5 -0.5 0.5 1
-0.5 -0.166667 0.5 1 -0.5 0.166667 0.5 1 -0.5 0.5 0.5 1
-0.5 -0.5 0.166667 1 -0.5 -0.166667 0.166667 1 -0.5 0.166667 0.166667 1
-0.5 0.5 0.166667 1 -0.5 -0.5 -0.166667 1 -0.5 -0.166667 -0.166667 1
-0.5 0.166667 -0.166667 1 -0.5 0.5 -0.166667 1 -0.5 -0.5 -0.5 1
-0.5 -0.166667 -0.5 1 -0.5 0.166667 -0.5 1 -0.5 0.5 -0.5 1] "specularcolor" [0 0 1]
AttributeEnd
AttributeBegin
ConcatTransform [1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]
MotionBegin [0 0.5]
NuPatch 4 4 [0 0 0 0 1 1 1 1] 0 1 4 4 [0 0 0 0 1 1 1 1] 0 1 "Pw" [-0.5 0.5 0.5 1
-0.166667 0.5 0.5 1 0.166667 0.5 0.5 1 0.5 0.5 0.5 1
-0.5 0.5 0.166667 1 -0.166667 0.5 0.166667 1 0.166667 0.5 0.166667 1
0.5 0.5 0.166667 1 -0.5 0.5 -0.166667 1 -0.166667 0.5 -0.166667 1
0.166667 0.5 -0.166667 1 0.5 0.5 -0.166667 1 -0.5 0.5 -0.5 1
-0.166667 0.5 -0.5 1 0.166667 0.5 -0.5 1 0.5 0.5 -0.5 1] "specularcolor" [0 1 0]
NuPatch 4 4 [0 0 0 0 1 1 1 1] 0 1 4 4 [0 0 0 0 1 1 1 1] 0 1 "Pw" [-1 1 1 1
-0.333334 1 1 1 0.333334 1 1 1 1 1 1 1
-1 1 0.333334 1 -0.333334 1 0.333334 1 0.333334 1 0.333334 1
1 1 0.333334 1 -1 1 -0.333334 1 -0.333334 1 -0.333334 1
0.333334 1 -0.333334 1 1 1 -0.333334 1 -1 1 -1 1
-0.333334 1 -1 1 0.333334 1 -1 1 1 1 -1 1] "specularcolor" [0 1 0]
MotionEnd
AttributeEnd
AttributeBegin
ConcatTransform [1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]
MotionBegin [0 0.5]
NuPatch 4 4 [0 0 0 0 1 1 1 1] 0 1 4 4 [0 0 0 0 1 1 1 1] 0 1 "Pw" [-0.5 -0.5 0.5 1
-0.5 -0.5 0.166667 1 -0.5 -0.5 -0.166667 1 -0.5 -0.5 -0.5 1
-0.166667 -0.5 0.5 1 -0.166667 -0.5 0.166667 1 -0.166667 -0.5 -0.166667 1
-0.166667 -0.5 -0.5 1 0.166667 -0.5 0.5 1 0.166667 -0.5 0.166667 1
0.166667 -0.5 -0.166667 1 0.166667 -0.5 -0.5 1 0.5 -0.5 0.5 1
0.5 -0.5 0.166667 1 0.5 -0.5 -0.166667 1 0.5 -0.5 -0.5 1] "specularcolor" [0 1 1]
NuPatch 4 4 [0 0 0 0 1 1 1 1] 0 1 4 4 [0 0 0 0 1 1 1 1] 0 1 "Pw" [-1 -1 1 1
-1 -1 0.333334 1 -1 -1 -0.333334 1 -1 -1 -1 1
-0.333334 -1 1 1 -0.333334 -1 0.333334 1 -0.333334 -1 -0.333334 1
-0.333334 -1 -1 1 0.333334 -1 1 1 0.333334 -1 0.333334 1
0.333334 -1 -0.333334 1 0.333334 -1 -1 1 1 -1 1 1
1 -1 0.333334 1 1 -1 -0.333334 1 1 -1 -1 1] "specularcolor" [0 1 1]
MotionEnd
AttributeEnd
Color [0 0 1]
AttributeBegin
MotionBegin [0 0.5]
ConcatTransform [1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]
ConcatTransform [2 0 0 0 0 2 0 0 0 0 2 0 0 0 0 1]
MotionEnd
NuPatch 4 4 [0 0 0 0 1 1 1 1] 0 1 4 4 [0 0 0 0 1 1 1 1] 0 1 "Pw" [0.5 -0.5 0.5 1
0.5 -0.5 0.166667 1 0.5 -0.5 -0.166667 1 0.5 -0.5 -0.5 1
0.5 -0.166667 0.5 1 0.5 -0.166667 0.166667 1 0.5 -0.166667 -0.166667 1
0.5 -0.166667 -0.5 1 0.5 0.166667 0.5 1 0.5 0.166667 0.166667 1
0.5 0.166667 -0.166667 1 0.5 0.166667 -0.5 1 0.5 0.5 0.5 1
0.5 0.5 0.166667 1 0.5 0.5 -0.166667 1 0.5 0.5 -0.5 1] "specularcolor" [1 0 0]
AttributeEnd
AttributeBegin
ConcatTransform [1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]
MotionBegin [0 0.5]
NuPatch 4 4 [0 0 0 0 1 1 1 1] 0 1 4 4 [0 0 0 0 1 1 1 1] 0 1 "Pw" [-0.5 -0.5 0.5 1
-0.166667 -0.5 0.5 1 0.166667 -0.5 0.5 1 0.5 -0.5 0.5 1
-0.5 -0.166667 0.5 1 -0.166667 -0.166667 0.5 1 0.166667 -0.166667 0.5 1
0.5 -0.166667 0.5 1 -0.5 0.166667 0.5 1 -0.166667 0.166667 0.5 1
0.166667 0.166667 0.5 1 0.5 0.166667 0.5 1 -0.5 0.5 0.5 1
-0.166667 0.5 0.5 1 0.166667 0.5 0.5 1 0.5 0.5 0.5 1] "specularcolor" [0 1 0]
NuPatch 4 4 [0 0 0 0 1 1 1 1] 0 1 4 4 [0 0 0 0 1 1 1 1] 0 1 "Pw" [-1 -1 1 1
```

```
-0.333334 -1 1 1 0.333334 -1 1 1 1 -1 1 1
-1 -0.333334 1 1 -0.333334 -0.333334 1 1 0.333334 -0.333334 1 1
1 -0.333334 1 1 -1 0.333334 1 1 -0.333334 0.333334 1 1
0.333334 0.333334 1 1 1 0.333334 1 1 -1 1 1 1
-0.333334 1 1 1 0.333334 1 1 1 1 1 1 1] "specularcolor" [0 1 0]
MotionEnd
AttributeEnd
AttributeBegin
MotionBegin [0 0.5]
ConcatTransform [1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]
ConcatTransform [2 0 0 0 0 2 0 0 0 0 2 0 0 0 0 1]
MotionEnd
NuPatch 4 4 [0 0 0 0 1 1 1 1] 0 1 4 4 [0 0 0 0 1 1 1 1] 0 1 "Pw" [-0.5 -0.5 -0.5 1
-0.5 -0.166667 -0.5 1 -0.5 0.166667 -0.5 1 -0.5 0.5 -0.5 1
-0.166667 -0.5 -0.5 1 -0.166667 -0.166667 -0.5 1 -0.166667 0.166667 -0.5 1
-0.166667 0.5 -0.5 1 0.166667 -0.5 -0.5 1 0.166667 -0.166667 -0.5 1
0.166667 0.166667 -0.5 1 0.166667 0.5 -0.5 1 0.5 -0.5 -0.5 1
0.5 -0.166667 -0.5 1 0.5 0.166667 -0.5 1 0.5 0.5 -0.5 1] "specularcolor" [1 1 0]
AttributeEnd
AttributeEnd
```

4.3.2 Added bonus of using material replacement

With these basics, we can really start having fun. For instance often we bake our geometry RIBs with geometry that we don't need for each creature. In *Pearl Harbor*, sailor geometry cycles were baked with a variety of hats, lifevests, shirts and pants all together in a single geometry RIB. The material RIBs however are baked with only references to a specific set of geometries, or in our case clothes. This means that at "material joining" time geometry not included in the material RIB can be culled before being handed to the renderer. This makes for a handy way of switching geometry on and off without having to bake geometry RIBs for each possible variation.

4.3.3 Build creatures in pieces

Along similar lines, we can again increase crowd complexity by not only using material variations but also by having variety in geometries. Recently in *Mummy Returns*, we used this technique to create thousands of Anubis warriors, each with its own geometry variation. We accomplished this by breaking up the geometry RIBs into various pieces and then mixing and matching the individual RIBs when we read them in. For instance each warrior had 3 bodies, 7 weapons, 2 earguards, 6 necklaces, 9 bracelets and 3 helmets. This meant that for every frame of every cycle we baked 30 small RIBs, each containing only one part of the total character. We then randomly select a body, a weapon, a helmet... and so on. This gives us 6804 geometry variations to choose from. Combine that with say 3 material variations and you've got over 20,000 variations.

4.3.4 Level of Detail

Another benefit to using procedural RIB is the speed and memory benefits you can achieve when combining it with *PRMan*'s level of detail functions. Especially in crowds it is common to be rendering crowd members at a variety of distances from the camera. Using level of detail we can switch between which RIBs are read based on their screen coverages. Figure 4.3 was generated using RIB similar to what's shown below. In this example we are simply using different colors to represent which RIB file is being read in. For our crowds on SW1, we used 3 different resolutions (plus another level consisting of a colored and scaled-to-proportion sphere), which meant we had to bake 3 resolutions for each cycle for every frame of the cycle. This may seem like a lot of RIBs to bake (and it was!), but the tradeoff we are making here is cheap disk space for expensive processor time and memory.

Here is a simple example that would generate an image like Figure 4.4:

Figure 4.3: *Star Wars I* – Level of Detail RGB representations

Figure 4.4: *Star Wars I* – Gungan LOD Levels

```
AttributeBegin

   Detail [ -1 1 -1 1 -1 1]

   DetailRange [ 200 500 1e38 1e38 ]
   Color 1 0 0
   Procedural "DelayedReadArchive" [ "$cycleName.$cycleFrame.rib" ] [ $boundingBox ]

   DetailRange [ 50 75 200 500 ]
   Color 0 1 0
   Procedural "DelayedReadArchive" [ "$cycleName.$cycleFrame.rib" ] [ $boundingBox ]

   DetailRange [ 10 15 50 75 ]
   Color 0 0 1
   Procedural "DelayedReadArchive" [ "$cycleName.$cycleFrame.rib" ] [ $boundingBox ]

   DetailRange [ 0 0 10 15 ]
   Color 1 1 1
   Sphere 1 -1 1 360

AttributeEnd
```

This results in RIB output stream shown in Listing 4.9.

Table 4.1:

LOD	Geometry	Displace/Bump	Textures	Transparency	Shader
High	All details, i.e., complex	Yes, with .mtl variations	Yes, with all details, with .mtl variations	Yes	Complex
Medium	A little simpler	Yes, but disp used as bump, with .mtl variations	Yes, with less details, with .mtl variations	No	Simplified
Low	One simple geometry (elongated sphere, or polygon mesh) per limb	No	No (1 color per limb)	No	Plastic
Box	`RiSphere` call	No	No (1 color overall)	No	Simpler Plastic

Listing 4.9: `output.3`

```
AttributeBegin
   Translate -0.328339 1.059637 0.269985
   Rotate 6.785024 0 0 1
   Rotate 26.693054 0 1 0
   Rotate -8.251540 1 0 0
   Scale 1 1 1

   AttributeBegin

      Detail [ -1 1 -1 1 -1 1]

      DetailRange [ 200 500 1e38 1e38 ]
      Color 1 0 0
      Procedural "DelayedReadArchive" [ "debrisRockyA.5.rib" ] [ -1 1 -1 1 -1 1 ]

      DetailRange [ 50 75 200 500 ]
      Color 0 1 0
      Procedural "DelayedReadArchive" [ "debrisRockyA.5.rib" ] [ -1 1 -1 1 -1 1 ]

      DetailRange [ 10 15 50 75 ]
      Color 0 0 1
      Procedural "DelayedReadArchive" [ "debrisRockyA.5.rib" ] [ -1 1 -1 1 -1 1 ]

      DetailRange [ 0 0 10 15 ]
      Color 1 1 1
      Sphere 1 -1 1 360

   AttributeEnd
AttributeEnd
AttributeBegin
   Translate 0.036857 1.047518 -0.287551
   Rotate -7.561695 0 0 1
   Rotate 27.674776 0 1 0
   Rotate 0.969217 1 0 0
   Scale 1 1 1

   AttributeBegin

      Detail [ -1 1 -1 1 -1 1]

      DetailRange [ 200 500 1e38 1e38 ]
      Color 1 0 0
      Procedural "DelayedReadArchive" [ "debrisMetalB.2.rib" ] [ -2 2 -2 2 -2 2 ]

      DetailRange [ 50 75 200 500 ]
      Color 0 1 0
      Procedural "DelayedReadArchive" [ "debrisMetalB.2.rib" ] [ -2 2 -2 2 -2 2 ]

      DetailRange [ 10 15 50 75 ]
      Color 0 0 1
      Procedural "DelayedReadArchive" [ "debrisMetalB.2.rib" ] [ -2 2 -2 2 -2 2 ]
```

```
      DetailRange [ 0 0 10 15 ]
      Color 1 1 1
      Sphere 1 -1 1 360

   AttributeEnd
AttributeEnd
AttributeBegin
   Translate -0.104504 0.996650 0.268462
   Rotate 7.380629 0 0 1
   Rotate 27.595234 0 1 0
   Rotate -2.873052 1 0 0
   Scale 1 1 1

   AttributeBegin

      Detail [ -1 1 -1 1 -1 1]

      DetailRange [ 200 500 1e38 1e38 ]
      Color 1 0 0
      Procedural "DelayedReadArchive" [ "debrisRockyB.3.rib" ] [ -1 1 -1 1 -1 1 ]

      DetailRange [ 50 75 200 500 ]
      Color 0 1 0
      Procedural "DelayedReadArchive" [ "debrisRockyB.3.rib" ] [ -1 1 -1 1 -1 1 ]

      DetailRange [ 10 15 50 75 ]
      Color 0 0 1
      Procedural "DelayedReadArchive" [ "debrisRockyB.3.rib" ] [ -1 1 -1 1 -1 1 ]

      DetailRange [ 0 0 10 15 ]
      Color 1 1 1
      Sphere 1 -1 1 360

   AttributeEnd
AttributeEnd
AttributeBegin
   Translate -0.166141 0.988375 0.005661
   Rotate 0.160605 0 0 1
   Rotate 28.286507 0 1 0
   Rotate -4.713767 1 0 0
   Scale 1 1 1

   AttributeBegin

      Detail [ -1 1 -1 1 -1 1]

      DetailRange [ 200 500 1e38 1e38 ]
      Color 1 0 0
      Procedural "DelayedReadArchive" [ "debrisMetalA.1.rib" ] [ -2 2 -2 2 -2 2 ]

      DetailRange [ 50 75 200 500 ]
      Color 0 1 0
      Procedural "DelayedReadArchive" [ "debrisMetalA.1.rib" ] [ -2 2 -2 2 -2 2 ]

      DetailRange [ 10 15 50 75 ]
      Color 0 0 1
      Procedural "DelayedReadArchive" [ "debrisMetalA.1.rib" ] [ -2 2 -2 2 -2 2 ]

      DetailRange [ 0 0 10 15 ]
      Color 1 1 1
      Sphere 1 -1 1 360

   AttributeEnd
AttributeEnd
AttributeBegin
   Translate 0.140685 0.931055 0.060383
   Rotate 1.813818 0 0 1
   Rotate 28.305006 0 1 0
   Rotate 4.225995 1 0 0
   Scale 1 1 1

   AttributeBegin

      Detail [ -1 1 -1 1 -1 1]

      DetailRange [ 200 500 1e38 1e38 ]
      Color 1 0 0
      Procedural "DelayedReadArchive" [ "debrisRockyA.2.rib" ] [ -1 1 -1 1 -1 1 ]

      DetailRange [ 50 75 200 500 ]
      Color 0 1 0
      Procedural "DelayedReadArchive" [ "debrisRockyA.2.rib" ] [ -1 1 -1 1 -1 1 ]

      DetailRange [ 10 15 50 75 ]
      Color 0 0 1
```

```
      Procedural "DelayedReadArchive" [ "debrisRockyA.2.rib" ] [ -1 1 -1 1 -1 1 ]

      DetailRange [ 0 0 10 15 ]
      Color 1 1 1
      Sphere 1 -1 1 360

   AttributeEnd
AttributeEnd
```

A useful trick for adjusting the level of detail ranges is to write a simple `RunProgram` which just echos the "detail" value to `stdout` (Listing 4.10). By placing this primitive at the same location as the object and running a "zoom" render of all levels of detail next to each other, one can find out what the corresponding detail ranges are by looking up the output in the shell.

Listing 4.10 `detail.py`: Detail printing tool.

```
#! /usr/bin/env python
# detail.py: Kevin Sprout - ILM
# usage: Procedural "RunProgram" ["detail.py" ""] [ bounding_box ]
#

import sys, os

sn = sys.stdin.readline()
while (sn):
        sys.stderr.write("%s\n" % sn)
        sys.stdout.write('\377')
        sys.stdout.flush()
        sn = sys.stdin.readline()
```

In general, try to set the overlapping ranges as small as possible. For those screen space areas where two levels need to dissolve into one another, both sets of micropolygons are generated and rendered which can be expensive in both time and memory.

One serious caveat associated with level of detail is how it works in conjunction with shadow buffers. Because the resolution of each crowd member is picked based on its screen coverage the resolution of a given character in the shadow render may not be the same as the resolution in the beauty pass. If the models are built well and luck is with you, you will hopefully be able to use lower resolution geometries in your shadow buffers than you use in your beauty passes. However it all depends on the situation.

Don't forget that in addition to using lower resolution geometries, the material RIB file also should simplify as creatures recede from camera.

4.3.5 What's Next?

Now that we know how to combine a geometry RIB with a material file at run time, we would like to leave as an exercise to the reader to make another `RunProgram` which inputs 2 RIBs, a mtl and a blending factor. Internally this procedure would linearly interpolate C.V. to C.V. the content of the two RIBs according to the blend. Of course, this would assume that the RIBs are compatible, ie that they have been baked from the same exact master topology. With such a tool in hand, one can create on the fly new transition cycles, which can help smooth the action. The choreography program, again as the brain controlling the whole performance, would of course have to additionally compute and output the progression of the blend.

So far most of our uses of procedural RIB have been using a pre-baked system and then reading those files from disk as needed or generating simple geometry like leaves on the fly. The next step is to do creature evaluation in the procedural commands themselves. As we've mentioned previously this would have to be very fast as even a few seconds per creature on a large number of creatures can

add up to long rendering times very quickly. But it might be a good technique when using procedural animation. Or for smaller scenes with fewer creatures it could create a simple way vary animation based on other factors such as wind or other dynamics in the scene.

4.4 Other procedural RIB uses

Figure 4.5: “Work in Progress” – procedural plants.

Creating a horde of synthetic actors is only one small way that procedural RIB generation can be used. Writing external programs or DSOs to actually create geometry on the fly is a very powerful and exciting use of these tools as well.

DSO’s are linked directly against the RenderMan libraries and can make use of all the standard primitive geometry types as well as material and transformation statements to create a whole new user defined primitive. This is a particularly effective way to create simple repeated geometries or complex organic structures such as trees and plants, possibly even using L-systems.

The simplest plugins may be used to create calls to `RiPoints` or `RiCurves` for use in creating things like dust, grass or hair. Specifying more complex structures such as leaves or even whole trees can be a great way to create a forest by specifying a few simple parameters for each plant.

4.4.1 Procedural animation of primitives

Figure 4.6: A simple procedural geometry and animation example.

DSO's also have the advantage that these geometries can be procedurally animated. By adding some sort of procedural motion to the C.V.'s as the geometries are created, you can easily simulate wind and other natural motions for leaves and grass that would be difficult to control any other way. These subtle motions can really bring a scene to life.

When generating geometry procedurally using a DSO it is also very easy to adjust not only the geometry as it is created but at the same time and using the same input information control and vary shading information. This allows the definition of procedural attributes used by shaders at render-time. For instance, different texture parameters can be specified for each leaf based on random numbers chosen as each leaf is generated.

Figure 4.5 demonstrates a simple set of geometry created by a renderman DSO. Wind motion and color variations are added and controlled directly by the plugin when the geometry is created. Listing 4.11 constains the source code for this DSO.

Listing 4.11: Example RiProcedural.

```
/* Ken Wesley - ILM 2001 */

#include <stdio.h>
#include <math.h>
#include <ri.h>
#include <string.h>
#include <stdlib.h>

#define X      0
#define Y      1
#define Z      2

#define R      0
#define G      1
#define B      2

#define BasisStep       1
#define NWide           4
#define NTall           7

int        nargs;
int        seed;
int        frame;
int        nNodes;

#define MaxNodes        2500

#define MinOffset         0.5
#define MaxOffset         20.0
#define CycleDistance        (MaxOffset/2.0)
#define CycleFrames        48

#define MaxScale 1.1
#define MinScale 0.9

RtToken        RI_RANDOMVAL;

float prob()
{
    return( (float)(random()/ 2147483647.0) );
}


void ConvertParameters(char *stringdata)
{
    /* Convert the data from the input string */
    printf("================================\n");
    printf("ken.so input string:\n");
    printf("%s\n", stringdata);
    printf("================================\n");
    sscanf( stringdata,"%d %d %d", &seed, &frame, &nNodes );
}

#define PatchWidth .5
#define PatchHeight 1.33
#define SphereRadius ( PatchWidth / 2.0 )

void instanceNode( float offset, float orientationAngle, int frame )
{
    int i, j;
    RtInt   nu, nv;
    RtFloat randomVal[NWide*NTall];
    RtPoint vertices[NWide*NTall];
    RtColor        color;
    float        thisYValue;
    float        thisZValue;
    float        sphereZValue;
    float        scaleVal;
    float        distanceAngle;
    float        frameAngle;
    float        heightAngle;

    /* set the control vertices for this node's B-spline patch*/
    distanceAngle = offset/CycleDistance * 2.0 * M_PI;
    frameAngle = (float)frame/(float)CycleFrames * 2.0 * M_PI;
    for ( i = 0; i < NTall; i++ ) {
        heightAngle = (M_PI/2.0)/(float)(NTall-2) * (float)i;
        thisYValue = ((float)i/(float)( NTall - 1 ))*PatchHeight
            - ( 1.0 / ( float) ( NTall - 1 ) );
```

```
        thisZValue = cos(distanceAngle + frameAngle + heightAngle)
            *(PatchWidth*2.0);
        if ( i == (NTall - 2))
            sphereZValue = thisZValue;
        for ( j = 0; j < NWide; j++ ) {
            vertices[i*NWide+j][X] = (((float)j/(float)(NWide-1))-.5)
                                        *(PatchWidth*2.0);
            vertices[i*NWide+j][Y] = thisYValue;
            vertices[i*NWide+j][Z] = thisZValue;
            if (( i == 0 ) && ( j == 0 ))
                randomVal[i*NWide+j] = prob();
            else
                randomVal[i*NWide+j] = randomVal[0*NWide+0];
        }
    }

    nu = NWide;
    nv = NTall;

    /************************/
    /* Begin RenderMan Calls */
    /************************/
    RiTransformBegin();
    RiRotate( orientationAngle, 0.0, 1.0, 0.0);
    RiTranslate( 0., 0., offset );
    scaleVal = prob()*(MaxScale - MinScale) + MinScale;
    RiScale( scaleVal, scaleVal, scaleVal );
    color[R] = 0.4;
    color[G] = 0.2;
    color[B] = 0.1;
    RiColor( color );
    RiPatchMesh ( "bicubic", nu, RI_NONPERIODIC, nv, RI_NONPERIODIC,
                    RI_P, vertices, RI_RANDOMVAL, randomVal, RI_NULL);
    RiTranslate( 0.0, PatchHeight*(NTall-2)/(NTall-1), sphereZValue );
    color[R] = 0.8;
    color[G] = 0.2;
    color[B] = 0.8;
    RiColor( color );
    RiSphere( SphereRadius, -SphereRadius, SphereRadius, 360.0, RI_NULL);
    RiTransformEnd();
    /**********************/
    /* End RenderMan Calls */
    /**********************/
}


void printUsage()
{
    printf("\n\n");
    printf("usage:  ken.so [seed] [frame] [nNodes]\n");
}


RtVoid Subdivide(RtPointer blinddata, RtFloat detail)
{
    int thisNode;
    float offset, orientationAngle;

    RI_RANDOMVAL = RiDeclare( "randomVal", "varying float");
    srandom( seed );
    RiBasis( RiBSplineBasis, BasisStep, RiBSplineBasis, BasisStep );
    if ( nNodes > MaxNodes ) {
        printf("This plugin can only handle %d nodes.\n", MaxNodes );
        printf("If you'd like more nodes, please re-compile with a greater\n");
        printf("        value for \"MaxNodes\".\n");
        exit(0);
    }
    for ( thisNode = 0; thisNode < nNodes; thisNode++ ) {
        offset = prob()*(MaxOffset - MinOffset) + MinOffset;
        orientationAngle = prob()*360.0;
        instanceNode( offset, orientationAngle, frame );
    }
}


RtVoid
Free(RtPointer data)
{
```

```
    /* This is where we would free data if ConvertParameters allocated */
}
```

4.4.2 Fur

In last year's "Advanced Renderman" SIGGRAPH course, Rob Bredow from Sony Imageworks demonstrated how a DSO was used to produce the fur on *Stuart Little*. Refer to the SIGGRAPH 2000 course notes for more information.[1]

4.4.3 Blobbies, particles and other fun stuff

There are far too many uses for procedural DSOs to go into them all, but a few other uses might be: a mechanism for building animated blobby structures using the `RiBlobby` primitive, creating simple particle system renders using the `RiPoints` primitive and creating ropes and string using the `RiCurves` primitive. Basically the only limit is the amount of coding involved and your imagination.

4.5 Let's look at pictures

Figure 4.7: *The Mummy* — Scarabs

The scarab shot from *The Mummy* (see Figure 4.7) is a nice example of a very simple crowd rendered using only DRA calls. A six frame cycle was prebaked out as RIBs and instanced on particles. Because of the simplicity of the creatures, there was no need for multiple resolutions or even material variations.

Figure 4.8 is another example of a simple setup from *Magnolia*. To make the rendering even more efficient, the static frogs (the dead ones on the ground) were computed once, then the moving geometries were depth composed among them as the animation evolved.

Figure 4.9 is a shot from SW1 that shows the extent of the Gungan army. Because the choreography was finalized in layers, the images were computed in 3 beauty passes. For each group, there were 4 lights (each of them computing shadows). There are more than 7000 creatures, all of them procedurally generated from only 14 baked cycles. 1340 textures were read, representing more than 1Gb of space. Even with levels of detail, the average rendering time per frame was 7 hours, and the memory spiked at 770 Mb. So it is still expensive!

[1] If you didn't buy a copy of the notes then, you could print them from the SIGGRAPH 2000 Course Notes CD-ROM, or download them from The RenderMan Repository, `www.renderman.org`.

Figure 4.8: Magnolia

Figure 4.9: *Star Wars I* – Gungan Army

4.5.1 Conclusion

It's been a little over 3 years since ILM began using procedural RIB generation. In that time it has become a crucial tool for producing special effects and has been used on nearly every show to pass through ILM since *Episode I*. As powerful as it is, we are only at the tip of the iceberg. The more we use it, the more uses we find for it. So if you haven't already started playing with these features, you should. It's amazing the fun you can have.

Acknowledgements

We'd like to thank the following people:

Ken Wesley for the text and examples of Section 4.4.1; Pixar for opening up the renderer; David Weitzberg and Mayur Patelm for their original `DJA.py` efforts; Hirami Ono, for her crowd particule representation on *Episode 1*; Dan Goldman and Ari Rapkin for their RenderMan expertise; Tommy Burnette for general python wrangling; Cliff Plumer and Philip Peterson for supporting this presentation; the ILM PR department for providing the pictures; and everyone at ILM who has helped with suggestions and bug fixes to turn this set of tools into a viable pipeline.

Appendix: Statistics

		Shortest Cycle		Longest Cycle		All Cycles	
		RIB Size	Frames	RIB Size	Frames	RIB Size	Frames
Gungan Cavalry	High	12 MB	39	1.3 GB	450	14.7 GB	5352
	Medium	9 MB	39	108 MB	450	1.1 GB	5352
	Low	2 MB	39	19 MB	450	209 MB	5352

		Shortest Cycle		Longest Cycle		All Cycles	
		RIB Size	Frames	RIB Size	Frames	RIB Size	Frames
Gungan Infantry	High	44 MB	29	635 MB	397	1.9 GB	11817
	Medium	3 MB	29	37 MB	397	1.2 GB	11817
	Low	330 KB	29	4 MB	397	251 MB	11817

		Shortest Cycle		Longest Cycle		All Cycles	
		RIB Size	Frames	RIB Size	Frames	RIB Size	Frames
Battle Droids	High	17 MB	29	118 MB	302	19.4 GB	11816
	Medium	2 MB	29	21 MB	302	1.2 GB	11816
	Low	2 MB	29	14 MB	302	250 MB	11816

	Number of cycles
Gungan Cavalry	27
Gungan Infantry	59
Battle Droids	49
Others	38
Total number of cycles/animations checked in and baked	173

Total number of frames for all cycles	118755
Average number of frames per cycle	173.6
Total time to bake all cycles	4576 hours
Average time to bake a cycle	32 hours
Total rib disk space for all cycles *	55 GB
Average rib disk space for a cycle *	325 MB

* RIB files in binary `gzip`'ed format.

Chapter 5

Volumetric Shaders Used In The Production Of *Hollow Man*

Laurence Treweek and Brian Steiner,
Sony Pictures Imageworks
laurence@imageworks.com
brian@imageworks.com

Abstract

Volume rendering solid textures and deriving shadeable surfaces from 3D noise textures.

A description of techniques and tools used in the creation of the Transformation Sequences for the film Hollow Man. This talk will cover some of shaders and plugins used to animated and render a disintegrating human body.

5.1 Goals

Our goals were:

- To create at render time the look of a mass of tissue inside animating NURBS geometry.
- The shape of the animating disintegrating volumes must be completely controllable over time.
- The inner surfaces and textures must look realistic and be capable of shading with the same quality and control as the geometric surface.
- Must work with camera and object motion blur from fast camera movement and frantic animation of the body.
- Volumetric shapes created by the shader must be self shadowing and able to cast shadows.
- Rendering times must be reasonable enough to be used on a large area of the screen for eighty shots in production.

5.2 Methods

At the beginning of the project we analyzed the storyboards that needed to brought to screen and started to determine what kind of technology could be used to produce the desired look and animation.

5.2.1 Method 1 – Particles

To fill the enclosed NURBS geometry with particles that pack inside the volume. The particles could then be switched on or off to create an animating volume.

Pro: This method would be fairly easy to setup and animate using traditional particle creation and animation routines.

Con: Using particles would create a huge amount of data to keep around considering the number of body parts that would have to use this effect. It would also be very difficult to shade these particle to look like the variety of tissue type we needed to emulate.

5.2.2 Method 2 – The Visible Human Project

To use the existing imagery contained in the Visible Human Project to fill the volume.

Pro: The Visible Human Project contains images of slices through a human body taken at 2mm intervals. When these are stacked on top of each other you have a voxel-set that describes the external and internal look of the body with medical accuracy.

Con: The problem with using this data lies in the difference between the proportions of the geometry and the voxel data. The geometry used had to match the scale and proportions of Kevin Bacon and the voxel data would line up as it was extracted from a much heavier body. To get these to volumes to occupy the same space would entail developing a palette of complex tools to morph the two together. The isolation of individual objects within the slice data would also have been a complex task as they were visually indiscernible within each image. A possible solution to this would have been to use the MRI dataset to try and isolate different parts based on relative density, but this method would also be inaccurate and need cleaning up afterwards. The static size of this data would be enormous to distribute to servers and artists needed to render and work with it.

5.2.3 Method 3 – Procedural generation of volumetric data at render time

To generate, using 3D noise, the surfaces and textures inside the volume.

Pro: Storage requirements are virtually zero. The same 3D noise used in the shading of the geometric surface can be extended to the interior volume to form a continuous effect.

Volumetric data is not static and locked. Shader parameters can be used to change the density and look of the volumetric data. As the data is created by the shader, it can also be sampled, and averaged at arbitrary rates.

Con: Procedural data has to look convincing. Image based data cannot be used within the volume to add a realistic look as in the way a texture map would be used on a surface.

5.3 Why use RenderMan?

There are varieties of translating a piece of volumetric data to a static image. These can range from transforming a block of voxel data into screen space to "ray marching." RenderMan does not seem to be the obvious choice of renderer to solve this problem — so why did we use Renderman?
Reasons to use Renderman:

- Shaders are comparatively easy to create. Many arithmetical and vector functions already built-in. More time can be devoted to created higher level functionally instead of creating basic libraries.
- Final "look" is a known quantity. This factor was one of the major influences on deciding which package to use as a development base. The majority of shader writers and artists at Imageworks are more familiar with the workings of RenderMan compared to other renderers and there is also an expectation of what the final image quality will be. There is also familiarity with important functions that determine that "look" such as texture mapping and noise functions.
- It was advantageous to use one renderer for parts of the body. Because the outside surface of the object was visible at the same time as the interior volume on any one frame, it meant that a shader that could render both surface and volume would make a better blend.
- The alternative would have been to render the surface with RenderMan and depth composite it with the volume from another renderer.

With the addition of DSOs to RenderMan the possibility of extending functionality greatly increases beyond the traditional shader additions to the package.

Writing new renderers or adding new rendering software to a facility is always a large undertaking and goes beyond the initial problem of producing an image that work. There are other considerations such as fitting into to an existing production pipeline and training artists to use new tools. Even though we did create a specific renderer for Hollow Man, we kept its usage to a minimum and only used it where absolutely necessary.

5.3.1 Added RenderMan DSOs

Even though RenderMan gave us a good base to start working with we knew that extra functionality was going to be needed to make the shaders work.

5.3.2 Ray Tracing

In order to ray march through a volume we needed the ability to ray trace the geometry.

We experimented with using BMRT in rayserver mode together with the rayserver DSO to test our theories. At the beginning of the project we realized that memory consumption and processor usage would be high so we decided to develop our own rayserver DSO instead of running two renderers in parallel with duplicated scenes.

Our ray tracing DSO worked in two modes, read and write. The "write" mode was used in a pre-rendering phase to create a file that could be randomly accessed for ray intersection calculations. During the pre-render phase all the micropolygons that RenderMan created were written to a file along with all of their attributes. Inside the file the micropolygons were stored in world space and could be converted to a Wavefront `.obj` file for debugging purposes.

In "read" mode the file could be randomly accessed so that a ray could be fired into a cell of polygons rather than the whole scene, giving much faster calculations times. Although this was not an ideal solution it did work well enough in production by reducing the memory footprint needed to

ray trace. The DSO was also able to pass back to the shader extra attributes such as the name of the texture map applied to the intersected geometry or its uv values.

5.3.3 Animation Data

The shaders needed to aware of animation curves that would drive the disintegration effect. Different parts of the body needed to know their relative position within the overall transformation animation. To give the point being shaded a reference point we created a DSO that could read curve data. While marching through objects we could search for the closest point on one of these curves. When the closest point was found, the arc-length attribute was calculated and passed back to the shader. Giving us a simple means of animating the visibility of P.

```
point Pcurr = transform("current", "object", P);
float anim_length = 195; /* Make visible every point that is within 195 units of  heart. */
float current_length = BIG_NUMBER;
while(current_length > anim_length) {
    current_length = find_closest_point(Pcurrent); /* call reference curve DSO */
    Pcurr += step_size * trace_direction;
}
calculate shading at position Pcurr;
```

In the code example above if we animate the value of `anim_length` then we get a wipe effect across every object with this shader and curve data. Figure 5.1 shows the placement of the curves inside an object (a femur in this case) which were used by the DSO.

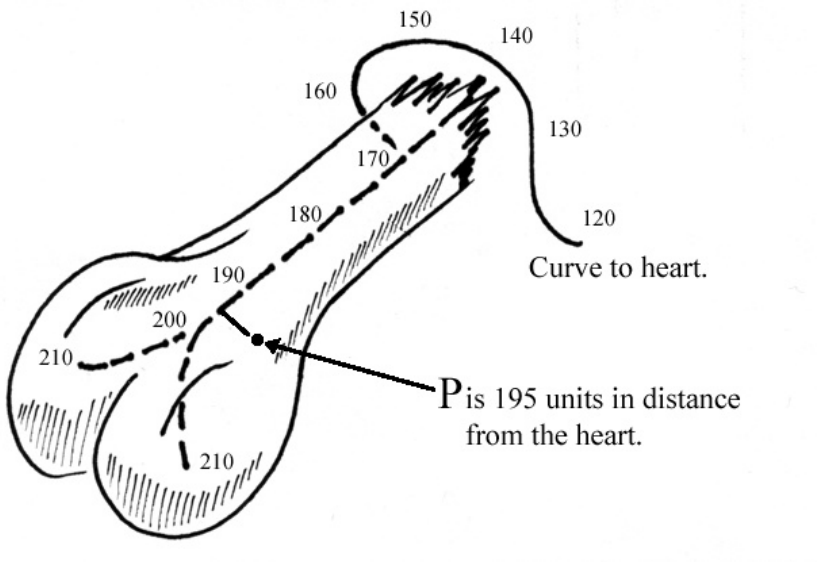


Figure 5.1:

This technique was used extensively in animating the transformation shots. The curves could be applied to many different type of objects and the same DSO could be used in many different shaders.

The curves themselves were actually generated by a Maya plugin developed by Scott Schinderman that let artists chain curves together to form a road map of the animation flow through the body. Adding or deleting curves from the tree could change the overall effect of the animation.

Figure 5.2: *Hollow Man*

5.4 Transforming Bones

The disintegrating bone effect was designed to look as if a cascading chemical reaction traveled from a source point in the bone, spreading outwards making the space occupied by the bone invisible in its path. The effect also needed to have a semi transparent amber-like leading edge that would illustrate the position of the chemical reaction between the bone and the reactive formula developed in the film's storyline. The effect needed to show five different types of materials during the course of the animation. In Figure 5.2, the bone effect is used exclusively but in all other shots the shader had to work with vary kinds of geometry, shaders and renderers surrounding it. The shader also had to hold up under a variety of viewing angles and animation conditions. The lighting design of the original plate photography also called for many soft shadow-casting lights to be used to mimic the fluorescent lighting of the laboratory.

5.4.1 The Layers

Layer 1) Wet Look.

This material needed to look as if a layer of fascia and collagen were coating the fat and outer bone materials. This layer was producing by taking point `P` at the surface and offsetting it by a fairly smooth noise value as in a regular bump calculation and then using the normal produced to calculate an extra specular contribution that was added in to the final lighting calculations.

```
point Pwet = P + wet_noise * N;
normal Nwet = normalize(calculatenormal(Pwet));
color CiWet = wet_colour * my_specular_function(Cl, Nwet);
Ci = surface_colours + (CiWet * WetAmount);
```

This kind of offset specular lookup produces a parallax shift in animation between the specular shading and the diffuse shading, giving the impression that there are two surfaces instead of one.

2) Fat.

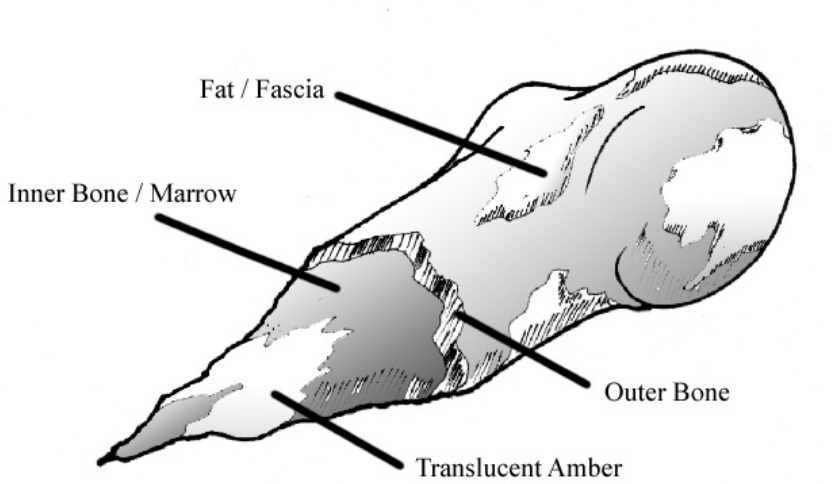


Figure 5.3: Bone layers.

The exterior of the bone needed a layer of fat added to it that also had to disintegrate along with the bone. This layer was produced by displacing the outer surface to form the impression of a clumps of fat stuck to the outer bone surface. The amplitude of the fat displacement was also used to blend in the amount of fat material at the surface. When the bone animated the amplitude decreased making the fat shrink closer to the surface and also mix in more of the outer bone surface until it disappeared completely.

3) Amber chemical reaction.

The amber like material needed to occupy a thin layer within the volume. The material needed to be semi transparent and reveal the bone interior below it. This was achieved by selecting a region in the depth of the bone that would be treated more like a traditional ray march through a gas cloud. Each sample added to the amber opacity and was clamped to a maximum accumulation value. A small amount of noise affected each sample as the result needed to look more glassy than cloudy. The accumulation of each sample differed in this region of the shader as the bone marrow needed to seen through the amber. This meant that instead of just doing an “under” compositing calculation on each sample, we need to use the density of the amber as a shadowing factor on the marrow underneath and use a “darken” operator instead.

4) Bone Surface.

This material as well as the fat and wet needed to work as static materials for the majority of the shots as well as being an element in a dynamically changing surface. The bone surface material contained bump maps and texture maps that were always calculated no matter if the actual surface was visible or not. This was done for several reasons.

a) It is not a good idea in shader to have texture calls and assignments inside “if” statements. You get unprecidictable shading when points on the same micropolygon do not have consistent data to blend between.

This is bad:

```
point Pobj = transform("current", "object", P);
point Pcurr = Pobj + step_size * ray_vector;
if (length(Pcurr - ObjCntr) < surface_depth)
    Ci = interior_colour;
```

```
else
    Ci = exterior_colour;
```

This is better:

```
point Pobj = transform("current", "object", P);
point Pcurr = Pobj + step_size * ray_vector;
float blend = length(Pcurr - ObjCntr) / length(P - ObjCntr);
Ci = (blend * interior_colour) + ((1 - blend) * exterior_colour);
```

How the blend value is filtered and manipulated is entirely up to the user, the important thing is that all points on the micropolygon have the same attribute list to interpolate.

b) Calculating surface texture map values was not a large percentage of render time.

c) Surface shading could used to blend from the outside to the inside. Instead of making a hard transition going from painted textures to procedural ones, a blend region could be employed to fade between the exterior surface and the interior volume.

5) Bone Interior.

This material was used to make up the majority of the texture and surface characteristics of the volume inside the bone. As the disintegration effect went deeper into the bone the density of the volume became more porous. This effect was produced by using a cylindrical coordinate system which meant that when the radius of the cylinder decreased the frequency of the noise increased. This section of the shader was used for the bulk of the volume inside the bone as it handled the calculation of data needed to do shading away from the geometric surface.

This shader was initially based on Clint Hanson's previous work with ray marching in RenderMan. It then proceeded to get more complicated with multiple surface types. We also took Clint's theories to the next level by adding the ray tracing ability mentioned above. With this trace DSO we could ray march through arbitrary shaped objects and not limit ourselves to spheres or geometry hard coded into the shader. The animation curve DSO also mentioned above gave us some key information to start constructing an animating shader and also a lightable volume. Figure 5.4 illustrates how more information than just the curve length was gained from that DSO.

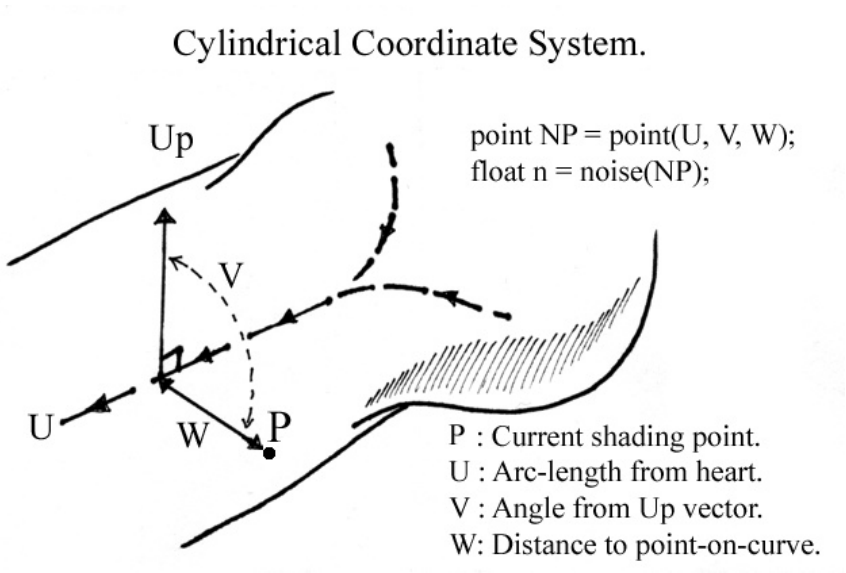


Figure 5.4: Cylindrical Coordinate System

The cylindrical coordinate system in Figure 5.4 was used to make the shaded and animating surface more interesting by adding noise. By comparing P's relationship to U (U being the arc-length of the curve) a point could be turned on or off based on this distance, $(P < U)$, giving a wipe effect along the length of U. By adding the value of W into the equation we could form a shape with a conical leading edge. Going further and adding a noise multiplication to W made the leading edge a noisy conical shape, which was more in the realm of the desired effect. The pseudo-code below gives you some idea of this worked.

```
float blood_length = 190; /*current animation value of the distance from heart to reveal */
float blood_length_offset = 10; /* length of conical leading edge */
float max_bone_depth = 5; /* maximum bone radius to expect */
float max_bone_depth_scaled = max_bone_depth *
                              clamp((U - (blood_length - blood_length_offset)) /
                              (blood_length - blood_length_offset), 0, 1);
if (W <= max_bone_depth_scaled) {
    shade point;
} else {
    keep marching;
}
```

Same with noise factored in:

```
float depth_noise = noise (point(U, V, W)) * noise_scalar;
if (W + depth_noise <= max_bone_scale) {
    shade point;
} else {
    keep marching;
}
```

By changing the value of `blood_length_offset` it was possible to change the length and steepness of the conical profile.

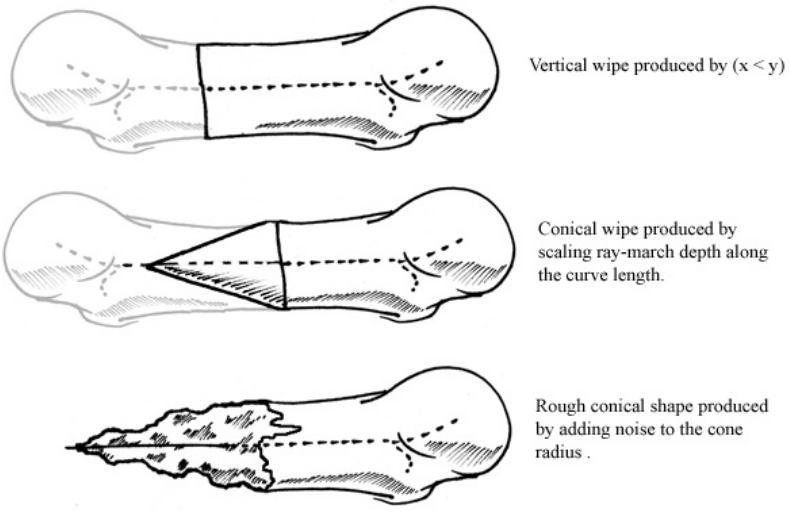


Figure 5.5: Bone wipes.

5.4.2 Going into shader "no man's land"

When writing RenderMan shaders you can get used to taking some high level functionality for granted, such as having `dPdu` and `dPdv` calculated for you giving you the ability to easily call

functions like calculatenormal. Filtered texture map lookups are also much easier when you have `s`, `t`, `u` and `v` right there on the microplygon. When you are dealing with volumes and marching away from the surface you soon begin to realize that these attributes and functions quickly become useless are you are trying to shade the look of a point in space that is really away from the one that RenderMan is trying to shade in screen space. This is easier to understand in diagrammatic form below.

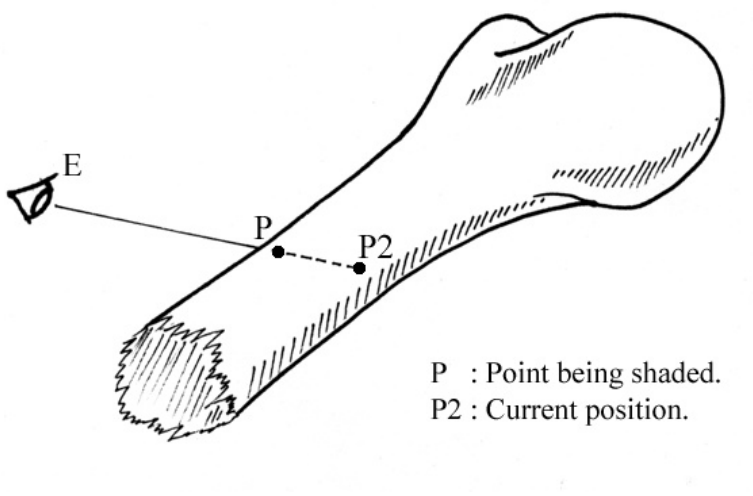


Figure 5.6: Bone ray marching.

Information created at point `P` by RenderMan becomes more and more meaningless the further we march towards point `P2` inside the object. Viewed from location `E`, `P2` occupies the same place on screen as `P` but in object or world space it is in an entirely different location. Executing `calculatenormal(P2)` will not give you the result you expected as `dPdu` and `dPdv` refer to attributes calculated for point `P`.

To remedy this problem it was necessary to calculate a new shading normal from within the shader. This was done by making use of the coordinate system described by U, V and W, mentioned above. In its simplest form the shader created 3 points to build a triangle. `P2` was already calculated as one point on the triangle as this was our current point. The other 2 points were calculated by gaining new noise values for U, V and W at offset locations based on a sampling size parameter. The code below shows how the triangle was constructed.

```
/* size for sub sampling based upon size of ray march step*/
float sampleSz = 0.001; /* the smaller this number the sharper the image*/
/* Current ray marching point in object space */
point PcurrObj;
/* closest point on curve in object space */
point CP0;

/* create vector for direction from the closest point on curve to the current point */
vector Vec0 = vector(normalize(PcurrObj - CP0));
float depth = length(PcurrObj - CP0);

/* curve tangent passed back by curve reading DSO */
vector crvTan;
```

```
/* make a new point further along the curve build P1 from*/
point CP1 = CP0 + (sampleSz * crvTan);
point P2 = CP0 +  (depth  + (noise(point(U,V,W)) * Vec0);
/* P1 is parallel to P2 but further along the curve */
point P1 = CP1 + (depth + (noise(point(U + sampleSz, V, W)) * Vec0);

/* point PP1 is P2 rotated around curve tangent by translating sampleSz to an angle*/
/* Vec1 is now the direction to P1 */
vector Vec1 = normalize(PP1 - CP0);
/* calculate new value for V because PP1 has been rotated around curve tangent */

/* Upvector was set to (0, 0, 1) as most bones are aligned vertically in rest position */
float V0 = Vec1 . Upvector;
point P0 = CP0 + (depth + (noise(point(U, V0, W)) * Vec1;
```

P2->P0 and P2->P1 are now 2 vectors that are perpendicular to each other and describe the triangle. Figure 5.7 shows this triangle and its relationship to the curve.

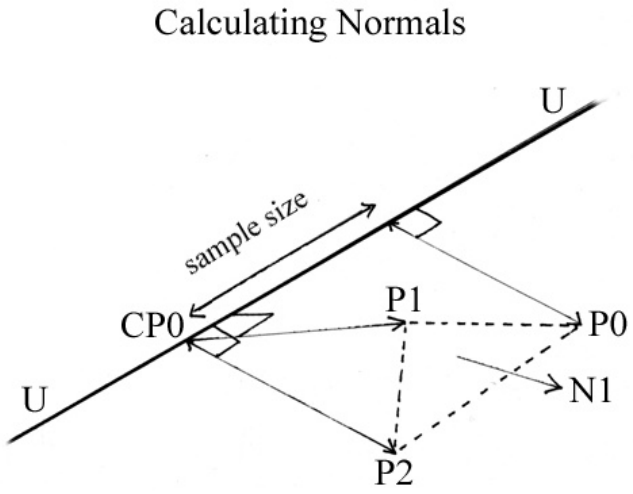


Figure 5.7: Calculating Normals.

By normalizing the vectors and take a cross product we had a vector (N1) that could be transformed into current space, (using ntransform), and then used for lighting calculations. Instead of just calculating the noise gradient, as you would do if you were ray marching through a cloud texture, this method gave us a cylindrical biasing for the normal which made the lighting fit closer to the outer surface lighting. If we had not done this we would have had a surface that dissolved to reveal a cloud and the shape definition would have been lost early on in the effect. When the current ray marched point was near to the geometry surface the normals N1 and N were blended to form a transition from smooth outer surface to a more honeycombed marrow interior.

The method described above can be turned into an iterative process whereby smaller and smaller triangles can be created until a minimum error tolerance has been reached. By constructing a larger triangle with a larger `sample_size` value we could create a lower frequency normal that be used for calculating a “wet look” specular contribution as we did for the geometric surface.

5.5 Rendering The Heart With Volumes

As well as the bones, organs and muscle needed to transform from invisible to visible. Each piece needed to appear as if it had thickness and volume as it made it's transition. Different rendering techniques were used on different body parts, as each piece would pose its own set of unique problems. We will look at the rendering for the heart in this section. The basic ray marching technique will be discussed in some detail, along with some of the problems and features that were implemented in order to produce production quality renders.

The heart was one of the more complex body parts that went through the transformation. It has four hollow chambers that twist around each other, with very thin walls that separated them. The shape is constantly deforming in multiple directions at once. The heart seemed like a good candidate for ray marching despite the complex geometry. In the original story boards, the heart would be transforming full screen, so the renders needed to be able to maintain a high level of detail. Because the shots would be so close to camera, we felt that surface rendering would not work. In the final transformation sequence, because of timing issues, the heart transformation was not featured full screen as originally thought, but the ray marching technique still worked well for it and many other object.

5.5.1 Basic Ray Marcher

Ray Intersection

The first task that the ray marcher needed to do is to define an "in point" and a "out point" for every eye ray that intersected the object. The in and out points basically defining the volume that will be marched though for each pixel of the image. If a ray tracer is used this task can be easy, since the render should know where all the geometry is in the scene. If a ray is fired from any point the render will return any object that was hit by the ray. RenderMan is not a ray tracer so finding the location of surfaces other than the current shading point can be difficult. If the volumes are limited to simple objects such as cubes, spheres, and other primitive shapes, they can be reconstruct in the shader with very little information. If the volumes being render are more complex, the problem of finding the in and out points becomes a lot more difficult in a non ray tracer. In RenderMan there is no way to find these complex surfaces without some outside aid. Luckily RenderMan has the ability to use DSOs that can extend its functionality. It is possible to write a DSO to raytrace objects. This is a good way to taylor a raytracer to meet all the needs that may arise when doing a particular ray march, but may not necessarily be the easiest thing to do. using BMRT as a ray server can also be a good idea with a lot less overhead than writing a custom raytracer. Depth map renders of backside geometry can also be used to get the depth of simple object but may not always work if the object has a lot of overlapping parts.

Marching Through The Volume

Once an "in" and an "out" point are found for the ray, the next step is to break the ray up into small steps along its length. Every time we make a step down the ray, we run a density function which tells us how thick the volume is at that point. To get the position of the sample point we could write something that looks like this.

```
volume_distance = length(out_point - in_point)
num_of_steps = volume_distance/step_size
step_vector = (out_point -  in_point)/num_of_steps
current_position = inpoint
while(current_step < num_of_steps) {
      current_position += step_vector
}
```

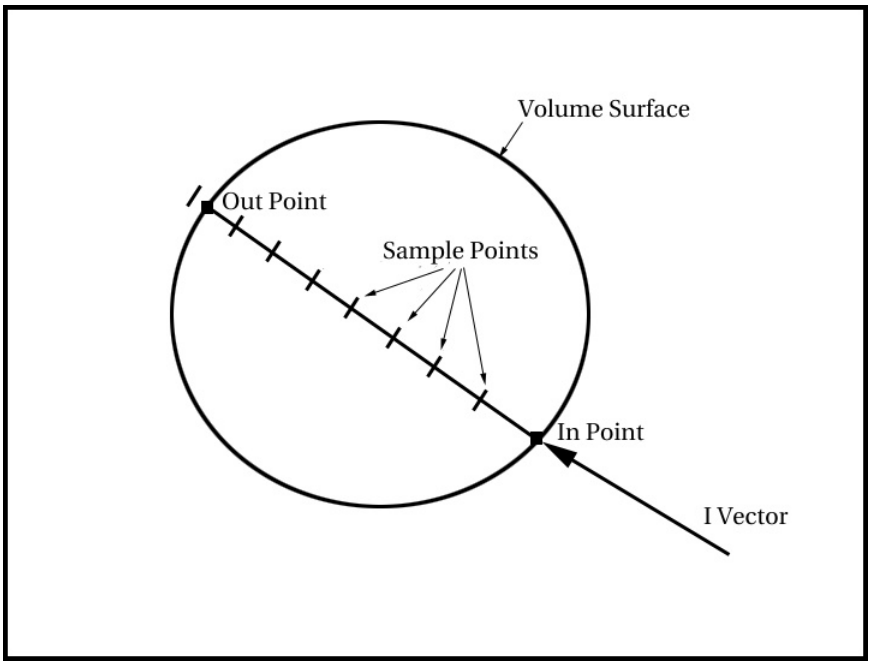


Figure 5.8: Sample points for ray marching.

If bigger steps are used, rendering times can be reduced dramatically, at the cost of low detail and banding artifacts. If smaller step are taken, the render will be smoother, but it can also slow down to a crawl if the step sizes are to small. Two things can be done to help minimize banding in the ray marcher, which will in turn allow for larger steps sizes. The first place banding can occur is at the back surface of the object. If the objects surface is curved or at an angle to the viewer, the number of steps taken through the volume may change from pixel to pixel. If the steps are very small, or the volume is thick enough that we don't see the back end, the problem may never be noticed. In semi transparent volumes, with larger step sizes banding will start to show up. To fix this banding, we can multiply the result of the last sample by distance to the "out" point divided by the step size.

As the ray marcher steps though the volume it may also be producing banding artifacts. If the steps are too big, or the noise in the volume is too detailed or too dense, the discrete steps between the samples start to show up. Jittering the sample points can be a great way to break up this type of banding artifacts. Jittering will produce a noise that looks like grain in the volume, but our eye finds this noise less distracting than the banding. It is not necessary to jitter every step taken though the volume. If the start point for the ray is jittered, the visual effect is the same as if every step was jittered, but only one noise call is made per ray instead of one call per step.

Density Function

At every sample point that the ray marcher takes, it need to run a density function. The sole purpose of this function is to return the thickness of the volume for that sample point. This function can sometimes be very complex if the volume needs to have a lot of detail. The density function will be mostly responsible for the final look if the volume. Along with the density function another function could be run to get color. This can be useful if the color of the volume has a different noise than the density.

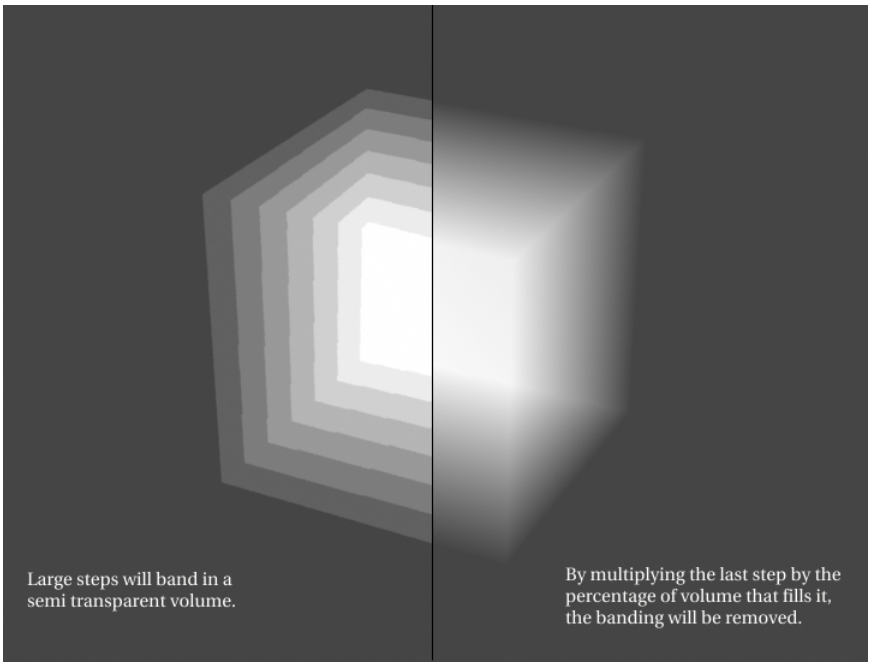


Figure 5.9: Eliminating banding with big step sizes.

Light In The Volume

If the volume needs to have shading, a normal needs to be calculated at every step. To calculate this normal, the density of the current sample point must be taken. Next the sample position is offset in each axis. For each offset another density function is calculated and the density from the original sample is subtracted. Each of these results is put into the corresponding component of a new vector. This new vector is called a gradient vector and always points towards the less dense parts of the volume. This vector works very well for the normal.

```
current_density = get_density(sample_P)
x_grade = get_density(sample_P - offset_in_x)-current_density
y_grade = get_density(sample_P - offset_in_y)-current_density
z_grade = get_density(sample_P - offset_in_z)-current_density
N_vol  = normalize(x_grade,y_grade,z_grade)
```

This normal calculation is very easy to make, but if you look the the above pseudo code you can see that the density function has now been run four times instead of one. If the density calculation is complex there could be an incredible time hit on the render. After the normal has been found for the sample, lighting is run as it normally would be with the exception that an illuminance loop should be used instead of the default diffuse and specular calls. The `diffuse()` and `specular()` functions assume that the light will be calculated at `P`. In the case of the volume render, the light should be calculated at the current sample point.

Compositing Samples

After the color and opacity has been calculated at a sample point, the value must now be added to the total accumulated volume of the ray. In some cases, the color and opacity can just be summed up to produce a volume that gets brighter and more dense as each step is sampled. This produces a

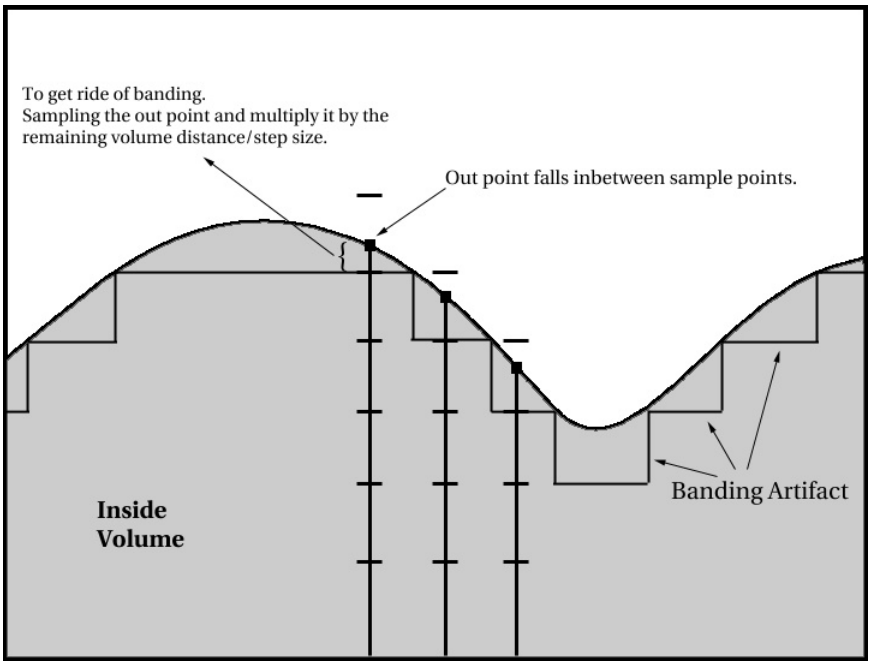


Figure 5.10:

volume which looks like it has some self illumination. If the render needs to match existing lighting from a plate, a better composite may need to be done. A simple over composite of each sample can produce nicer lighting for more solid volumes and will retain dark areas and give better depth cueing as the volume moves. Once the samples are composited, they are essentially collected at the shading point P on the object. This fact is important to keep in mind. RenderMan shades surfaces, not volumes. which means that we may run into trouble later when trying to do things like motion blur or shadows.

After all the steps are complete, the ray march has a structure that looks something like this:

```
get in and out intersection points of ray I with object
divide ray into sample steps
while (current sample < total number of samples) {
      call density function
      call color function
      calculate normal
      calculate lighting
      composite samples
      jump to next sample point
}
set output color to the totaled color samples
set output opacity to the totaled density samples
```

5.5.2 Speeding Up A Ray Marcher

It is easy to produce impressive results with a very simple ray marcher. Because it can be so easy to set up simple tests that show a lot of potential, we can sometimes be lured down the path of using a ray marcher on a production, only to abandon it later because of speed issues. As a simple ray marcher is extended with all the things it needs to produce production quality images, we find that

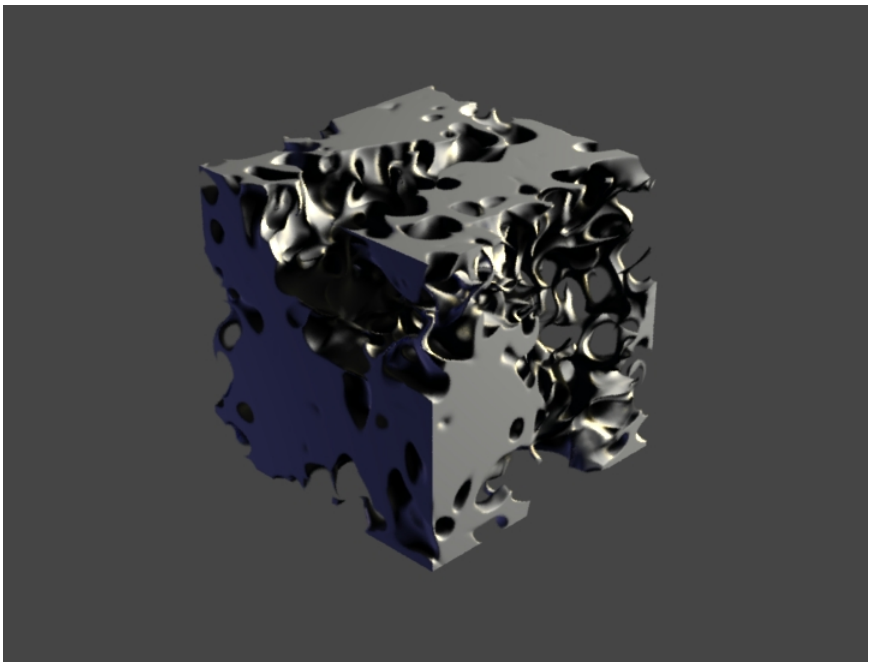


Figure 5.11: Shading.

it can become bogged down very quickly to the point of being impractical. The ray marcher is very repetitive, because of this, it sometimes only takes one slow calculation placed in a frequently run part of the shader to slow the render down to unbearable speeds. This was a concern when deciding to use ray marching for the transforming sequences in *Hollow Man*. Everything that needed to be added to the ray marcher had to be carefully considered based on the benefit of the feature compared to the speed. Optimizing the ray marcher became an essential part of the success or failure of this technique.

Density Function

The density function will probably be the place where the render is most likely to get bogged down. This function usually starts out very simple, but quickly grows in complexity as the look of the volume is refined. It is usually filled with fractal noises and power functions or what ever else is needed to achieve the desired look. This function is also most likely to be called more than any other function. As an example if a density function was created that took .01 of a second to calculate, at first this may not seam like that much time. If we render a volume that fills a 720x546 video frame, with an average of 100 samples per pixel. the function would get called 39,312,000 times in an unoptimized ray marcher. This would take 109 hours to run. If we added shading calculations to this, the time to run the destiny functions would increase four times, taking 436 hours to run. Add a raytraced shadow that runs another 100 density samples per step and it will now take 11,247 hours to calculate all the density functions. In comparison the .01 second function would take 1.09 hours if it was calculated only once per pixel in the video frame. In this worse case scenario the ray marcher could take over ten thousand times the render time of the surface render.

Minimizing Density Calls And Other Calls

In the renders that were done on HollowMan, the volume would animate from a totally empty void that would grow across the object with an abrupt transition from empty to solid densities. In the early frames of the animation there was a lot of empty space. This empty volume would take a incredible amount of time to render, with a mostly black frame as the final result. The easy way to speed up these empty areas was the addition of a conditional inside the step loop that checks the density and only executes any other calculations if it is grater than zero.

```
while (taking samples) {
      get density
      if(densiy > 0){
              get color
              get normal
              do shading
              composite samples
      }
      goto next sample
}
```

With this simple addition the render runs quicker through the empty areas. But the time which it takes to run the density function in these empty areas still adds up. In most of the volume renders done on *Hollow Man* the density function could be separated into two phases. The first calculating a low frequency animated pass that would drive where the volume was on a particular frame. The second was a set for fractal noises which created the detail of the volume. It was possible to split these into two functions, the first one which we will call the `active_volume` and the second one the actual "density" function.

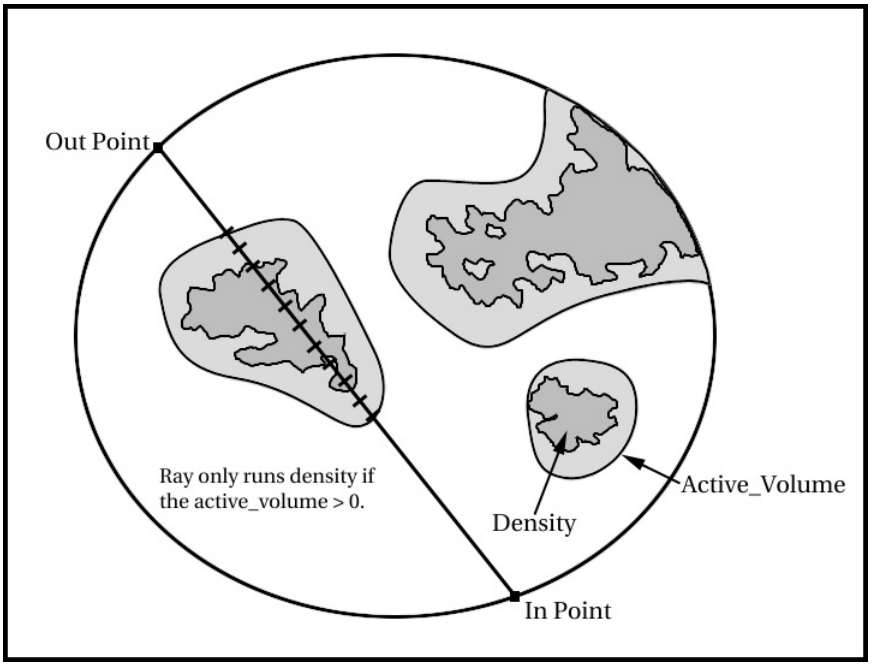


Figure 5.12: Active volumes.

The `active_volume` can be any low frequency function that returns values greater that zero in places where there is a chance that the density function might produce results greater than zero. If

the active_volume is 0, then there is no chance that the density will return anything. When the density function is called it will first call the active_volume to see if it should continue running though the rest of it's function.

```
function get_active_volume()
{
      return quick low frequency pattern
}

function get_density()
{
      get_active_volume
      if (active_volume > 0) {
              run slow fractal noises
      }
      return density * active_volume
}
```

In this case the density is multiplied by the active_volume to ensure that there can not be any density grater than zero outside the active part of the volume. Now as the ray marcher steps though empty space the density function is only run when there is a good chance that it will produce more than a black void. Once these two speed ups were implemented, early frames of animation which were taking hours to render, were now taking minutes.

Minimizing Step Calls

Another way to speed up the ray marcher is to cut down on the number of sample points all together. The easiest place were steps can be cut out is when the density of the volume adds up to one. Once the density of and object has reaches one, there is no chance of seeing any thing behind it at that point. The ray march can stop early, saving valuable time, especially on very dens parts of the volume. The other place where it makes sense to cut down on the sample points is in the empty parts of the volume. At fist this sounds easy to do, take big steps until you hit some density. Once you hit something, backup a little and start marching through with a finer step size to get the detail. This does not work quite that easy though. As large steps are taken through the volume there is a chance of totally missing small details all together. Not knowing that the step has just jumped over a detail, there is no indication that the area should be run though in finer detail. If large steps are used to only look for the active_volume there is less of a chance of missing pieces because this function should be a lower frequency than the density function, but near the edges there will still be pieces that are sometimes missed. The easiest way around this is to add a border around the active function, so the bigger steps don't have to actually hit the active area before it reduces the step size. The bigger the border the less chance of errors, but the less the time savings as well.

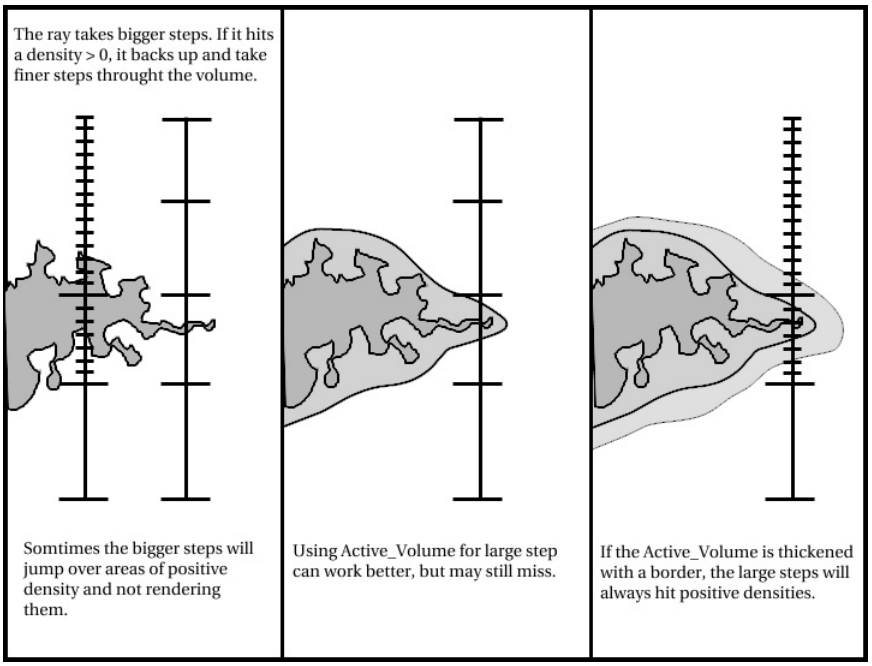


Figure 5.13:

Listing 5.1 is a surface shader that implements a simple ray marcher.

Listing 5.1: `srf_vol_cube.sl`

```
/* srf_vol_cube - Brian Steiner - Sony Pictures Imageworks

   This shader raytraces a box that is one unit in size,
   and then ray marches through the volume
   StepSize            - distance between sample points.
   StepJitter          - 0-1 jitter the sample position.
   Density             - volume thickes per unit.
   Epsilon             - offset for calculating gradient normal.
   Vol_Mult, Vol_Offset - animation controls.
   Do_Shading          - if 1, shading will be calculated.
   SurfNormalDepth     - the mixing depth from surface
                         normal to volume normal.
   Additive            - if 1 add samples, if 0 over samples .
   ShowActiveVol       - if 1 show the active volume instead of density.
   RunShadowPass       - set to 1 if running a shadow pass.
*/


/*------------------------------------------------------*/
/* fnc_traceBox returns an intersection point on a box */
point
fnc_traceBox (float XMin;
              float XMax;
              float YMin;
              float YMax;
              float ZMin;
              float ZMax;
              float idx;
              string refractSpace;)
{
    extern point P;
```

```
    extern vector I;
    extern normal N;
    vector Rd,Ro;
    point Ri;
    vector Pn;
    float D,T;
    float TMin = 1000000;
    vector IN;
    normal NN;

    IN = normalize(I);
    NN = normalize(N);
    Rd = vtransform(refractSpace,IN);
    Ro = transform(refractSpace,P);

    /*plane_z_min*/
    Pn = (0,0,1);
    D = ZMin * -1;
    T = -(Pn . Ro + D) / (Pn . Rd);
    if(T > 0){
        TMin = T;
        Ri = Ro + T*Rd;
    }
    /*plane_z_max*/
    Pn = (0,0,-1);
    D = ZMax;
    T = -(Pn . Ro + D) / (Pn . Rd);
    if(T > 0 && T < TMin){
        TMin = T;
        Ri = Ro + T*Rd;
    }
    /*plane_x_min*/
    Pn = (1,0,0);
    D = XMin * -1;
    T = -(Pn . Ro + D) / (Pn . Rd);
    if(T > 0 && T < TMin){
        TMin = T;
        Ri = Ro + T*Rd;
    }
    /*plane_x_max*/
    Pn = (-1,0,0);
    D = XMax;
    T = -(Pn . Ro + D) / (Pn . Rd);
    if(T > 0 && T < TMin){
        TMin = T;
        Ri = Ro + T*Rd;
    }
    /*plane_y_min*/
    Pn = (0,1,0);
    D = YMin *-1;
    T = -(Pn . Ro + D) / (Pn . Rd);
    if(T > 0 && T < TMin){
        TMin = T;
        Ri = Ro + T*Rd;
    }
    /*plane_y_max*/
    Pn = (0,-1,0);
    D = YMax;
    T = -(Pn . Ro + D) / (Pn . Rd);
    if(T > 0 && T < TMin){
        TMin = T;
        Ri = Ro + T*Rd;
    }
    return Ri;
}


/*-------------------------------------------------*/
```

```
/* active_volume - controls animation in the volume */
float
active_volume(point Pos; float vol_mult, vol_offset;)
{
    return (noise((Pos+30.445)*2)-.5+vol_offset)*vol_mult;
}



/*-------------------------------------------------------*/
/* density function will return the final volume density */
float
get_density(point Pos; float vol_mult, vol_offset;)
{
    float dens = 0;
    float activeVol = 0;
    float offset_active = .1;
    float mult_active = 20;
    activeVol = active_volume(Pos,vol_mult,vol_offset);
    dens = pow(1-abs(noise(Pos*7)*2-1),3);
    dens += pow(1-abs(noise((Pos+24.72)*7)*2-1),3);
    return activeVol + (dens-2);
}


/*---------------------------------------*/
/* normal calculation inside the volume */
normal calcGradeNorm(point Pos; float vol_mult, vol_offset, dens, epsilon;)
{
    normal Nd;
    Nd = normal (get_density(point (xcomp(Pos) - epsilon, ycomp(Pos),
                                    zcomp(Pos)),vol_mult,vol_offset) - dens,
                 get_density(point (xcomp(Pos),
                                    ycomp(Pos) - epsilon, zcomp(Pos)),vol_mult,vol_offset) - dens,
                 get_density(point (xcomp(Pos),
                                    ycomp(Pos), zcomp(Pos) - epsilon),vol_mult,vol_offset) - dens);
    Nd = ntransform("object","current",Nd);
    return Nd;
}



/*-----------------------------------------------*/
/* shading function returns diffuse ans specular */
void get_shading (point Pos;
                  normal Nf;
                  vector V;
                  float Roughness;
                  output color diff;
                  output color spec;)
{
    extern vector L;
    extern color Cl;
    diff = 0;
    spec = 0;
    illuminance (Pos, Nf, radians(90)){
        diff += Cl * max(0,normalize(L).Nf);
        spec += Cl * specularbrdf(L, Nf, V, Roughness);
    }
}



/*-------------*/
/* nomal mixer */
normal
fnc_normalMix (normal N1; normal N2; float mixer)
{
```

```
    float N1_mag = 1;
    float N2_mag = 1;
    normal NN1 = normalize(N1);
    normal NN2 = normalize(N2);
    normal result;
    N1_mag *= 1-mixer;
    N2_mag *= mixer;
    result = normalize(NN1 * N1_mag + NN2 * N2_mag);
    return result;
}



/*---------------------------*/
/* main ray marching shader  */
surface
srf_vol_cube(float  StepSize       = 1;
             float  StepJitter     = 0;
             float  Density        = 1;
             float  Epsilon        = .001;
             float  Vol_Mult       = 1;
             float  Vol_Offset     = 0;
             float  Do_Shading     = 1;
             float  SurfNormalDepth = .05;
             float  Additive       = 1;
             float  ShowActiveVol  = 0;
             float  RunShadowPass  = 0;
    )
{
    point  inPoint_obj  = transform("object",P);
    point  outPoint_obj = fnc_traceBox(-.501,.501,-.501,.501,-.501,.501,1,"object");
    vector V  = normalize(-I);
    normal Nf = normalize(N);
    float  Roughness    = .21;
    color  diff = 1;
    color  spec = 0;
    float  vol_length   = length(outPoint_obj-inPoint_obj);
    float  numOfSteps   = vol_length/StepSize;
    vector step_obj     = (outPoint_obj-inPoint_obj)/numOfSteps;
    vector step_cur     = vtransform("object","current",step_obj);
    float  curStep      = 0;
    float  density_sum  = 0;
    color  color_sum    = 0;
    float  shad_sum     = 0;
    float  remainder    = 100;
    float  cur_density  = 0;
    color  cur_color    = 0;
    float  density      = StepSize * Density;
    float  jitter       = (random() - .5) * StepJitter;
    float  cur_depth    = 0;
    point  Pcur_obj     = inPoint_obj + jitter * step_obj;
    point  Pcur        = P  + jitter * step_cur;

    Oi = 0;
    Ci = 0;

    /*---------------------------*/
    /*     step loop             */
    while(curStep < numOfSteps && density_sum < 1){
        cur_density = 0;
        cur_color = 0;

        /*--- Run Density Function ---*/
        if(ShowActiveVol == 1)
            cur_density = active_volume(Pcur_obj,Vol_Mult,Vol_Offset);
        else
            cur_density = get_density(Pcur_obj,Vol_Mult,Vol_Offset);

        /*--- If Density > 0 Run The Rest Of The Loop ---*/
```

```
        if(cur_density > 0 && RunShadowPass == 0){
            cur_color = cur_density;
            cur_color = 1;
            if(Do_Shading > 0){
                if(cur_depth > 0){
                    normal Vol_Nf = calcGradeNorm(Pcur_obj,Vol_Mult,Vol_Offset,
                                                  cur_density,Epsilon);
                    Vol_Nf = normalize(Vol_Nf);
                    Nf = fnc_normalMix(Nf,Vol_Nf,clamp(cur_depth/SurfNormalDepth,0,1));
                }
                get_shading(Pcur,Nf,V,Roughness,diff,spec);
            }
            cur_color = (cur_color * diff) + spec*(1,.8,.2);

            /*---- if sample is not a full step ----*/
            remainder = numOfSteps - curStep;
            if(remainder < 1){
                cur_density *= remainder;
            }

            cur_density *= density;
            cur_color *= clamp(cur_density,0,1);

            /*---- Composite Sample ----*/
            if(Additive > 0){
                /* Just Add Up Density */
                density_sum += max(0,cur_density);
                color_sum += clamp(cur_color,color 0, color 1);
            }
            else{
                /* Do Over Instead of Add */
                cur_color = clamp(cur_color,color 0,color 1);
                cur_density = clamp(cur_density,0,1);
                color_sum = ( cur_color) * (1-density_sum) + color_sum;
                density_sum = (cur_density) * (1-density_sum) + density_sum;
            }
        }
        else {
            /* if Shadow Pass */
            if(Additive > 0){
                shad_sum += max(0,cur_density);
            }
            else{
                cur_density = clamp(cur_density,0,1);
                shad_sum = (cur_density) * (1-shad_sum) + shad_sum;
            }
            if(shad_sum >= .5){
                density_sum = 1;
                color_sum = 1;
            }
            P = Pcur;
            /* Displace Point To Current Sample */
        }

        /* jump to the next sample point */
        curStep += 1;
        Pcur_obj += step_obj;
        Pcur += step_cur;
        cur_depth += StepSize;
    }

    Ci = color_sum;
    Oi = density_sum;
}
```

5.5.3 Integration

Once we have got the basic ray marcher running in a some what optimized form. It's time to start thinking about how we are gone to integrate the volume with our specific geometry and fit the final renders into the plates. The first problem was to figure out the best way to trace the geometry. In the case of the heart the object has a very complex shape. It is an object that has two hollow chambers in the main part and two other hollow chambers on either side.

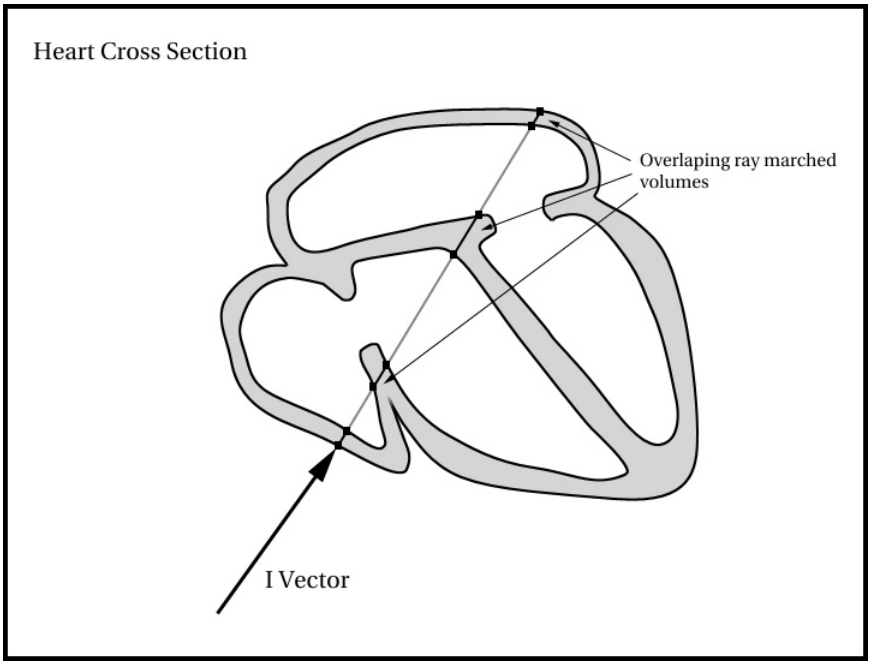


Figure 5.14: Heart cross-section.

Raytracing

Because all the chambers of the heart move in different directions at different times, and some of the surfaces were so close, there was concern about surfaces interpenetrating. Because of possible interpenetration, the first version of the ray marcher ran only on the closest point `P` for any `I` vector. From that closes point it would raytrace all the surface behind it. The shader kept track of when it entered a volume or left a volume by checking to see if the normal was pointing in the same or opposite direction of the ray. A layer variable was kept, to which the number one was added to when it went into the volume and one was subtracted as it left the volume. The ray marcher was then told to only march on a specific layer.

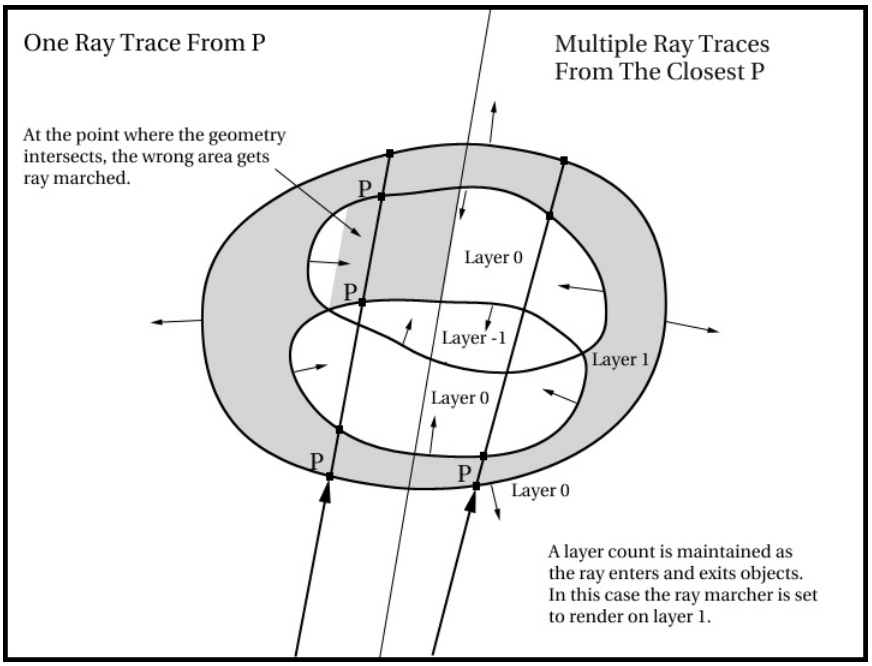


Figure 5.15: Volume booleans.

This made it possible to maintain the proper volumes even if they interpenetrated with other volumes. In the end this technique created more problems that it solved. Motion blur, displacement, and transparency became difficult to deal with. It seamed to make more sense to just insure the the surface would never interpenetrate and only do one ray march per shading point. The one reason this layered approach is mentioned, is that it can be used as a way of making Booleans on very simple shapes to get interesting volumes.

5.5.4 Moving And Deforming Geometry

Making the volume work with deforming and translating geometry was also needed to complete the shots of transforming organs. Most of the deformations on the model are usually done with clustering CV's to skeleton joints and blend shapes. The problem with these deformations working in the volume is that they are only surface effects. Each CV holds the information that tells it how to deform. What skeleton am I attached to? What weighting do I have? Where is my next blend shape position? As soon as we try to deform a sample point which lies off of the surface these questions become very unclear. Now, not only does the deformation need to be recreated, but deciding which mixture CVs get the information from needs to be figured out as well. Lattices are a lot easier to deal with inside a volume, because instead of deforming only the CVs, they deform the space and the CVs just come along for the ride. The nice thing about this is that you can add a point any where in the base lattice and it will automatically jump to its position in the deformed lattice. This means that any sample point can be deformed inside the volume and no information needs to be known about the surface. So the decision was made to deform ray marched geometry with lattices. When rendering in the volume we actually want to do the inverse of the lattice deformation. A DSO was written for RenderMan that could read the positions of the base and deformed lattice and return any point to it's undeformed position in the volume. The thought was that this DSO would need to be run at every step to accurately represents the deformed volume, but after some tests it was apparent that only the begin and end points needed to be run through the inverse lattice, then the steps could

be linearly interpolated through the volume. This worked very well for the heart's deformations because the walls were thin, so the inner volume would naturally deform in a similar way as the surface. Since the inverse deformation was now run only on the surface, Pref could be used instead, which could use other types deformation as well. This does, however, have limitations. Depending on how the object is deformed, a ray traveling through it may be a curved line, instead of a straight line from in point to exit point. If the volume is radically deformed, then running the inverse lattice at every sample point may be the better solution.

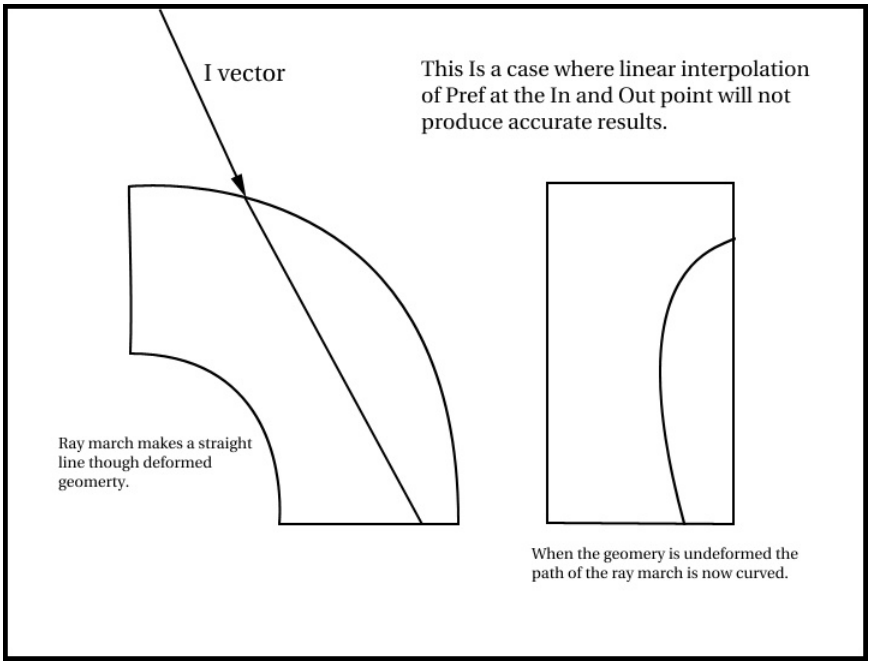


Figure 5.16: Ray marching through deformed volumes.

Motion Blur

Motion blur inside a volume can be very hard to do. One way in which it could be achieved in the volume is to sample different slices of time at everly step and average them together. Provided that you average enough slices you could get very accurate motion blur, but the render times would be extremely long. If the actual object that contains the volume is moving, we can use RenderMan's motion blur and get very good results. If the object is translating, the motion blur is fairly accurate. If the object is rotating the volume does not blur correctly. Since all the march samples are projected onto the front surface the blur will only go in one direction. If the rotation is blurred accurately, the back half should be blurring in the opposite direction than the front. Even though RenderMan's motion blur is not always accurate in the volume it looked good in most situations.

Shadow Maps

In order for our renders to fit into the plate, shadows need to be rendered as well. If the volume is very transparent ray marching the shadows may be the only option to get an accurate shadow. Although this can produce very impressive results, it is a big hit on rendering time. The other problem with ray marching shadows is that they work great for self shadowing, but not so great for cast shadows onto other objects. In the case of the heart, the volume was either totally empty or

totally solid. There were not very many semi transparent regions. This made the object perfect for shadow maps. Because shadow maps are either on or off, they are good at representing the solid volumes better than the soft semi transparent ones. When RenderMan calculates the shadow map it uses the distance to P as it's zdepth value. This is not necessarily the position where the volume becomes solid. In order to get the proper zdepth, P needs to be displaced to the sample position were the volume turned solid. This displacement can be very abrupt and will most likely tear apart the geometry, but since the displacement is done in the direction of the I vector, any tearing or stretching is unnoticeable from the rendering view.

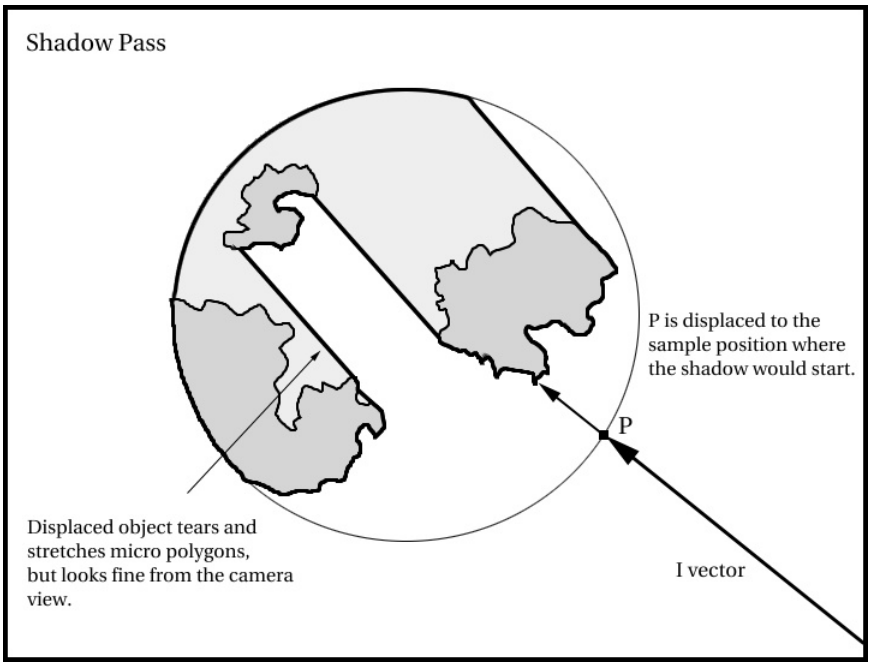


Figure 5.17: Shadow pass.

Animation In The Volume

Animation was another consideration, when we tried to figure out if ray marching would be a viable solution. We needed to have something that was controllable, and quick enough to test many iterations. Early on it was decided that curves could be used to fill the space of the volume and drive the animation. Each curve had the distance from the point of the original injection where the transformation started from. Then the animation was controlled by a parameter that told the shader how far down the blood stream the effect had gotten too. Some custom tools were written to help connect groups of curves and keep track of the arc length of all the connected curves. Individual or groups of curves could also be scaled or offset to adjust the starting point and speed at which a curve would animate through the volume. The curves were then converted into particles that would lie along the length of the curve, with it's arc length mapped to them. At render time the shader called a DSO that read through the particle and figured out which particles were within in a certain radius of the current volume sample. If the animation curve was greater than the particle's mapped arc length, then the particle receives a radius and strength which would get bigger as the animation curve got greater than the arc length of the particle. The particles are then summed together so that they blend with other particles from the same curve, and from different curves in the case of the heart. Finally the DSO returns a float that would be zero in the places where there was empty space and one were

the volume was fully on. In the case of the heart this process would be run in the function that found the `active_volume`. If the function returned a number greater the zero then the density function would be run.

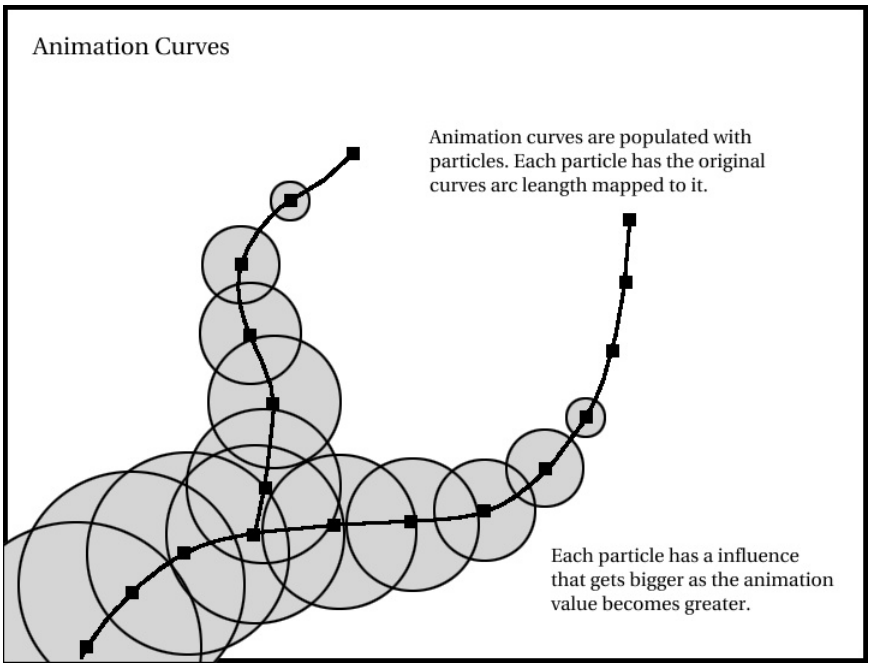


Figure 5.18: Animation curves.

The curves weren't always necessarily used for all the animations. In some causes simple falloff functions or grades were written in the shader. Then the falloff function would be transformed into coordinate systems that were animated inside Maya. This made it easy to layer more animation onto existing curve animation, or layer a bunch of these simple fields on top of each other to get a complex animation without the curves.

Volume Age

When the heart animated on, the muscle formed and filled out the volume. A few seconds later the muscle was covered over with a smooth outer casing and then a layer of fat followed. In order to run these secondary effects through the heart, the volume needed to have some sort of age associated with it. By having densities become more dense as the animation progresses we can tell which areas of the surface are older than others by looking at the density. By having densities greater than one we could trigger the secondary effects at higher values like three, ten, or a hundred. The higher the density needed to be to trigger the effect, the longer the delay would be until it happened in the animation. when the actual density of the volume was calculated a clamped version of the density was used.

5.5.5 Conclusion

Surface rendering can not always accurately represent objects that have inner volumetric detail. There are many interesting ways to to fake thickness and volume with surfaces. Displacement, transparency, and other shading tricks can often be used to create fast and believable objects that feel as if they have volume. In a production the fastest techniques are usually preferred, but sometimes

there is no good substitute for actually calculating the volume for an object. Ray marching can be a good way to accurately calculate these volumes, and it can be implemented in a surface renderer like RenderMan. In a ray marcher some features we usually take for granted may become a lot harder to implement, debugging becomes a little trickier, and render time can get a lot longer. If we are willing to bear with and/or overcome these obstacles, the final results can be well worth the effort.

Bibliography

Arvo, J., Cook, R., Glassner, A., Haines, E., Hanrahan, P., Heckbert, P., Kirk, D.; 1989. *An Introduction To Ray Tracing.* San Diego: Academic Press Limited.

Ebert, D., Musgrave, F., Peachey, D., Perlin, K., Worley, S. 1994. *Texturing And Modeling A Procedural Approach.* Cambirdge: AP Professional.

Color Atlas of Anatomy : Rohen, Yokochi and Lütjen-Drecoll Fourth Edition published by Williams & Williams.

Credits

Peter Polombi
Laurent Chabonnell
Tasso Lappas
Clint Hanson
Brian Steiner

Chapter 6

Using Maya with RenderMan on *Final Fantasy: The Spirits Within*

Kevin Bjorke,
Square USA
bjorke@squareusa.com

6.1 Introduction

Final Fantasy was a project that brought together a number of different technologies that hadnt been used together before, or had never been used on the scale at which we deployed them. We brought together aspects of game technology with movie technology, brought together motion capture and keyframed animation on a feature-film scale, and brought together multiple rendering technologies.

We used RenderMan — specifically *PRMan*— to render most of the elements in *Final Fantasy*. But *PRMan* was not the initial choice to render this story. Initially, it was believed that the entire film would be rendered using the renderer included in Maya 1.0. A short test film was made with Maya, and a few key scenes from the final movie were already produced before the studio began looking at RenderMan as a way to accelerate some of the rendering being done for visual effects on the film. In late 1998, a few quick tests were done, *PRMan* rendered the tests more quickly than Maya, and a spontaneous executive decision changed our course from a project rendered in Maya with perhaps a little RenderMan for VFX into a project rendered in primarily using RenderMan under Maya control.

There were a few problems with this, of course — among them, that no one really knew if we could get Maya (by then at version 1.5) to work properly with RenderMan. Because of time constraints, it was crucial to get things functional as quickly as possible, so we chose to purchase Pixars *MtoR* software and integrate it into our general Maya-based workflow.

Purchasing *MtoR*, by itself, did not mean instant compatibility. Even the definition of "compatibility" was difficult to nail down at the time — while most stakeholders were happy as long as a high-quality image rendered quickly, and others who were eager to use prman features that they hadnt had available in Maya, there were also others who at first would be satisfied only if the image rendered in RenderMan was *identical* to the image rendered in Maya.

Because *Final Fantasy* was already in production, there was little time for prep work and design of the underlying software infrastructure. The scenes and models already existed, and had to be moved from Maya into RenderMan immediately. In classic "production software" fashion, many of the tools slammed together in those first days, by programmers who were working on RenderMan

in their "spare time" alongside other production tasks, became the well-worn foundations of many of the larger and more-capable systems that we ultimately used to finish the film.

Its a straightforward technical task to compare algorithms between renderers; its relatively straightforward to convert a Maya shader network into a shade tree for RenderMan shading; its even sort of easy to emulate the Maya Blinn and Phong functions and to get a visual result that looks very, very, close to a frame rendered in Maya. The goals are relatively clear-cut. Whats harder, *much* harder, is to reprogram the *users*, to wean them out of an off-the-shelf environment with a small number of ready-made rendering solutions into the much broader world of fully-controllable rendering — to open up a much larger box of paints.

And whats harder still is to decide just which parts of that paint box you really want to open, and which ones to leave shut. These questions ultimately came to color every part of our RenderMan-related processes for *Final Fantasy*.

6.1.1 Using RenderMan

Many of the typical "rules" familiar to RenderMan users were promptly tossed out the door, since we had to be backward-compatible with many existing models, textures, animation, cameras, and users. Almost all of the models were polygonal, and usually very densely tessellated, with few uses of cubic patches or NURBs.

Most of the staff were adept with Maya, but had never used RenderMan before. Very few had mathematical or programming backgrounds, so our approach was to walk a line between pushing the coolest and most innovative techniques, while capitalizing on the existing staff skill sets.

During the course of production we used Maya versions 1.5 through 2.5. Maya 3 wasnt used in the core production and so neither was RAT4 or *Slim*. Because of the size of the production, we began quite some time ago — while *PRMan* 3.8.0.11 and Maya 1.5 may not seem like cutting-edge tools today, two and a half years ago, they were the latest available. Many of the refinements, tricks and workarounds we developed and struggled with using RAT3 have happily found their way into the newer RAT4 releases.

Compared to projects like *Toy Story*, very few different shaders were used in *Final Fantasy*. Since users who could create shaders themselves were rare, a small number of general-purpose shaders were used that would provide a close emulation of the "one-stop-shop" functionality to which users had already become accustomed using 3D packages. As you will see, we used only a single shader type for the hair of every character, and a single surface shader for every characters face. We used only four different surface shaders for virtually every object in sets like the Barrier Control below.

This method was more like that used by packaged software — a small number of shaders with lots of parameters, many of which were unused for any given shot. While this made the shader GUI cumbersome, made maintenance and localized upgrades difficult, and raised the memory cost for shader loading at render time, it kept the learning curve relatively short for modelers and texture artists. As a general policy, it was determined that an hour of artist time was far more expensive to lose than many hours of computer time. We could always buy more computers!

Besides the "core" use of RenderMan to create images of our characters and sets, many more-esoteric uses for it were devised by the VFX crew for the creation of everything from particle clouds and heat signatures to the writhing entrails of slain aliens. To describe all of them would take several more sets of course notes, so for the present Ill concentrate on the more mundane uses, such as the human characters.

6.1.2 Layering

Like most modern productions, *Final Fantasy* was rendered in multiple layers, even though it was 100% CG and didnt need to be mixed with live-action photography. One reason for the layered

Figure 6.1: While there were many shader instances, only four different shaders were used for all surfaces in this image.

approach was the large memory footprint of the models — characters like Aki and Gray, with many animation controls and dense geometry, were simply impractical to render together except in rare circumstances (such as when Gray is carrying Aki).

Some models, like the barrier control above, needed to be split into layers so that subsequent effects layers could be sandwiched in between the different sections. For example, an explosion inserted behind the tower, but in front of the pipes (a Matte object could have also been used in this instance, but application of the towers alpha channel was more efficient).

Finally, so many artistic possibilities are available using modern compositing tools that layer breakdown provided a great deal of freedom for the director to alter color, contrast, and even to paint-in lighting or shadows onto some elements long after the 3D rendering process was complete. We decided to always relegate some aspects of production to compositing — for example, the application of atmosphere depth shaders was avoided, since this could be trivially added and tweaked in comp. We likewise rendered most reflections as separable elements, so that final balance could be adjusted once all the image elements were prepared.

6.1.3 Using MtoR

MtoR converts Maya scenes to RIB in a very simple and predictable way. While the later versions of MtoR have provided more control over RIB conversion, the early versions had no direct support for many of the features of RenderMan that we would like to use, and some parts of the Maya file-referencing scheme worked at right angles to the file structure anticipated by MtoR.

Fortunately, MtoR did have a “RIB box” mechanism, which we could attach to Maya nodes and use to insert arbitrary snippets of RIB into parts of the scene. It also contained a Tcl interpreter, which had been originally included mainly to provide math expression parsing for animated parameters in Glimpse. MtoR runs the Tcl *subst* function on all strings before writing them into the RIB stream, so we could include Tcl commands directly into the RIB boxes, simply by enclosing them [in brackets].

Simple experiments quickly turned into large ones, and it was apparent that we could use Tcls

"source" command to read-in not just the occasional math expression, but entire Tcl programs. Pixar was even nice enough to add an environment variable to MtoR (`$RAT_TOR_EXTENSIONS`) so that we could get our programs to load automatically, each time the MtoR plugin was loaded. We called this collection of mini programs `movie_init`.

A few `movie_init` functions were just utilities that could be used as Glimpse parameter expressions, but most of them were tied closely to the RIB-generation process, and `movie_init` was eventually infused into every shot in the movie.

By using Tcl and `movie_init`, we had four different programmable contexts for rendering:

1. Mel scripts, which usually executed "static" changes to the overall shot data.
2. Maya expressions (and sometimes Mel script nodes), which dynamically altered channel values on a frame-by-frame basis.
3. Tcl and `movie_init`, which provided control on a per-RIB-pass and per-appearance level during RIB generation.
4. The RenderMan shading language, which executed during the render process.

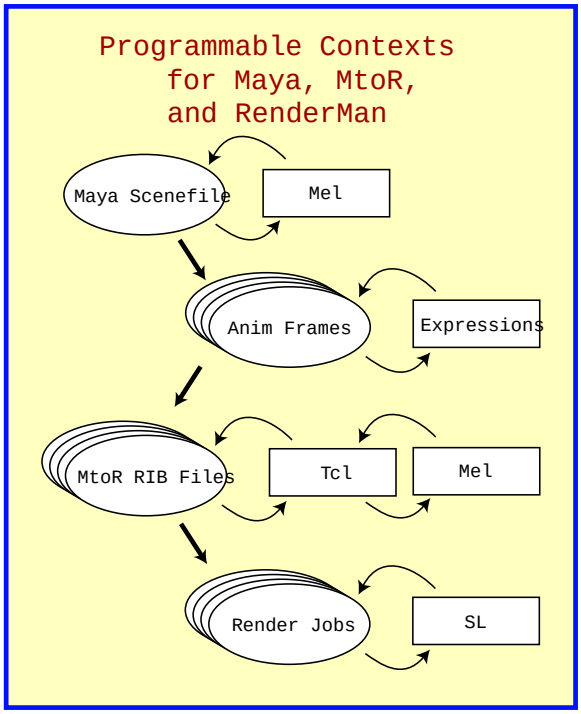


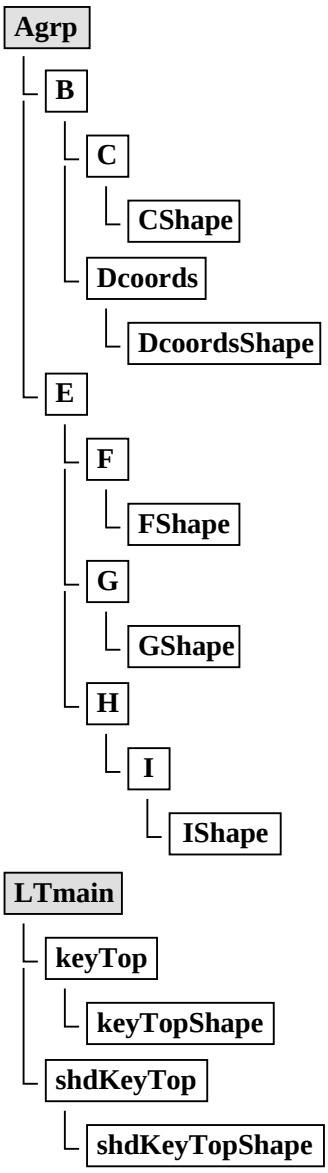
Figure 6.2: Contexts.

`Movie_Init` Architecture

RIB files made by *MtoR* are extremely predictable in structure, and `movie_init` follows that structure. The structure is sketched below. We always used "subframe" encoding, meaning that RIB

descriptions for shadow and environment maps were written as individual files, and we always wrote one frame per RIB. This granualrity made it easier to control and easier to distribute render jobs between multiple CPUs.

Maya Hierarchy



Simplified RIB Output (RGB pass)

```
# Declarations
FrameBegin
   # Camera & Display Options
   # Frame RIB Box
   [movie_init]
   WorldBegin
      # Light Declarations
      TransformBegin
         Identifier "keyTopShape"
         LightSource 1 "sqL6tiny"
      TransformEnd
      # World RIB Box
      [initIllumStates]
      AttributeBegin
         Identifier "|Agrp|B|C|CShape"
         Surface "sqMateriaV"
         # Attached RIB Box
         [linkMe]
         # Geometry Data
      AttributeEnd
      AttributeBegin
         Identifier "|Agrp|E|F|FShape"
         Surface "sqMateriaV"
         # Attached RIB Box
         [linkMe]
         # Geometry Data
      AttributeEnd
      AttributeBegin
         Identifier "|Agrp|E|G|GShape"
         Surface "sqMateriaV"
         # Attached RIB Box
         [linkMe]
         # Geometry Data
      AttributeEnd
      AttributeBegin
         Identifier "|Agrp|E|H|I|IShape"
         Surface "sqMateriaV"
         # Attached RIB Box
         [linkMe]
         # Geometry Data
      AttributeEnd
      AttributeBegin
         Identifier "|Agrp|B|D|DcoordsShape"
         # Attached RIB Box
         [linkMe]
         CoordinateSystem "DcoordsShape"
      AttributeEnd
   WorldEnd
FrameEnd
```

The sketch shows a simple scene. The Maya scene graph is effectively “flattened” — each nodes place in the hierarchy can be determined by its fully-qualified path name (the indentation has been added to make the data more human-readable). Some nodes receive special handling from MtoR — light sources like “keyTopShape” are always written by MtoR at the top of the shallow scene graph, so that they are globally available to all subsequent geometries; shadow cameras like “shdKeyTop-Shape” are normally hidden, and do not appear at all. Coordinate systems like “DcoordsShape”

(whose names must be globally unique) are written at the end of the RIB stream, after all geometric nodes.

Three RIB boxes provide entry points into the RIB stream for `movie_init`: two boxes for the header, and a RIB box which we attach to *every* geometric node, called "linkMe." The contents of these boxes, for the Maya user, are very simple, just the name of a single Tcl function:

RIB Box	Contents
Frame	[movie_init]
World	[initIllumStates]
LinkMe	[linkMe]

Each different `movie_init` function could tune itself to the current context (shadowing, node settings, etc). Any values returned by those functions would be inserted as text directly into the RIB stream.

Finally, the user had to have a way to pass information to Maya that `movie_init` could understand. We just used attributes of Maya nodes. In a few cases, we used attributes directly connected to the nodes in question, but we also used a global node we called *shotinfo*. At the start of each frame, the `[movie_init]` proc would execute, and search for the Maya text attribute "shotinfo.tcl" — if it was found, the contents were evaluated as Tcl commands. A Maya-based text editor window was provided to make the creation of these commands easier.

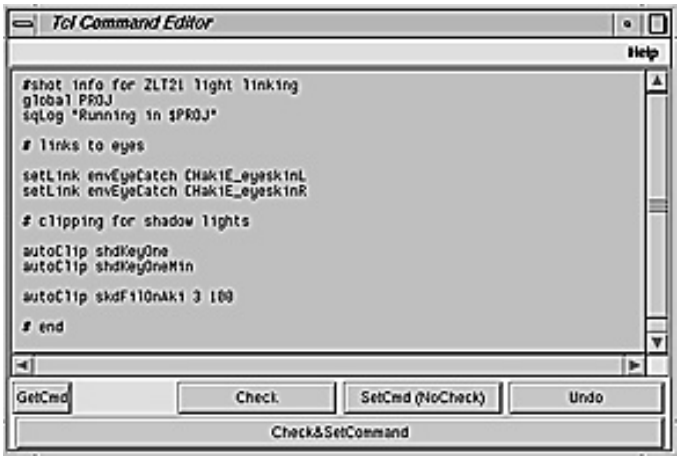


Figure 6.3: Maya editor for "shotinfo.tcl" attributes.

The `[movie_init]` function provided a variety of other services that werent available in RAT3, such as statistics control, shadow map clipping planes, midpoint shadow backplanes, configuration management and logging, and it provided a number of vital "backdoor" entry points for shot debugging.

This may not seem like a typical usage for Maya, since it wasnt GUI-based or strongly connected to Maya node networks, it worked well for our users. The users didnt really have to learn Tcl — while the full Tcl language was available via *shotinfo*, most users just learned a few important commands and used those exclusively.

Light Linker

When we began using MtoR with Maya 1.5, Mayas light linker was poor. Lights were linked, not according to different geometric nodes, but to shading sets — so you could link a light, say, to all copper objects — but not to a specific teapot.

As a workaround, a linker was added to `movie_init`. Each lightsource shader contained a Tcl proc called `[addL]` that would register the light with a Tcl list as each light was added to

the RIB stream; then those lights could be linked or unlinked by [linkMe] according to flags set by commands in shotinfo.tcl. The proc [initIllumStates] would initialize the On/Off RiIlluminate state for all lights, and then the [linkMe] proc, attached to every geometric node, emitted all the appropriate per-node RiIlluminate commands.

With Maya 2.0, a much better light linker was provided in Maya, and the movie_init linker was retired — for about three days. Lighting artists found it easier to see all their links together in a text window, and it was easier to copy link setups from one shot to another by using the movie_init light linker, so it was retained and was used on most shots in *Final Fantasy.*

Callbacks

The [linkMe] proc also provided an alternative means for adding RIB to a scene. MtoR permits only one RIB box per node, and the RIB boxes are only associated with shape (leaf) nodes. A system of callback procs was provided — procs that returned RIB commands could be added to any point in the scene heirarchy, and their output would be tagged onto the normal [linkMe] output.

This provided a good deal of flexibility for final rendering, since we could alter the RiMatte and other states of objects on a per-layer basis simply by altering the Tcl code for the scene, rather than rebuilding and relinking the individual RIB boxes.

Reference Handling

movie_init initially gained a foothold throughout the production by providing a workaround to a Glimpse problem: how to handle the names of objects and coordinate systems in Glimpse, when Maya may alter those names based upon file-referencing?

For example, a modeler may assign the shader coordinate system for an object to be "blackBoaMasterShape." This renders perfectly on his desktop. When the model is exported, and then another artist reads that model as a reference, the names of the nodes will change — Maya will add a prefix to show that they are references. So "blackBoaMasterShape" may have invisibly changed to "SPboa_blackBoaMasterShape."

A small movie_init function, [wPrefix] (included in listings at the end of these notes), solved this problem. Every shader was attached to some object, that that name was available as a Tcl global variable ($OBJNAME). So we created a simple rule: *no underscores in coordsys or reference prefix names*, and then [wPrefix] could examine $OBJNAME, identify any potential prefixes, add them (since in most cases the model and its coordinate system would be contained in the same referenced file), and return the possibly-altered name.[1]

In place of "blackBoaMasterShape" the artist would specify "[wPrefix blackBoaMasterShape]" (or have this proc inserted automatically during the model-delivery process).

Configuration Management

Because many models and files were being manipulated by many people, configuration management was crucial. In the early days, we were surprised by unexpected model changes — as an example, occasionally a character model would change after their shadow maps had been calculated, but before their RGB final images had been rendered. If the shape of the model had altered, the shadows would no longer line up, creating dark blotches that were quite difficult to explain. Such changes were at first very hard to track.

movie_init provided us with a tracking mechanism. Since it could make Mel calls during RIB generation, one of movie_inits jobs was to identify the correct versions of all input Maya files, and write their Ids into the render log and as RIB file comments.

Mdump & QMD

Glimpse encodes its data as a gigantic binary string. This string contains all the data Glimpse needs to convert a scene into RIB, including shader parameters, data types, and so forth. The downside is that it made the information unreadable unless the scene was actually loaded into a Maya session — and then only approachable through Mel or through the Glimpse UI.

[1] A similar proc is provided in RAT4: [objPrefix] uses different parameters but performs some of the same functions.

We wrote a command-line utility called *mdump* that would read a scene in Maya batch mode and dump out the Glimpse partition (along with Maya data and data gleaned from "shotinfo.tcl") as structured text. Mdumps became a useful diagnostic throughout the production, for use not only with programs like *vi* and *grep* but also *diff*, so that unexpected model changes were easier to track down.

Since shaders used on Final Fantasy tended to be large and monolithic, with potentially hundreds of parameters, *mdump* files could get to be huge. For most cases, a lot of those parameters were unused. A "smart mode" was added that would report only on shader parameters that had been changed from their default values. To do this, mdump had to make multiple passes. First, it would identify all shaders; then it would launch the *sloinfo* program to extract the expected default values from each shader; finally it would compare the listed values with the values present in Glimpse, and only print the values that differed.

Mdump would print out the shaders and info of an entire scene and all its referenced data. We often found that we only wanted the data defined there in a specific file — not its references. *Qmd*, for "quick mdump" could give us such results in just a few seconds. *Qmds* method was very simple: the Maya ASCII scene file would simply be stripped in *sed* of all nodes except its local Glimpse partition, and then the result read into *mdump* with "smart mode" enabled. This was particularly useful in lighting.

Caches — Static & Animated

RIB representations of characters and sets can be very large. Mayas normal mode of operation is to have the full model resident in memory and to launch Maya render jobs directly from that model. Since we were using a separate renderer, we could afford to cut corners and use RiArchive in tandem with simpler Maya models.

Creating a cache for use with RiArchive is simple for static models like sets.

Create two versions of the model — a "beauty" version, and a simple version for use as a proxy.

1. Make a scene containing the beauty model and render once to make a RIB file.
2. Edit-out the camera and display data (using a text editor or simple script in perl, Tcl, etc), leaving only the Declarations and `AttributeBegin`/`AttributeEnd` blocks of the nodes you want to preserve. This file is your RIB cache, say "airport.DHcache.rib"
3. Load the proxy model, select all geometric nodes, and set the Maya "primaryVisibility" attribute to "off." Add a locator to the model, with "primaryVisibility" set to "on" — attach a RIB box to this locator (or use a `movie_init` callback) that contains `ReadArchive "airport.DHcache.rib"` and save the proxy model.

The proxy model should be very simple — just enough detail to work on the shot. The full detail will only be present in the final render. This not only speeds up your RIB generation, it can greatly accelerate the Maya refresh rate.

Refinements of this simple technique can include making twin proxies for models that create different RIB representations for normal and shadow rendering, using `RiProcDelayedReadArchive`, attaching cached geometry to moving nodes (the ordering of the shaders and `ConcatTransform`'s must be shuffled to support this!), or building animated caches. At various times, we applied all of those refinements for use on *Final Fantasy.*

Two complications can arise from this technique: first, scenes that call the proxy model must be sure that they define all the necessary resource paths needed by the beauty model. Second, if light sources are present in the cache model, they must have their lightsource ID numbers edited.

Normally, each lightsource is assigned a sequential light number. If a light ID number is repeated, subsequent `RiIlluminate` commands will be directed at the most-recently-defined lightsource — even if the *AttributeBegin/End* scope containing that extra light has already been popped.

Models built using "magicLight" and similar shaders have light sources directly attached to the geometry, and dont need to worry about `RiIlluminate`. To avoid any potential conflicts with light IDs that might be declared in a scene file that uses a RIB cache, we would simply edit the values of all such lights within the RIB cache files to be 65565.

Renderfarm

Our render farm on *Final Fantasy* was heterogeneous — we ran Maya and MtoR on SGI Irix machines, while most rendering occurred on arrays of linux machines. The linux farm served mostly for prman rendering, but also served as a testbed for other software Square was developing apart from the *Final Fantasy* movie project. This division between RIB generation and rendering encouraged us to use subframe RIB generation, since linux parallelism was easier to implement.

Initially, we used LSF to administer the farm, but ultimately found it expedient to write our own job-control software that would be easier (and cheaper) to tune according to the varied needs of the farm. We also made it compatible with Alfred, so that individual users could user farm-administered rendering machines for *netrender* jobs.

6.2 Characters

Human characters are the most striking obvious feature of *Final Fantasy*, and they were by far the most laborious to build, to light, and to render.

6.2.1 Hair

Hair in *Final Fantasy* was rendered using `RiCurves` primitives. This in itself is not a new technique, but our implementation was unique. Hair, unfortunately, remains a very brute-force-intensive part of character animation and rendering.

An early design decision was whether to use Maya curves directly, to use a MtoR ribgen appearance (a plugin written in C), or to delay hair generation until render time and use an RenderMan geometry DSO. We opted for using MtoR ribgen appearances, because:

- The hair generation and animation itself occurred within Maya, using proprietary plugins. The data was already resident in memory for use by the MtoR plugin — this data and relationships would need to be duplicated for a RenderMan DSO.
- By using a ribgen appearance, we could control large numbers of hairs through a small number of Maya nodes — a node for each hair in Maya would have been unworkable (and MtoR would have attached one shader per hair!)
- By using a ribgen appearance, we could have more-exacting control over the exact values passed to the RIB stream than if we had used MtoRs built-in curve-to-`RiCurves` conversion.
- By defining hair in RIB, it could be potentially cached for use with `RiReadArchive`. On some motion-blurred frames, Akis hair could take up to 30 minutes in RIB-conversion calculation time, so RIB archiving posed a large potential savings in compute time.

Different characters had differing hair needs. Aki had the longest hair in the film, and up to 90 percent of the RIB representation for Aki was contained in the storage of her hairs. A fully-tressed Aki typically had a RIB representation that was about ten times that of Ryan, who had a very short patch of fuzzy hair just at the top of his head. Aki was our worst-case character, with respect to hair — she was about twice as complex as any other character in the film, and almost all of that complexity came from her hair.

Aki was in almost every shot, and her hair made her expensive to render. During the year 2000, almost 30 percent of all render cycles were used rendering just Aki . Considering that upwards of 80 percent of Akis render time could be spent rendering just her hair, this gave us a lot of incentive to render it as efficiently as possible.

Shading rate had little impact. Generally the strands were smaller than the shading rate already, so changing the shading rate from 0.5 to 1 or even to 10 rarely changed the render time by more than 3-5% (and because these strands were so fine, we often had to use very high pixel sampling rates to avoid aliasing — "turn it up to eleven" eventually gave way on some shots to sampling rates of 17 or 25).

The strands were long, thin, and thus were rarely fully obscured by one another or by any single primitive in the head. This means that RenderMan occlusion culling was generally ineffective in reducing the number of strands before they were diced and shaded, so shading efficiency became the single most-important key to getting Aki (in particular) to render more quickly. Since most likely every strand would be shaded, and the shading samples would be small, and worse yet the shading grids would probably be smaller than four samples per grid, it was important to keep shading complexity at a minimum.

One way we made Aki more efficient was to have different versions of her hair, which were chosen based on the demands of each shot. Aki had three versions of her hair: tagged Low, Medium, and High. Most shots used the high-detail version, but it was possible for the lighting artist to use a simpler version while preparing a shot, decreasing the render times for their test renders. The lower-detail versions had fewer (but thicker) hairs.

Animated RIB caches were prepared for some shots — especially if we anticipated a lot of re-rendering. Typically we would use the caches only while doing lighting setup — by avoiding motion-blur we would speed-up the lighting process and halve the size of the required RIB files. But this meant that the archives were inappropriate for the final, motion-blurred renderings. At the end of the pipeline, those archives would be swapped-out, just before final render, and the last pass would be done without the benefit of the archive files.

Characters with shorter hair could sometimes have their hair entirely replaced with a single, static RIB element. For the male characters like Gray or Neil, only the portion of hair near the nape of the neck actually animated with respect to the character deformations — whenever those areas were hidden, we could just use a locked version of the hair via `RiReadArchive` and not have to worry about it (this archive could also be used for motion-blurred renders). Similarly, Janes pulled-back hairstyle was essentially rigid everywhere save the ponytail.

An unusual feature of our hair genrib node was that it could generate different versions of the hair for the RGB render and for shadow passes. For shadow passes, we would usually set a shadow simplification value — the number of hairs in shadow passes would be reduced by this factor, while the width of the shadow strands would be likewise increased. This often provided a shadow that was adequate for use in the scene, with fewer high-frequency parts, and that rendered much more quickly than a shadow map containing all of the hairs. Users could control the level of shadow simplification via a single Maya attribute.

In the case of RIB cached hair, this meant that we needed two archive files —one cache for normal hair, and one cache for shadow hair (in some shots, there were even multiple versions of the shadow hair used, with different simplification values). Fortunately, shadow hair tended to consume far less disk space than the regular hair. Furthermore, since it rendered quickly, the lighting artists would often use the shadow archives in place of the highly-detailed archives while they did test renders.

The largest single speedup we could apply to hair was in careful control of shadow parameters when lighting. We often found that shots could be sped up by 30%, 60%, even 300%, simply by altering the shadow settings, or sometimes unlinking shadowed lights and ignoring them or replacing them with shadowless lights — without compromising the look of the shot. Well return to this subject later, when discussing light sources.

6.2.2 Hair Shading

The shader for hair was a simple one. This was an important feature! Shading efficiency was crucial. The method used was a close relative of the shading method used in the 1992 Siggraph paper "A Simple Method for Extracting the Natural Beauty of Hair," by Anjyo, Usami, and Kurihara. As noted in that paper, the algorithm performs better on dark hair than on light hair, and our gray-bearded character, Sid, was particularly difficult to shade well.

Color selection was done through a range of colors that was initially a painted gradient. The texture artists could paint whatever colors they liked into this map, and then randomized parameters in the hair RIBs could be used as indices into this map. Unfortunately, this method proved to be very inefficient. But when the gradient maps were replaced by calls to the shader language `color spline()` function instead, using the same colors, render times improved by about 30%.

6.2.3 Skin

Convincing skin remains a difficult challenge for computer graphics in general. CG models and surface-shading techniques focus almost entirely on thin, infinitely-thin surfaces. While this is appropriate for metal or painted objects, objects made of materials that interact with light in more complex ways are difficult to model and shade. Living tissues, which are largely liquid and which exhibit large variation in appearance across different parts of the same individual, are generally not well-served by current CG paradigms.

Some Characteristics of Real Skin

In nature, skin has wide variety. A single individual may have fine, smooth skin on their cheeks and deeply pored rough oily skin on their nose; they may have soft palms and callused fingers; they have gray-pale legs and tanned arms. Different individuals may have widely varying tones due to differences in age, ethnicity, fatigue, and cleanliness; they may also wear a variety of deliberate covers or alter their appearance through makeup. Real skin, being composed of cells that are mostly liquid, is partly transparent and many important appearance effects come from internal scattering. Different areas of skin undergo expansion and compression from movement, speech or mood; changes in color due to pressure or condition of the subcutaneous layers can be seen not just in the pursed lip or the squeezed arm, but also in expressions such as a blush or a murderous pale. Skin may be altered artificially by scarring, tattoos, and surgery. Skin across the body is covered with hairs with further variety of color, density, and direction. Most men shave their faces, and their appearances change over the course of a day. A character may be covered by layers of perspiration or dirt. All of these effects may appear differently at different scales, and whats crucial in a closeup may be either distracting or just lost in a long shot.

Most of these skin characteristics have been studied and attempts have been made to directly model them with varying degrees of precision (and success). The problems inherent in these approaches usually stem from model complexity and from inadequacies of the algorithms used.

Alternatively, attempts have been made to measure and copy skin characteristics directly via photos or goniometry, ignoring the underlying optical mechanisms and just attempting to copy the appearance from life. Problems inherent in this approach include inflexibility, difficulties in modeling variations across a single individual, and the challenge of finding a match between the desired character and the source model.

For *Final Fantasy,* our characters were fictional creations, and couldnt be photographed in advance. The people charged with creating each characters appearance were not programmers or scientists, but skilled painters. Their reference materials were not scientific papers on energy distribution, but the pages of *Glamour* and *GQ* magazines. Because of the history of the production, the basic shading model had to be able to work not only with RenderMan, but also with Maya. As a result we opted for schemes that emulated some important skin characteristics, but were handled via a

few easy-to-understand controls by texture artists who generally worked without intensive technical oversight. A single surface shader, *sqFlesh2*, was used for the skin of every human character in the film.

Side Shading

The skin shader makes extensive use of maps and "facing ratio" shading — that is, functions which alter their behavior based upon the relation between the surface normal and the view direction. Since the normal and view vectors are readily available in RenderMan shaders, we can make any sort of function based upon the cosine of the incident view angle, which will be `(N.L)` when the vectors are normalized. This is a very standard shading idiom.

For the skin shaders used in Final Fantasy, multiple relationships were defined, to create some of the illusion of different dermal layers. Two diffuse-reflection components were created, and the shading rolled over from one to another based on an exponent — that is,

```
mix(colorA,colorB,pow((N.L),sideShadeExp)
```

The specular components were also mixed, but with different exponents. The specular functions contained terms for base specularity, for response to mapped reflections, and different specular functions were used and mixed to create the illusion of separate specular reflections from the skin itself and from water and oil on the skin surface.

Bump maps (separate maps for diffuse and specular components) were likewise mixed according to the facing of the surface, so that the texture painter could control the rolloff of smoothness of the skin along the contour edges.

While a more exacting mathematical method may have been available, this method permitted the texture artist, who was already a skilled painter with an eye for important details, to reach an approved look without having to learn anything in depth about light transport, filtering, and optics before they could even begin to create characters. Once adept at using these shaders, the texture artists often used them successfully in other, sometimes unexpected ways — for example, the flesh shader was also used for teeth.

Far, Medium, Closeup

While a single suite of maps and settings could be used for a character, the skin, like the hair, had different versions for different kinds of shots. The balance of features, the strength of the bump mapping, and the highlight balances were all features that could potentially change based upon the distance to the camera.

At the extreme, such as the very tight eye closeup of General Hein in his escape pod, a special version of the model was used, more-detailed than for regular use and tuned to the specific shot (similarly, we used custom versions of Akis hair for her surgical operation and for her zero-gee encounter with Gray — we also used special versions of other body parts, such as the super-detailed shoes, complete with magnetic locks, used for a single closeup on Akis heels near the beginning of the film).

Subsurface Scattering & Wrap

While the basic coloring modulated its effects based upon the view direction, it didnt do anything unusual based on the incident `L` direction — just regular calls to `diffuse()` and to our slightly-customized Phong() and Blinn() specular functions (the characters made little use of the built-in `specular()` function, mostly for the sake of remaining compatible with Maya).

To emulate scattering of light underneath the skin, we added a second diffuse()-like function to provide a faint blood-colored layer that propagated beyond the range of the normal 180-degree

visible hemisphere. So for any given point on the surface, it may receive light not only from the lights immediately in front of it, but also from lights slightly behind it.

This model of subsurface scattering, based on the normal-to-light angle captured by `(L.N)`, means that the amount of scattering is highly dependant upon the curvature of the surface — on a face, the breadth of scattering on a broad curved area like a cheek will be much greater than on an area with a high degree of curvature, such as the ears or the nostrils. This is an incorrect result and for a time we considered adding yet another texture map to the characters to correct for these effects. In practice, however, it was decided that the negative effects were only apparent on rare occasions (such as severe toplight), and in those instances the lighting artist could simply modify the shader scattering values for that single shot.

Warping light to "bend" past the horizon has also been used for "wrap" lighting, in which the lightsource is distorted to deliver illumination analogous to that of a disk-shaped source. Although lights and some surfaces were developed to handle this sort of lighting, they were ultimately never used in *Final Fantasy* due to schedule and consistency constraints.

As with any type of shading that involves `illuminance` angles of greater than 180 degrees, a number of shadowing problems cropped up — we found a few solutions, which will be described later.

For small thin details like ears and the short webbing between fingers, we also added a paint-controlled backlit layer. For shots that used it, the backlighting was particularly striking on the hands.

Oren-Nayar Diffuse Function

For a short while, we experimented with Oren-Nayar style diffuse shading for the skin. The Oren-Nayar equations provide a "flatter" appearance than the regular `diffuse()`, and the effect is in some ways similar to real skin — especially youthful skin that contains more fluid in the individual cells than older skin. Older skin renews itself more slowly and tends to have a drier, less-luminous appearance due to the lack of collagen and new cells.

Ultimately, Oren-Nayar was removed partially because it worked *too* well — in tests made with a simple replacement of the `diffuse()` calls with an Oren-Nayar equivalent on Aki, a number of comments were received that she looked too young, and that no one would accept her character if she looked fifteen years old.

The method worked — but would have required additional artist time to rebalance Aki overall, so the idea was left unused. Since youthful appearance is also the goal of most day-to-day makeup, this technique may find increased application in the future.

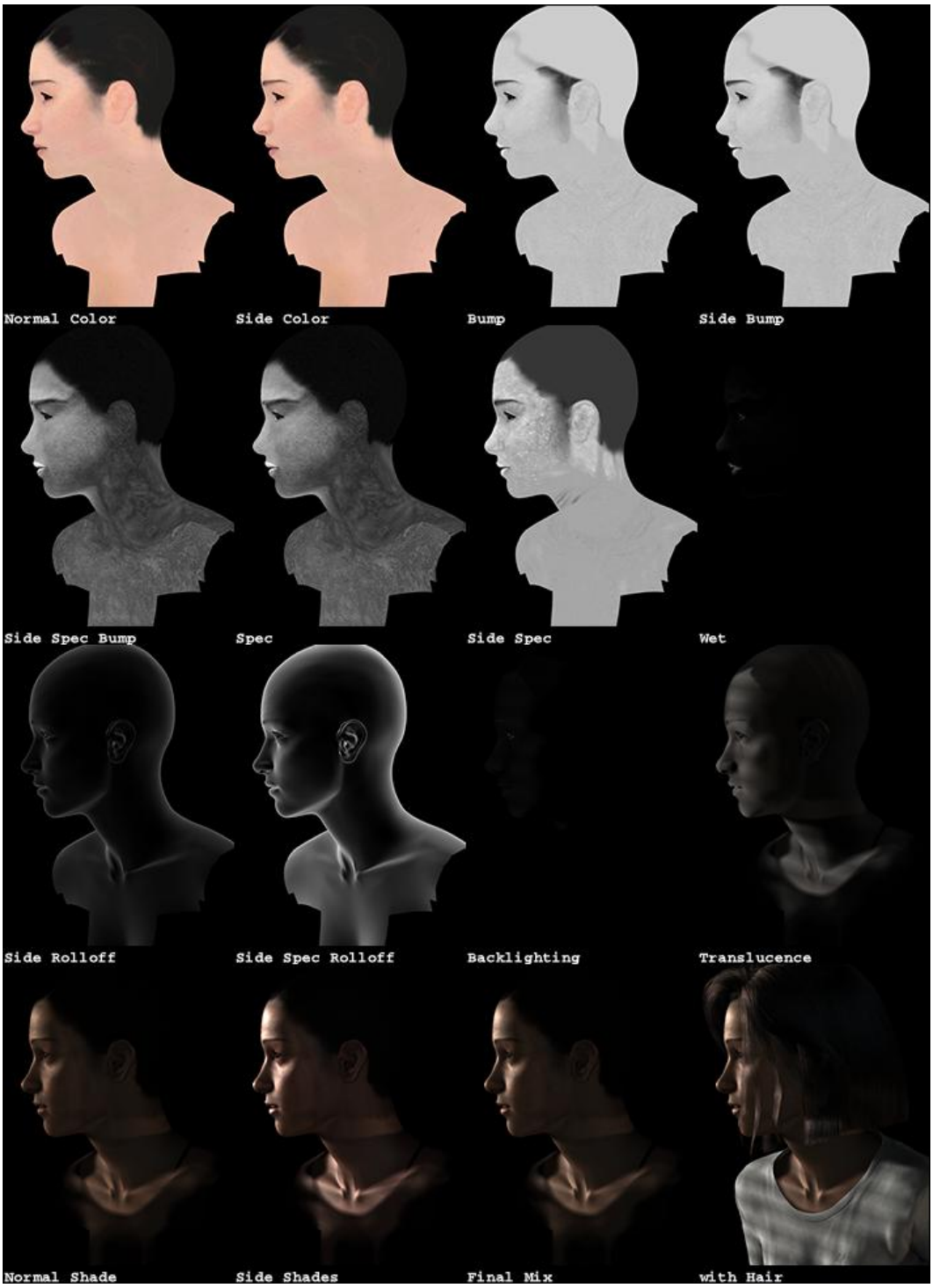


Figure 6.4: Visual components of Akis skin, editable by character painters. ©2001 FFFP. All Rights Reserved. Square Pictures, Inc.

Fuzz

Experiments were made using a faint "peachfuzz" element in the skin shading. Again, the additional setup time required, especially to add peachfuzz to characters that had already had one look approved, prevented its use in final imagery. We did, however, apply the same shading equations to cloth, as described below.

Figure 6.5: Aki with a uniform layer of "peachfuzz" to catch fine rimlights (e.g., the outline of her nose).

6.2.4 Clothing

Some of the clothing in *Final Fantasy* was rendered using simple plastic and metal shaders. It should come as no surprise to anyone that scenes that featured the Deep Eyes armor, which was built of rigid metal parts involving no cloth, skin, or hair, were scenes that could be set up and rendered most quickly.

Akis spacesuit was built with displacement and bump shaders to emulate a rubber weave, but the shading was very much like that of most common CG objects. Similarly, the lab aprons used by Dr. Sid and his assistants were designed with a very clean look in mind.

The unarmored uniforms of the soldiers and officers, however, and many of the clothes worn by incidental characters such as the Council and the people attacked on the rooftops of New York, required a different set of shaders, which eventually consolidated into a single large shader called "sqCloth2."

"SqCloth" began life as a "facing ratio" shader, much like (in fact derived from) "sqFlesh." It was quickly apparent when lighting scenes in darkened environments, however, that this shader was inadequate for soft fabrics like tee-shirts and wool uniforms, particularly because they wouldnt get backlit rims similar to those found on real fabrics.

Fuzz

The solution to this problem was to create a small library of "fuzz" or "vertical texture" functions, similar to those described in Goldmans 1997 Siggraph paper on "fake fur rendering." Adding a little

extra “fluff” to the fabric provided good backlighting for several scenes. But adding it presented us with a classic production problem.

By the time “fuzz” was added, many shots had already been completed without using fuzz. Some shots had been approved in lighting review, but had yet to render. Other shots in the same sequence needed fuzzy rim lighting. Since the approved shots had already been done without using any fluff, how could we add the fluff to the single shared shader, without altering shots that had already been approved and that might already be rendering? Would we have to request extra versions of all the characters from the character group? We could generate new versions of the characters with a new shader, but the modelers were reluctant to do so, and due to the architecture of MtoR and Maya, it was difficult to swap large numbers appearances while maintaining all of the customized parameters.

The workaround chosen was to use RenderMan light categories.

Each “fluffy” surface shader was provided with a string parameter called *ContreJourCategory.* By default, this string had the value “back.”

When shaded, only light sources with categories matching *ContreJourCategory* were used in the fuzz calculations. If there were no such lights in the scene (and none of the shots already approved would have this category defined), the fuzz didnt happen. If *ContreJourCategory* was empty (`""`), then all lights were part of the fluff calculation, so that future models could use the fuzz according to the discretion of the modeler or texture artist.

This use of categories meant that scenes prepared without regard to sqCloths fuzz parameters would simply render without using fuzz. Scenes that did use the fuzz with the category lights also benefited, because it meant that the backlights and rims could be separated from the primary scene lighting, so that the contre-jour effects could be tuned for the specific needs of each shot.

Spacesuit Displacements

As mentioned above, Akis spacesuit made heavy use of displacements. Some of the displacement maps were very detailed. Many other objects used displacements, and MtoR has a simple built-in rule for using displacements in shadow maps: *always use displacements in the shadow maps.*

This presented a problem because often the displacement was important for final appearance, but often not at all important for the shadows. We didnt always want the displacement for the shadow pass, or sometimes we wanted a different displacement. Displacements are potentially expensive, especially when characters like Aki were actually carrying light sources that were very close to the displaced surfaces. Sometimes displaced surfaces were completely obscured most of the time.

We could have two copies of the model, one for rendering and one for shadow mapping, but the models were already so heavy that it would have been inappropriately expensive for the user at their workstation to use two copies of a model like Aki.

We came up with a simple cheat to solve this problem, by using the prman 3.8 `attribute()` shading-language function.

By calling `is_shadow_pass()` (listing at the end of these notes), the displacement shader could quickly determine if it was being rendered as part of a shadow map. An extra toggle parameter was added to displacement shaders, providing users with a way to override this behavior; but this simple function allowed most shadow passes to be accelerated sometimes by as much as a factor of two.

6.3 Shaders

6.3.1 Design Principles and Idioms

MtoR comes with a number of readymade, very powerful shaders. When we began production using MtoR, the users were quick to pick up these shaders.

Unfortunately, the shaders dont come with source code, and that presented real problems later. While we generally used only a small number of different shaders, it was important that we have the

source code for all shaders, to alter them for special needs or to fix them when things went wrong — and sometimes they did.

While MtoR shaders were used in the early days, we quickly declared the packaged shaders off-limits, and eventually all old models with MtoR shaders needed to have those shaders replaced by similar home-brewed equivalents.

Parameter Names

Users were familiar with Maya and other systems like Power Animator. It may seem trivial, but renaming basic parameters such as “Kd” and “swidth” to their Maya equivalents “Diffuse” and “Filter” made the shaders far more approachable to users who had never used RenderMan before. We adopted (and enforced) Maya-standard names whenever available, barring a few rare exceptions like “color,” which is a reserved word in the RenderMan Shading Language.

Tcl & Comment Fields

Due to the pervasive presence of `movie_init`, nearly every Square shader contained a string parameter called “Tcl.” The shaders accepted the parameters, but never used it in the shading calculations. This parameter was the home to any procs that needed to be connected to that appearance but which didnt need to pass a value to the shader, such as the proc `[addL]`, which registered lightsource shaders with the `movie_init` light linker. Typically the value actually passed by “Tcl” was an empty string (`[addL]` returned `""`), but it proved invaluable in some circumstances, since it also provided us with a means for inserting debugging information into RIB shaders.

Light source shaders also had an additional hidden Tcl parameter called “NodeName,” which was used for examining and debugging light intensities on specific surfaces. Users never saw this parameter, but it was available to the lightsource and surface shaders via the `attribute()` function.

Since so many surfaces and lights used the same shaders, we also provided another unused string called “Comment” for many shaders, so that users could have a place to make user annotations in Glimpse appearances.

6.3.2 Lights

While the surface shading tended to focus on a small number of general-purpose shaders, the lighting could afford to be more flexible and a wide number of specialty light shaders were developed during the course of *Final Fantasy* production.

As with most productions the most common lights were spotlights and the most common spotlight shader was called *sqLight.* Many variations of *sqLight* were used, and all of them are direct descendants of the well-known *uberLight* shader.

Figure 6.6: Lights were often "split" to provide separate specular and diffuse properties. Since hair is highly specular, altering the proportion of specular light creates the illusion of altering Dr. Sids age.

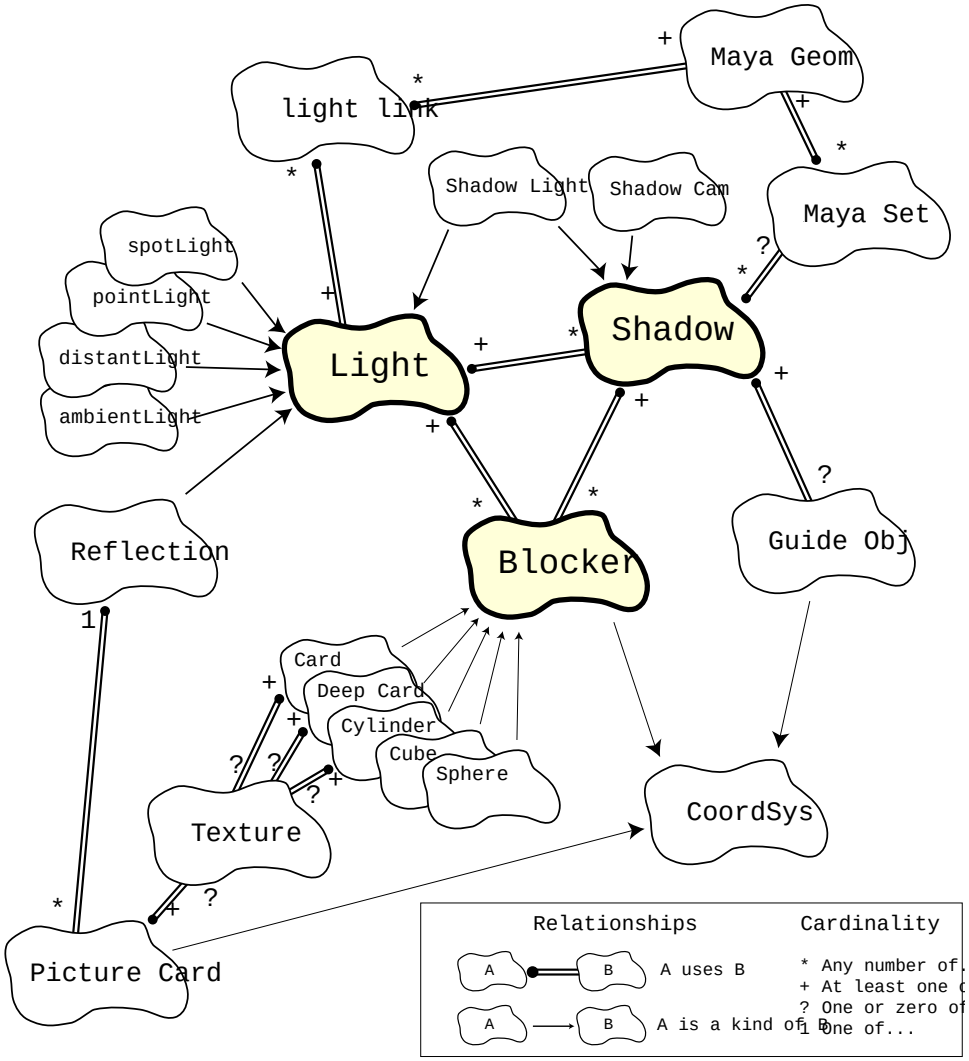


Figure 6.7: Booch-style diagram of object classes used for lighting. Maya pushbutton scripts were provided to create/edit objects and relationships based on these classes.

Blockers

The notes for *uberLight* mentioned that likely extensions included adding multiple blockers and support for multiple shadow maps, and *sqLight* contained both. Most lights needed at least two shadow map slots available, and most blocker setups often needed multiple blockers too. By the end of the film our most-common "base" lamp used two of each, but some varieties needed as many as fifty shadow maps in the same light, and a dozen blockers were not uncommon.[2]

The MtoR default display for a named mtorCoordinateSystem was a card, perfect for use to preview blocker locations in Maya.

[2] An obvious way to do this would be to use shading language arrays for all data types, and wouldn't map Maya attributes onto array elements. Therefore, an array-based approach was unavailable. Instead, we used the unix text preprocessor `m4` — our shader *Make* rules mapped .m4 files as the source for .sl files, which were further compiled into .slo shaders for *PRMan*. Compile-time options allowed us to quickly create many different versions of sqLight containing exactly the features and numbers of shadows and blockers that were desired for a particular application, all from a single core .m4 source file.

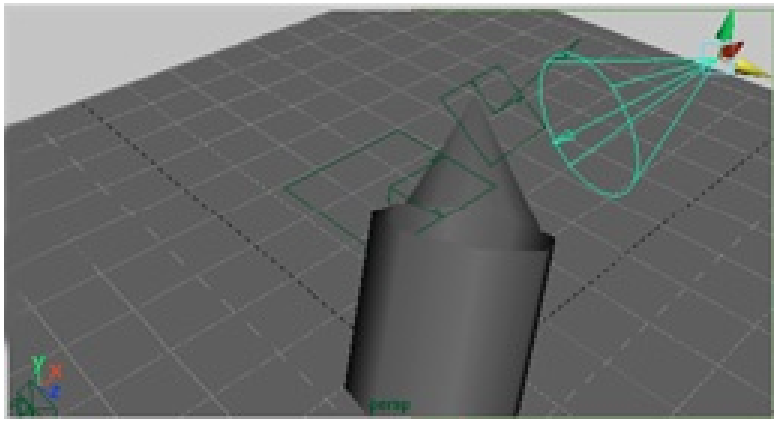


Figure 6.8: Using mtorCoordSys objects as blockers. Maya preview showing a light source and two coordinate systems (rectangles)

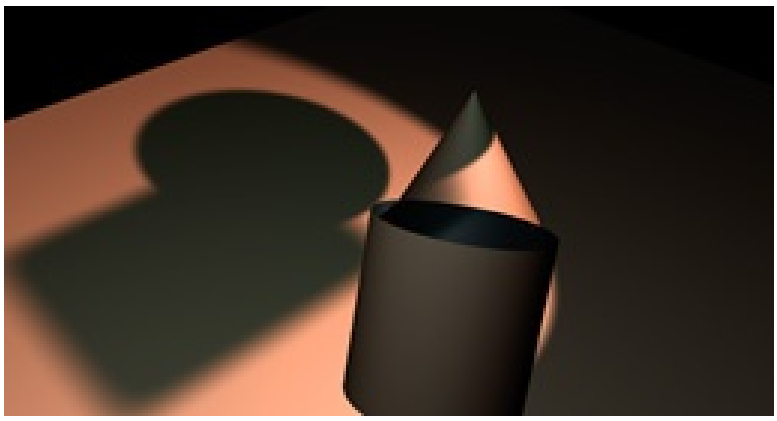


Figure 6.9: Rendered view showing round and square blockers.

A blocker-made shadow really defines a volume — the volume is projected from the lightsource through the blocker and into the scene, just like a normal shadow. We let the users select whether they wanted perspective projection or parallel projection, and we also permitted them to apply texture maps to the blocker shape, so if the square or circular shapes were inadequate, it was easy to just quickly paint any shape you like and apply it as the blocker outline. Being able to "fly" cookies around this way proved to be far more useful than simply applying them to the light cone (though we reserved the special coordinateSystem name, "shader," for blocker and gobo effects that we did want to be applied directly to the light cone).

Blocker textures could be colored, too — we used the shading language `textureinfo()` command to examine the texture, and alter the blocker behavior automatically, based on the number of channels present in the texture:

Number of Texture Channels	**Behavior**
One	A mask channel, so alter the blocker shape
Three	A slide — color the blocker
Four	A slide with a shape mask

MtoRs coordinate systems could also be displayed as cubes, cylinders, or spheres, so we let the

user choose those solid shapes too — not to be projected through space, but as explicit volumes that could be scaled, moved, and rotated freely as objects in Maya without regard to the light direction. We also gave users control over the *handedness* of the blockers, so that a volumetric blocker could define not only a volume within which *no* light appeared, but could also be used to define a volume "window" so that light *only* appeared inside the blocker volume.

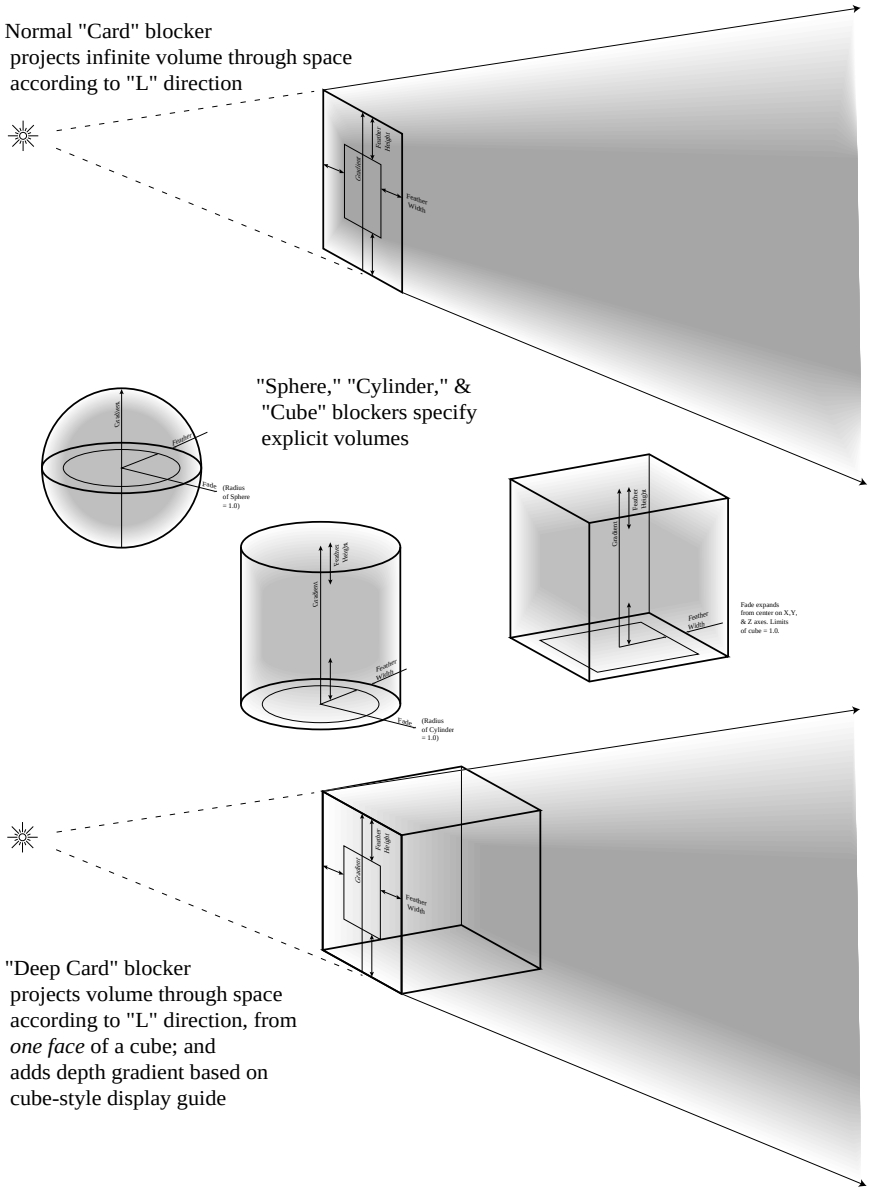


Figure 6.10: Five kinds of blockers.

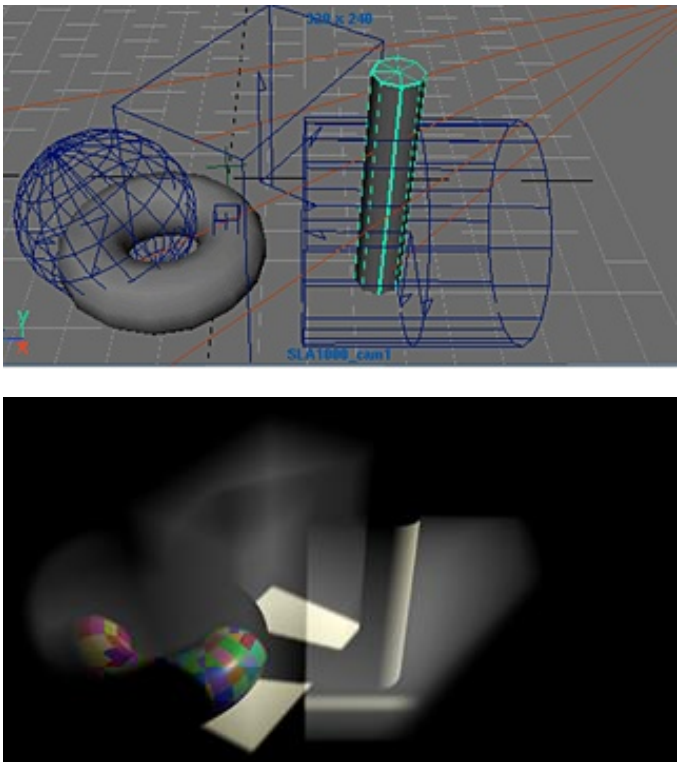


Figure 6.11: Inverted solid blockers used to constrain the illuminated area. Only surfaces within the solid blockers are illuminated (volume fog added to make the blockers easier to see).

Using arbitrary volumes for light control (including the existing *uberLight* "shaft" capabilities) was a very powerful tool for lighting the many complex sets that were used in Final Fantasy. The volumes all had edge-feathering controls similar to the feathering of planar blockers in *uberLight* — we also added a vertical (in blocker coordinates) gradient control, so that blocker effects could be shaped even further.

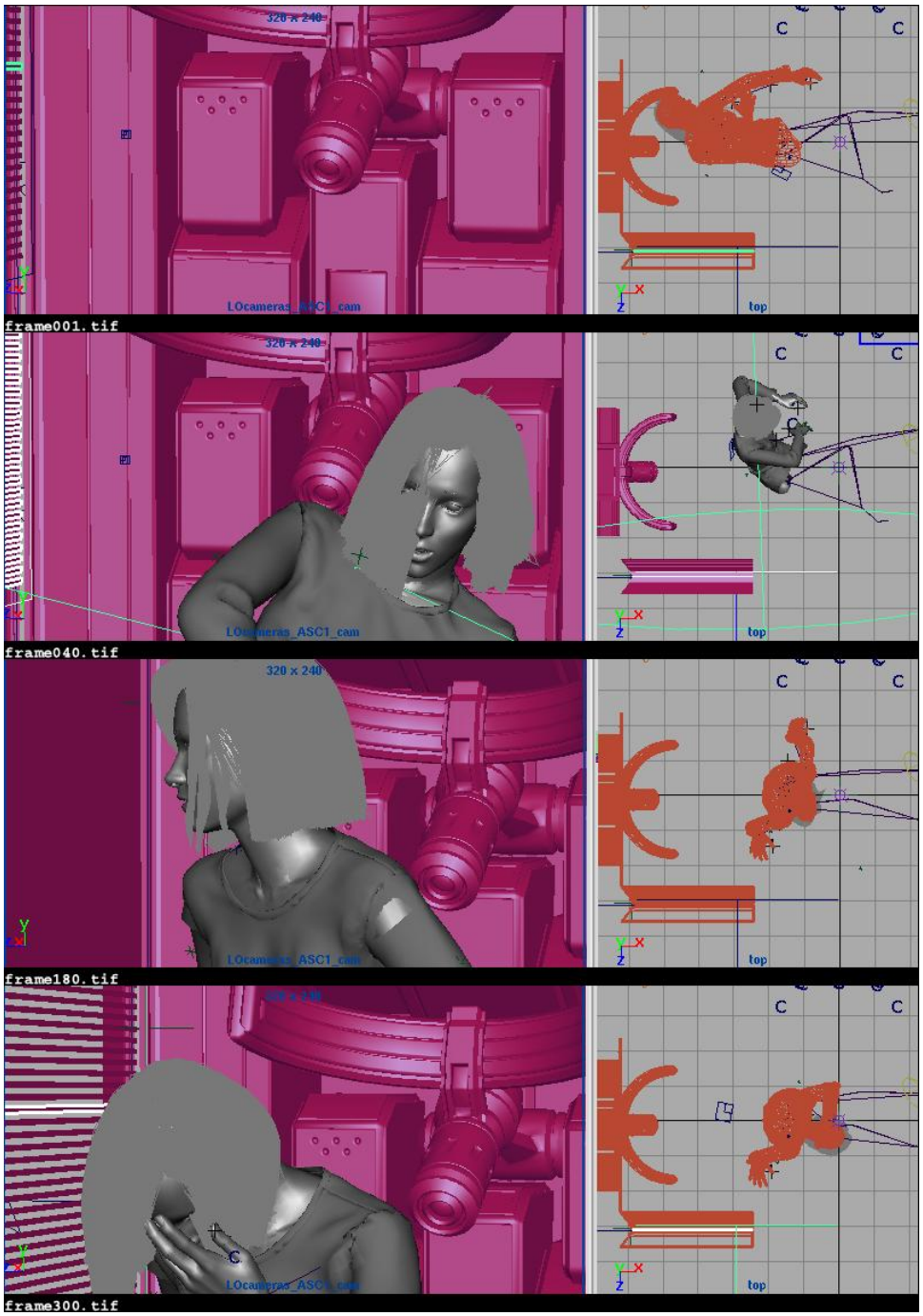


Figure 6.12: Blockers used in a simple scene. The window was only partially modeled, so an inverted card blocker was used to create a shadow for its overall off-screen shape. The blocker was textured to create the outline of the window blinds, and the texture pre-blurred so that execution time was much faster than if a softened shadow had been used. The same coordinate system used for the window blocker was also used as a "softbox card" to create reflections in Akis eyes. Although the light was actually coming from the front, it was requested that it appear to be coming from behind, so a cube blocker was added to mask the area where Aki is lying at the beginning of the shot. Finally, two more blockers were used on the background to constrain and shape the light coming from a hologram.

Figure 6.13: Final image — haze added in composite.

We further extended the idea of the card blocker by adding a *deep card* blocker. The deep card was represented as a cube in Maya, with its pivot point along one of the cube faces (rather than the center of the cube). The pivot face behaved like a regular card-style blocker, projecting its shadow into the scene, but the depth of the card was used to determine a depth-based gradient — the card effect would fade in along the distance displayed. This, combined with the vertical gradient and invertable handedness of the blocker volumes, meant that we could be very free to grade and shape the light (regardless of its apparent direction based on the shader language `L` vector).

In *uberLight*, blockers are used as a way to modulate the intensity of a light, as a replacement for shadowmaps. With the addition of gradients, blockers can be used as a more-general 3D function, whose output is a float. Typically, we use these functions as shadows and reduce the light intensity based on the blocker density. In sqLight, we decided that there were other attributes of the light that we might alter, too. For example, we might want to remove the specular component of a light for just one small area, or alter the effects of other blockers or shadows. Much in the same way that surface shaders can use maps to control just about any surface attribute, any light attribute that could be passed as a numeric shader parameter can be tweaked and noodled spatially by using a blocker.

Shadow parameters were the most usefully-tweaked in this way. We regularly used blockers to eliminate unwanted shadows, to blur (or unblur) shadows around contact areas, and to manipulate shadow densities as they passed through translucent media like hair and smoke. Altering a shadows density and blur via deep card blockers became a favorite way to get a cheap approximation of prman soft shadowing without the overhead and with less testing, since the deep cards cube display showed you the specific region of space where the effect would take place.

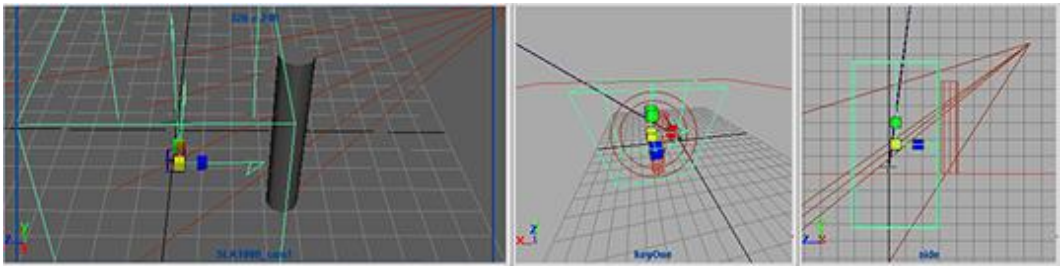


Figure 6.14: “Deep card” preview object, highlighted in green, as seen in scene camera, lamp, and orthographic views.

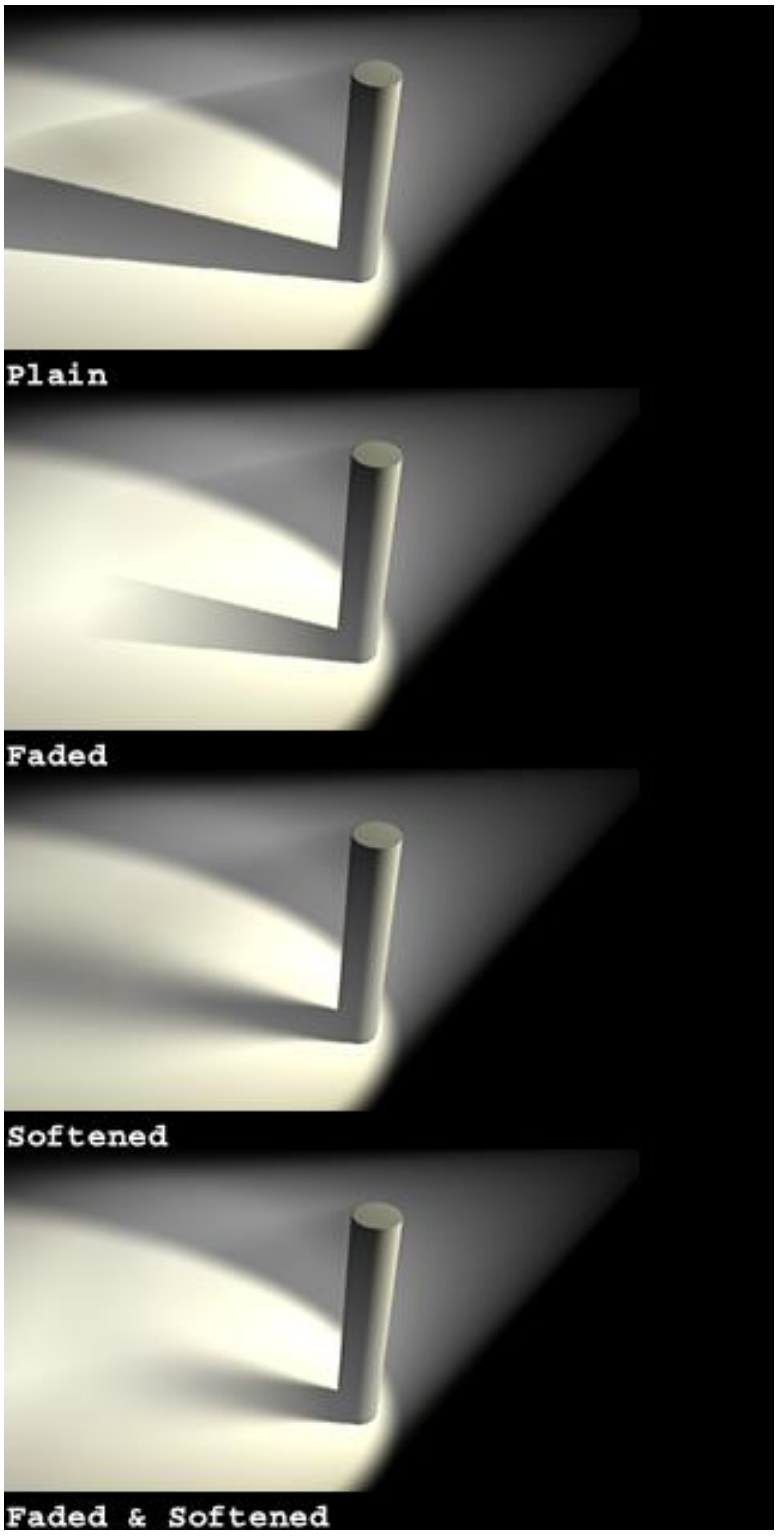


Figure 6.15: “Deep card” effects applied to a single shadow. In pixels to the far left, once the fade level reaches 100% the shader no longer makes “shadow()” calls, further improving render efficiency, especially for heavy amounts of blur.

Deep card blockers, like other blockers, could also be inverted, and used to fade in a shadow, rather than fade it out. Using a deepcard in this way, it was possible to grade-in hair shadows, so that they were translucent as they began but became fully opaque across the width of a characters head — in some ways, a very simplistic form of "deep shadow map."

Figure 6.16: "Deep card" used as a simple "deep shadow" function. The backlight shadow on General Heins hair increases in density toward the camera.

This use of blockers was further refined by adding controls to each individual map, so that we could tune the interplay between blocker effects and shadows. Sometimes, we wanted to fade or blur individual shadows, while leaving others unchanged — so "fadeable" and "softenable" flags were available on each individual shadow map, as user options.

Figure 6.17: Using "softenable" flags. Shadows from the window blind have softening applied and have been positioned (on General Hein only) using "offset," described in the text. Self-shadowing from the same light on General Hein has not been softened.

The basic internal architecture of *uberLight* had to change quite a bit to accommodate these ideas. An important change was to reverse the order of execution between blockers and shadows, so that blockers could be evaluated and accumulated *before* shadow maps were accessed — required since we want to accumulate blocker effects before we access the shadow maps, so that shadow blurs can be tweaked by blockers.

Putting the blockers first also means that in some cases, we can skip the shadow-map access altogether, which can have a big impact on accelerating the shot. If an object is completely obscured by blockers, or if shadows are being masked by blockers, then the shader can skip the shadow-mapping step entirely.

Blockers also found uses with other kinds of light sources — for example, inverted blockers were used to restrict ambient and point light sources, to emulate glow from the many hologram displays seen on sets throughout *Final Fantasy* (especially when characters reached inside the holo displays).

Meters

Like *uberLight*, *sqLight* provided a metering distance. We added a "named meter" feature, so that an mtorCoordSys node could be used at the meter point for multiple lights. The lights could be positioned arbitrarily, and their intensities and ratios defined, and then the lights could be further moved around without worrying about changes in intensity. The meter distance is the worldspace distance from the named coordinate system origin and the lights own "shader" origin. The user could choose either to use a named coordinate system or to just use a numeric distance (typically a number set by Maya as the distance to the lights point of interest), or both (they would be added together).

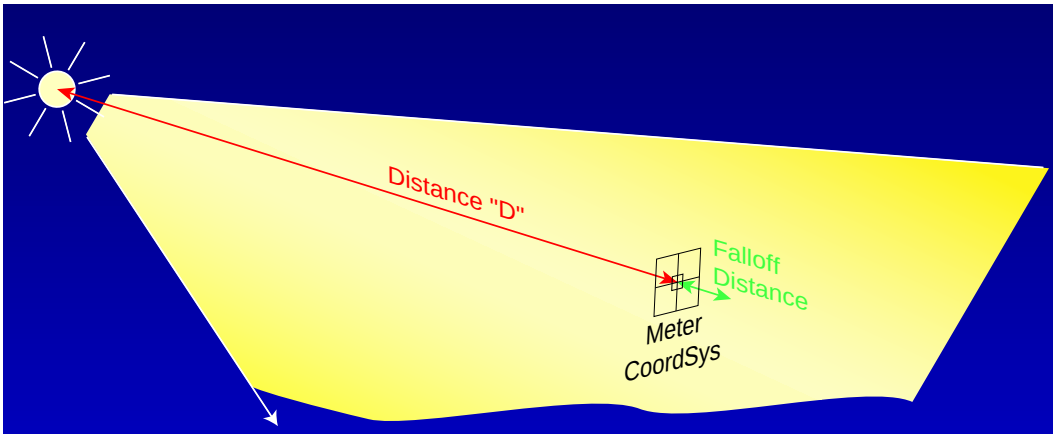


Figure 6.18: Using a named meter plus a meter distance.

Catchlights & Reflections

Without using a rayserver or BMRT, *PRMan* still accommodates a few different kinds of texture-mapped reflections, and we used all of them in *Final Fantasy.*

Planar reflections were often performed, using the now well-known trick of inverting a camera across the plane of the flat surface, converting its image to a texture map, and mapping it back into the "master" camera view by indexing it with "NDC" or "screen" coordinates. This method is described in the MtoR tutorials and has direct MtoR support.

We altered the method, because we had a large compositing department that could also do this sort of screen-to-screen mapping. For planar reflections, rather than using a light shader to map them back into the scene, we would just trap the reflection-cameras rendering, skip the texture-mapping step, and deliver the images to a compositor along with a flat mask element to show the compositor where we thought the reflection should go. This method was not only favored by the director, who could then alter the reflection independently from general lighting, but it also provided us with an easier means to combine reflections of multiple elements — if an element changed, its reflection could be re-rendered independently on the others, and just comped back into the final image.

Wraparound environment reflections were also used, and the refinements described in *Advanced RenderMan* were supported, particularly the use of giving Environment() calls with a known radius, rather than being projected infinitely. The "sqEnvironment" shader (similar to *ratEnvironment*) or its relatives was used on most outdoor scenes and many indoor ones.

We also used blocker-like cards as reflection objects. The RAT packaged "tvscreen" shader works the same way. We extended the "tvscreen" idea and called it "softbox." (a listing for a version of this shader, called "softboxes," is provided at the end of these notes).

A softbox light consists of a lightsource shader and some number of cards. The lightsource does a one-bounce raytrace based on the view and surface-normal vectors, and returns the color of the card, if the ray strikes the card. The cards can be textured (following the same rules as blockers, as described above); they can optionally obscure one another; they can be shaped round or square, just like blockers; and they can also be told to obey the inverse-square law, which provides a very realistic (specular) portion of the lighting from a large rectangular source.

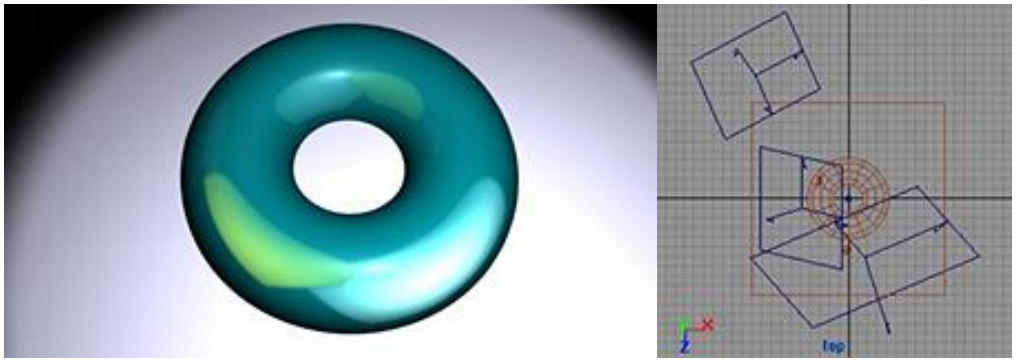


Figure 6.19: Softbox example using three cards. The cards obscure one another, and the intensity of the cards is modulated by the inverse-square law.

When a ray strikes no cards, it returns black. Eventually we realized that we would like to have the cards obscure not only each other, but also any overall environment map — so later versions of "softbox" also incorporated a regular `environment()` call, which would be accessed for rays that werent obscured by reflection cards.

Since reflection cards could be light or dark, they could be used not only as light sources, but also as a convenient way to mask-off the effects of environment maps.

As final refinements, we added support for shadow maps, so that softboxes could act in concert with other kinds of light source; and noise controls, so that the reflected rays could be slightly wobbled. This made surfaces look slightly less-perfect when we wanted reflections of less-than-mirrorlike surfaces like floors and walls. The result was similar to bump mapping, but unusual in that the reflection vector was being perturbed directly, not the surface normal — so the perturbation happened completely within the light source, without regard to the surface shader.

Softboxes were used a lot. Initially, they were introduced to provide an alternative form of specular light — we could declare a light source as "__nonspecular" and then replace the missing highlight with any shape reflection we liked. We also found softboxes useful as a way to create fake rimlighting on surfaces that didnt support special rimlighting code.

The heaviest use of softboxes was as eye catchlights. The characters in *Final Fantasy* are constantly interacting with projection holograms, computer monitors, and other large visible light sources such as windows. By using textured softbox cards linked just to the eyes, often with animating textures, it was possible to integrate these sources in a more-believable way, and to create a more lively look for character eyes throughout the movie.

To facilitate this approach further, the eyes (and hair) shaders could generate secondary-output masks for reflection compositing.

Materia control over reflections and blurs

A problem we encountered in dealing with reflection lightsources was that Maya, and the standard RAT shaders, provided a reflection texture as part of the *surface* shading. The model makers had been using this feature, adding cloud textures and similar generic reflections. This worked well for

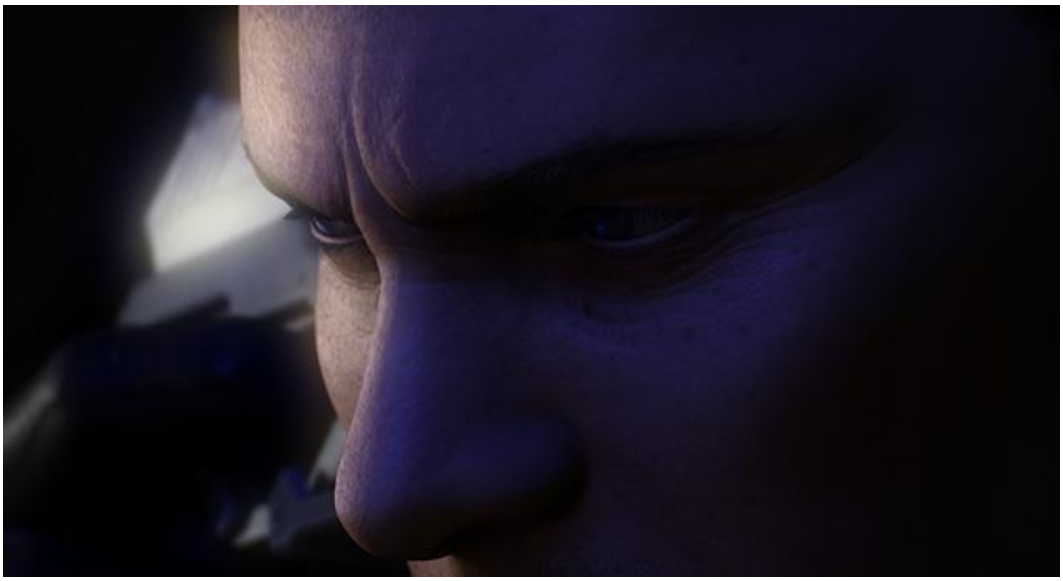


Figure 6.20: Eyecatches could be rendered separately. In this scene, the eye reflections were rendered several weeks after the main images, so that the design of the control panels could be revised without impacting other schedules.

their own purposes, and for getting model designs approved, but had terrible consequences when attempting to integrate those models into a lit scene. The reflections didnt match and were out of place. We needed to remove those textures and replace them with light-based reflections.

We still needed to respect the designers original intentions, however — some parts of a model may be shiny, and others not shiny. Some surfaces may have mirror-sharp reflections, while others are shiny but dulled or brushed, with blurred reflections.

The shader message-passing mechanism saved us again. We added a switch to reflective shaders like “sqEnvironment” and “softbox” — the switch was called “obeyMateria.”

When “obeyMateria” was turned on, the lights would search for float parameters that controlled reflection mapping. If found, and if non-zero, then the light would let the surface parameters influence the intensity and blur of the reflection map.

6.3.3 Shaftlight Preview

Many advanced features of the light source shaders were almost entirely unused because artists had a hard time pre-visualizing them. An example is the “shaft” capabilities of uberLight. Available in all versions of this shader, its hard to visualize in Maya. Any important tool we needed to develop was a shaftlight previewer made from Maya wireframes. Once this previewer was built, use of the shaftlight features greatly increased, and it became a standard light source, especially for set lighting — instead of a rare curiosity.

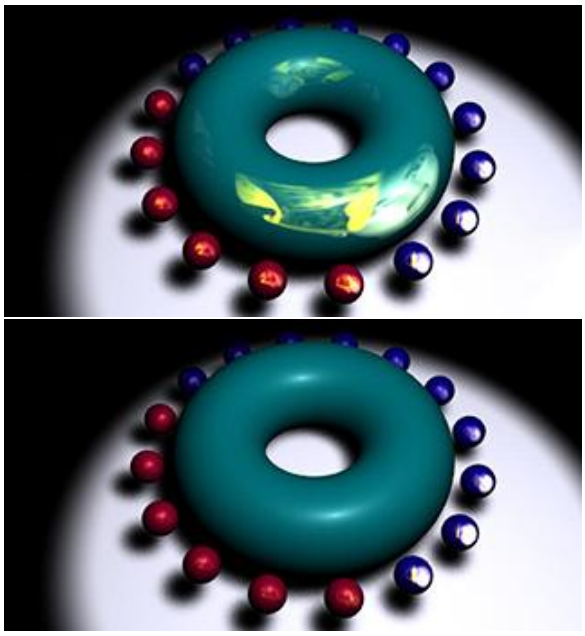


Figure 6.21: Using "obeyMateria." In the top view, lightsource reflections are applied uniformly to all surfaces. In the lower view, "obeyMateria" has been enabled, and reflection intensity and blur are controlled by surface parameters set by the model maker.

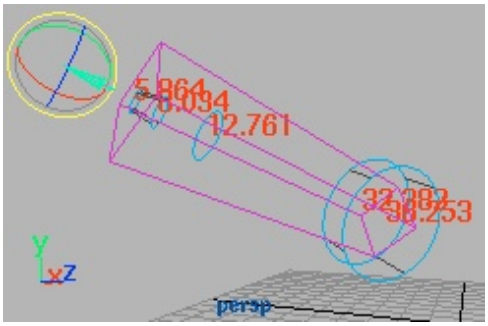


Figure 6.22: Maya display of shaftlight preview object.

6.3.4 Shadows

Shadow handling and shadow control are some of the biggest tasks facing a lighting artist. We took considerable pains with our shadow maps, and developed some slightly unorthodox ways of dealing with shadowing issues.

Multiple shadows

All but the most bare-bones light shaders supported multiple shadow maps. As mentioned above, some versions of *sqLight* supported as many as fifty shadow maps, and for some scenes (think: crowds) every single one of those slots were used.

Even for simple scenes, the use of multiple shadow maps was encouraged and often required. There were a few reasons for this.

Figure 6.23: Shadows in General Hein. In this case, the chair could be safely integrated into Heins shadow. ©2001 FFFP. All Rights Reserved. Square Pictures, Inc.

Most shots were rendered in multiple layers — if Aki and Gray were together in a shot, and each was rendered separately, it was likely that we would have a shadow from Aki for light “shdOverLeft” and also a shadow for Gray from the same light. When rendering Gray, we would need to have the light using “gray.shdOverLeft” also call “aki.shdOverLeft.” Even a very simple scene may have many shadows which cross between the final rendered layers.

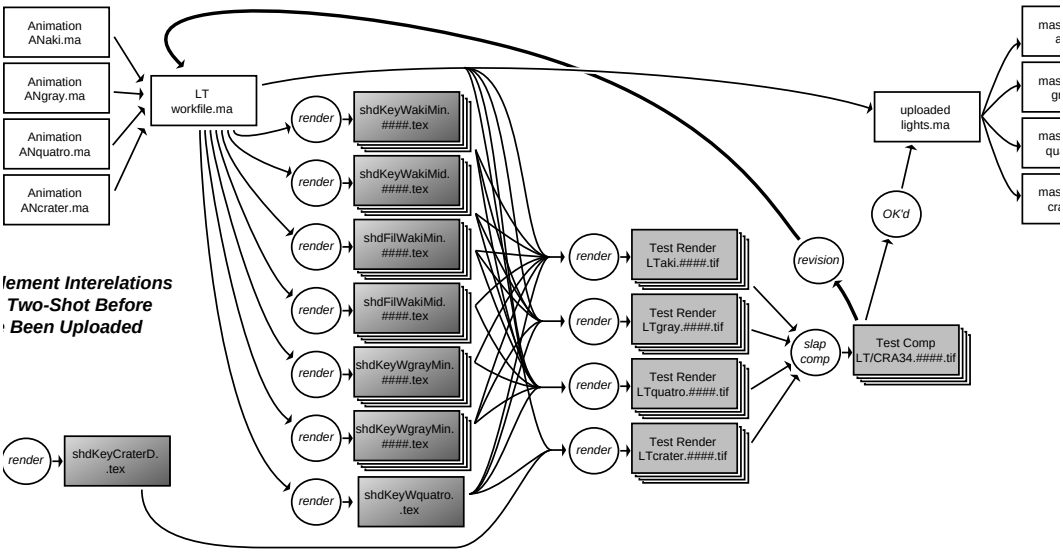


Figure 6.24: Shadow and scenefile relationships for a simple scenes preparation before final render.

Characters were typically rendered using multiple shadow maps for the characters themselves. We would typically use different shadows for the solid parts of the body and the hair, so that they could be handled separately. The densities, blurs, and other parameters of the hair were typically very different from the skin portion of the shadowing, and were rendered in a different map.

Shadow Storage

As a rule, it was efficient to precalculate and store shadows between render passes of the same shot, rather than recalculating the shadows each time the shot was re-rendered. Not only did this let us share shadows between multiple layers in the same image, it had a number of other benefits, particularly in quality control, tuning, and efficiency.

If a single shadow map was miscalculated, or failed to calculate properly (and occasionally identifying problem shadow maps was *very* difficult), then an entire frame would have to be recalculated. Since shots typically had anywhere from 5 to 20 shadow maps, even a failure rate of, say, 1/4 of a percent could ultimately mean that a 100 frame shot would typically have several bad frames due to bad shadow map calculations, and sometimes these errors might be hard to spot at first glance.

It became part of our standard habit to make small grayscale movies of each animated shadow pass, and to review those movies before sending any shot to final render. Reviewing these intermediate results trapped many errors — not only just render glitches, but artist errors as well.

While saving shadows may sound like an expensive proposition from a disk-storage point of view, there was a hidden storage benefit. A typical shadow map would typically consume one or two Megabytes of space. The RIB file for that shadow map could be anywhere from 5 to 100 Megabytes. So when shadows were calculated on the fly for each frame and then discarded, a 10-shadow scene could easily consume hundreds of Megabytes of RIB storage *per frame* while rendering, while the entire shadow storage for the same 100-frame shot might only consume the same amount as that one frames RIBs. If frames were being calculated in parallel (and usually were), then the total peak-use disk savings could be in the tens of Gigabytes.

Finally, once the shadows were calculated and saved, they needed no more CPU time for re-render. Most shots were rendered multiple times between the end-of-animation check and the final beauty pass — so eliminating those redundant shadow renders could become a significant time saver later.[3]

MtoR stored all shadow maps in a single directory, and those directories could sometimes become choked with huge numbers of textures. Simple perl and Mel scripts could clean these files up into subdirectories, and alter the shadow references to point there. In the end, seeking a simple way to manage this led us to decide that light sources should *never* cast shadows — that *all* shadows should come from dedicated “shadow cam” lamps.

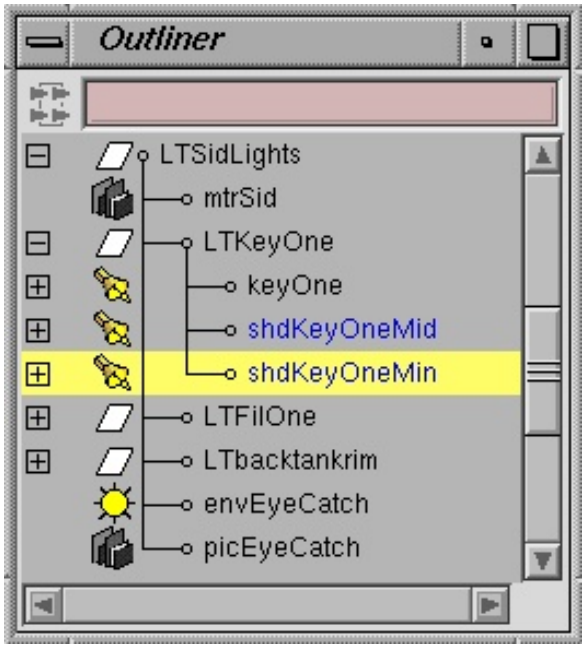


Figure 6.25: Typical “lighting kit” — one illuminating “boss” light, and two shadow cameras for min-hider and midpoint-hider shadow maps.

By separating shadow lamps from illuminating lamps (which we called “boss” lamps), it was

[3]When saving shadows, it was important for use to specify “Open on Frame” motion blur for MtoR. This meant that the motionblur interval for a 180-degree shutter would be from time=0.0 to time=0.5. All *PRMan* shading occurs at the open frame time (0.0). The other option, “Center on Frame,” would use an interval from -0.25 to 0.25. If the shadows had been precalculated, typically with motion blur turned off, they would be aligned for time=0.0. If they were later used in a “Center on Frame” rendering, they would be a quarter-frame out of sync, which would lead to an interesting-to-watch but undesirable “lag” effect.

simpler to manage shadow generation in Maya by just toggling the visibility and template states of the shadow casters, supported in the basic Maya UI, rather than having to fiddle with the more-esoteric Glimpse parameters.

Shadow lamps had a special dummy shader applied, and the `movie_init` light linker would automatically unlink them from the scene if encountered – so if they were accidentally included, theyd be harmless. The special "shadLight" shaders (one for spot sources, another for pointlights) didnt render in the scene, but they did have slots for some extra parameters — not for final rendering, but simply to color and tag their visible icons in the Glimpse palette.

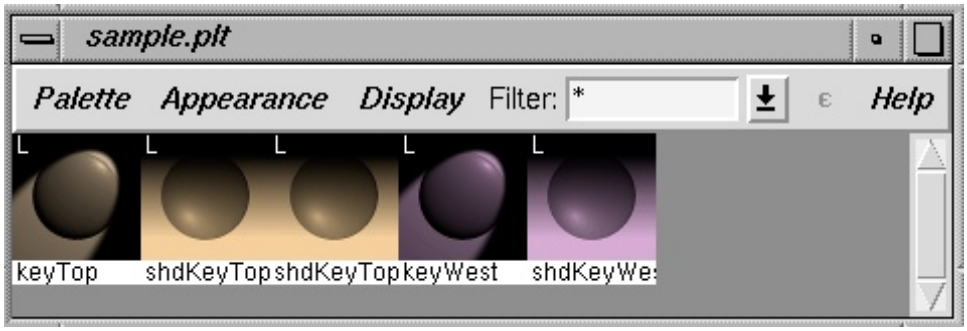


Figure 6.26: Shadow-camera lights still had appearances in Glimpse. Colors could be used to provide a quick visual reference of the scene organization for lighting artists.

MtoR supported a "Map Depth" parameter, used to specify shadow far clipping planes. But RAT3 set the shadow-render near clipping plane to fixed value of 0.001 — a useful number for many shadows, but a real problem at times too. In particular, the shadows cast by lights carried by characters (such as the Deep Eyes goggles) tended to have very severe eyesplits problems when rendering.

As a workaround, and since shadow lights were dedicated exclusively to shadow use, we could use the Maya "decay region" markers to indicate the desired near and far clipping planes, and then have `movie_init` override the clipping values written by MtoR. This provided the Maya artists with another built-in UI that could be easily grabbed and dragged in a Maya window.

Shadow Guides

Often, characters or lights moved in a shot, and animating the clipping planes could be a chore. To simplify those tasks, we added Maya-based *shadow guides* — a shadow guide was simply a sphere, represented by an mtorCoordSys object, and then shadows could be constrained to this target. The shadow cone angles and clipping planes would be automatically set by Maya expressions to exactly enclose the guide, and the guide itself could be handily dragged, constrained to character locations, etc. This made shadow setup very fast for animated characters or moving lights.

Figure 6.27: A problematic scene without shadow-clipping control. The shadows cast by the goggle light (inside the lamp housing) would included the goggle assembly and lenses, unless we had either appropriate shadow-map clipping or struggled with complicated manipulation of shadow sets. ©2001 FFFP. All Rights Reserved. Square Pictures, Inc.

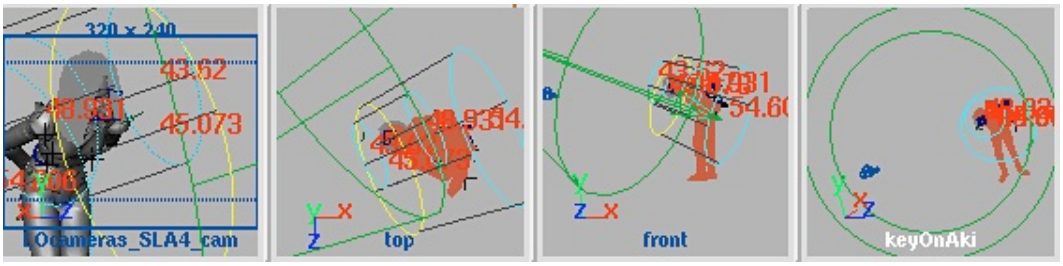


Figure 6.28: Using shadow guides. A small shadow guide has been constrained to Akis head, for shadowing her hair as she moves. A larger guide is constrained to her waist, and shadows her body. The rings show the cone and clipping bounds of the shadow maps. The view from the boss lamp "keyOnAki" shows that only a small area of the lamp needs to be shadowed. Shadow map resolutions on the hair in closeup might be 4096x4096 or higher — if the entire lamp angle was shadowed, the shadow map might have needed to be 32768x32768. ©2001 FFFP. All Rights Reserved. Square Pictures, Inc.

Min + Mid by name

As of *PRMan* 3.8, a number of different algorithms could be used to determine the depth values of a shadow map. The "midpoint" hider method is the same as that used by Maya, and was much-favored by users as the simplest shadow to use — in all but a few cases, the shadow bias could be left at zero.

For some shadowing applications, however (particularly hair and some kinds of set lighting), "min" shadows were still preferred. MtoR didnt provide a direct means to choose the hider, so that task too fell to `movie_init`. The same Tcl code that determined clipping planes would also assign

Figure 6.29: Shadowing for hair almost always needed to be separated from skin self-shadowing.

the shadow hider for each individual shadow, based upon the name. We chose "midpoint" hiding as the default, but shadows that contained "Min" or "Max" in their names would use those hiders instead. Again, the separation of boss light and shadow light made this easier to manage.[4]

Hair was particularly troublesome for midpoint shadowing. As mentioned earlier, hairs tended to be smaller than a pixel — sometimes even when using shadow simplified versions. As sparse areas of hair animated, midpoint shadows tended to jump forward and back quite a bit, as the different hairs obscured or crossed one another. If shadow blurring was enabled, this would lead to a distracting buzz in the shadow. Using "min" shadows for hair generally avoided this problem.

Only one character was partially immune to this midpoint-hider problem, because her hair was so thick — Aki. But since she was our featured character, it was rare that we were able to skimp on her shadow detailing.

`movie_init` also automatically provided a backplane for "midpoint" shadows, so that surfaces with no second layer behind them wouldnt disappear entirely from the shadow — instead they would be assigned a value equal to the midpoint between the surface and that backplane (in otherwise empty shadow pixels, that backplane would disappear). Keeping those midpoint values close to the actual surface was yet another incentive for us to have well-chosen shadowmap clipping planes.

Midpoint catcher & layer issues

While midpoint hiding works well for the point-sampled case, it can have problems in situations where shadows have been blurred — and shadows will typically be blurred, even in the simplest production cases. Most midpoint-related problems were associated with interior concave surfaces.

[4]Shadow-specific `movie_init` usage was a leading reason for us to prefer the "subframe" method of RIB generation — while nominally more expensive to generate, individual shadow RIB files were much easier to manage and to alter via `movie_init`.

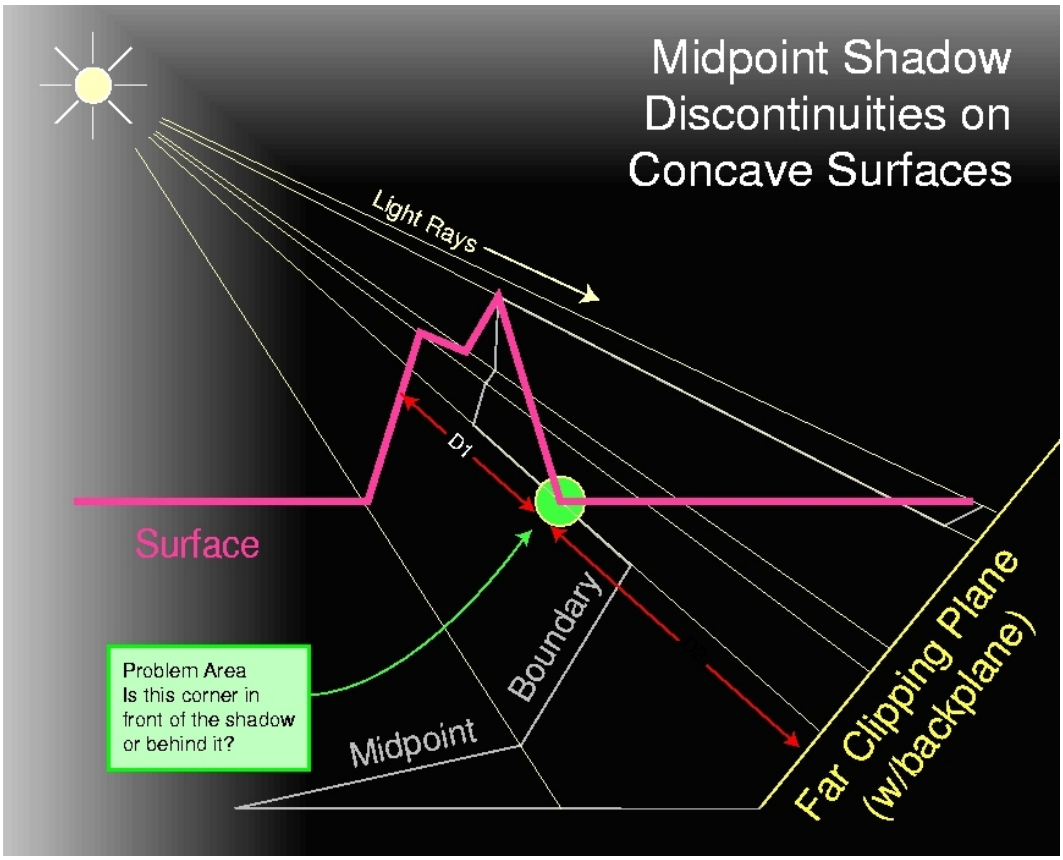


Figure 6.30: The depth of a midpoint shadow is altered by surfaces that may be well-behind the desired shadow depth. In this example, the surface point indicated by the green circle could potentially be both in and out of the shadow-mapped volume. If that point is visible from the camera, if will flicker or appear to have unwanted illumination. Either the midpoint shadow needs to be more finely tuned, or a min-distance shadowmap is required.

Midpoint shadows are easy to recognize as grayscale images by their "x-ray" quality — we can see a lot of the internal details of a model. In models that are mostly hollow, this can mean dramatic changes in the depth samples for a smooth surface, based upon the surfaces hidden behind it. When the shadow is blurred, these different values can "pollute" the shadowing — areas may become unexpectedly darkened, or areas that we thought were in shadow may be lightened because of these hidden hazards. We found these unwanted shadow artifacts to be particularly common on areas of high curvature on the face — noses, eyebrows, the undersides of necks, etc.

Switching back to "min" could solve some problems, but then gave the artists new headaches in the form of handling shadow bias.[5] As a means to solve most of the problems in a relatively painless way, we first started insert extra hidden objects into the characters — a disk to trap shadows at the hole in the neck, spheres inside the skull, etc. Eventually, we added a secondary surface to the interiors of character faces. This "midpoint catcher" surface would deform with the character envelope and only appeared during shadow renders — while extremely-glancing shadows still had occasional problem, especially in extreme closeup, the midpoint catchers solved most of the mid-

[5]MtoR shadow bias trick: create a dummy coordinate system, place it alongside the surface, and scale it down to the size you think is appropriate for shadow bias.... not fancy, but it provides the user with a quick visual reference for selecting shadow bias. One can even use the MtoR mattr() function to copy the scale directly into the shader bias parameters.

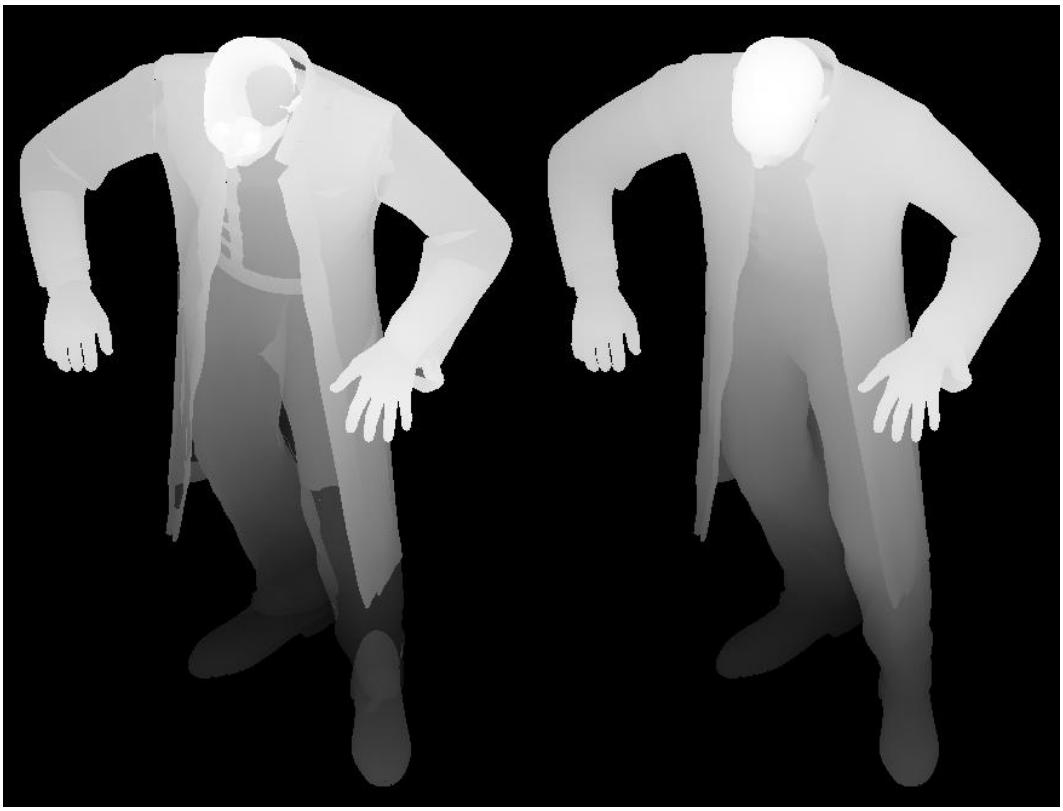


Figure 6.31: Midpoint and min shadows on General Hein (sans hair).

point problems, permitting the artists to continue using midpoint shadows for most shots without ever worrying about shadow bias.[6]

Shadow offsets and Insets

Often, shadows ended up needing to be nudged a bit in one direction or another. The `shadow()` "bias" control provided us a method for pushing the shadow away from its source, but often we wanted to scoot a shadow slightly in other directions.

Shadow offsets were provided in some varieties of sqLight, so that shadows could be adjusted and moved in space without having to move the light or the shadowing object(s). This was often useful for shots where specific features (such as a characters eyes) needed to land exactly in the light, but the "real" shadow wasnt cooperative. The method is very simple: instead of calling `shadow(P,args...)`, we instead pass an additional *shadowOffset* vector into the shader, and call `shadow((P+shadowOffset),args...)`.

As mentioned in the section on skin shading, often we wanted surfaces to receive light from lights behind that surface, either to emulate subsurface scattering or to emulate area-light "wrap." Shadowing becomes problematic in such cases, because the shadows cut off the light before it reaches those over-the-horizon points.

[6] All *PRMan* users should also know: you can add bias to any shadow, whether it was calculated as "min" or not! This simple truth often solved shadowing problems. Once the map is made, it's just a map, the renderer doesn't care how the depth samples were made, and it will happily add bias to any of them.

Figure 6.32: Shadow offset. In the second image, (0,0,3) has been added to the point passed in `shadow()` calls.

We developed a few mechanisms to solve this problem.

The first was to use light categories, split the lights into two different sources, and render one with a shadow and one without. This method was complicated and didnt handle many cases.

The second was to tag some shadows as "translucent," and to use message passing between those light sources and surfaces to determine if a surface needed special handling. If it did, then additional light would be passed in a secondary parameter called "sukeru" (Japanese meaning "see-through"). The light sources would compare the angle between the surface `N` and the incoming `L` vectors — if `N` was facing away from the light, then the shadows would be suppressed and "sukeru" would be present. This method was also complicated, required tagging lights, and was susceptible to aliasing problems along contour edges.

The third method proved the most successful and the simplest. We simply displaced the point `P` used for shadowing along the surface normal `N`, forcing the shadow to be sampled inside the object. This shadow "inset" could be used like shadow bias (though more expensive to use, since its calculated by the shader), and was easy for the users to understand and use whenever shadowing artifacts were seen along contour edges. We found the inset method to be useful in other cases, too — particularly when a lightsource is grazing a surface and the very-stretched shadow exhibits aliasing, we could just inset the shadow and ignore it.

So to account for both shadow offsets and insets, we would use slightly-modified `shadow()` calls: `shadow((P+ shadowInset * normalize(N)+shadowOffset),`*`args...`*`)`

Insetting the shadow is somewhat similar to the solutions described by Sony at last years RenderMan course (they either used a scaled-down copy of the geometry to make the shadow, or blurred the shadow map). The advantage of shadow insetting is that its a single control and contained entirely within the light shader.

Shadow Blur

Shadow blurs varied a lot from one lighting artist to another, and shadow blur was a huge contributor to long render times — especially on Aki. The chart below tells a lot of the story.

The effects in this chart apply four shadows, and Akis hair shadows havent been tuned. By unlinking some of the shadow information from Akis hair itself, the overall frame times could be kept down somewhat while still permitting blur on the skin.

Whenever possible, shadows from other objects or characters were best converted to a non-shadow texture and used as a slide or cookie. A textured blocker with a pre-blurred texture could render orders of magnitude faster than a heavily filtered shadow map.

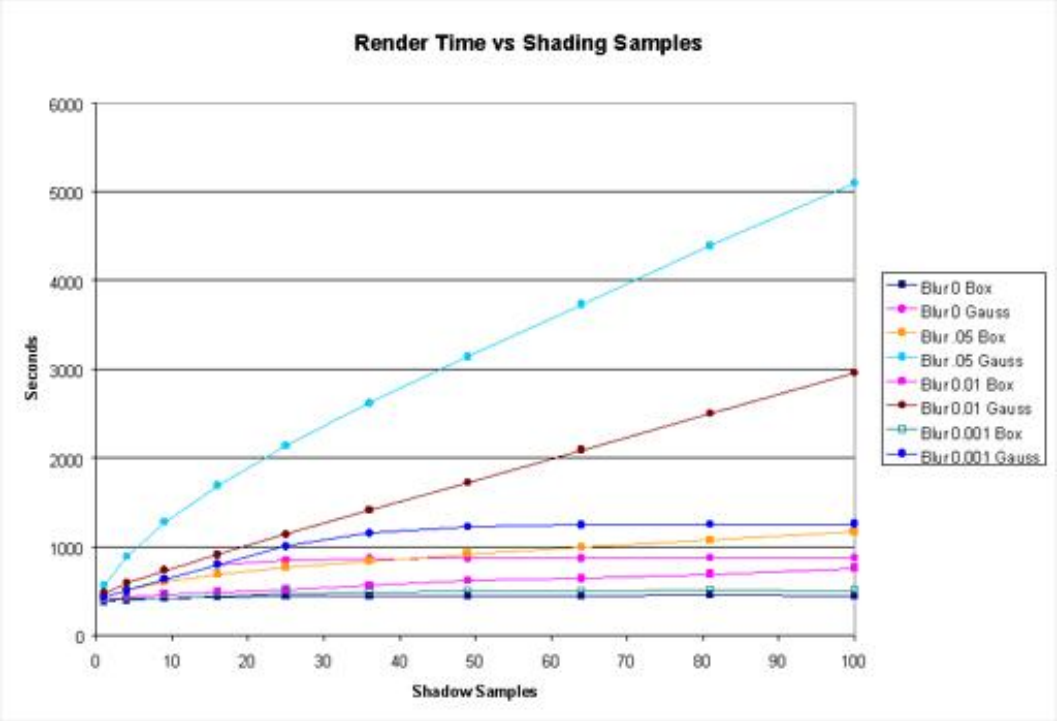


Figure 6.33: Effects of different shadow blur methods and sampling rates on the sample images of Aki in the following figure.

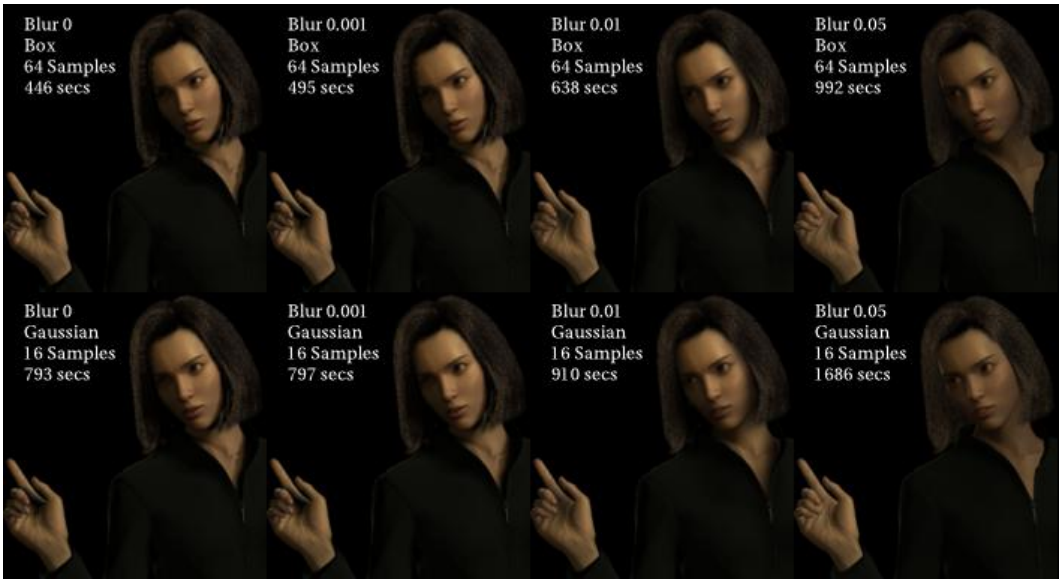


Figure 6.34: Part of a chart used as a guide for artist use of shadow blur and shadow sampling. Note that shadow blur is a function of shadow view size, not 3D worldspace size, so this guide can show aliasing effects and relative render times, but not "how much blur is appropriate for Aki?"

6.3.5 Other Surfaces

Most surfaces in Final Fantasy used large general-purpose shaders. These shaders allow a "layering" approach like that used in "magicLight" — additional surface information was included in a lightsource, and the lightsource closely-bound (attached) to the surface. The "baseLayer" surface can iterate through these special lights and extract all the texturing and coloring information it needs to add any number of extra layers.

Our in-house variants were designed to interchange information with special light shaders like "softbox" and to be consistent with Mayas conventions. We also added a number of optional secondary-output channels to these shaders, including the ability to create masks, to tag surfaces with unique numeric ID numbers, and to push shadow data into an additional channel we called "kagee."

"Kagee" is a Japanese word meaning "shadow picture." By using "kagee" as a secondary output, we could avoid having to render scenes multiple times with shaders like the "ratCollector" shadow-catcher. T downside of "kagee" was that current implementations of prman force the renderer to use point-sampled pixels when secondary outputs are enabled.

When kagee was used, shadows could be captured in a separate image, and then the shadow and basic RGB could be combined in compositing. There were a few problems with the this approach, including the presence of undesired highlights, but at times it was the fastest and most-practical way to bring together multiple elements.

We also made occasional use of surfaces that contained their own calls to the `shadow()` functions, so that shadow mattes could be made without any light sources at all. A function of the form:

```
color cMixed = mix(colorA,colorB,shadow("myShadow.0034.tex"));
```

Figure 6.36: Shadow maps for victims of the New York massacre, shot "BNA3." ©2001 FFFP. All Rights Reserved. Square Pictures, Inc.

The most extreme example was in the New York City phantom massacre scene, where some shots had to show the shadows of many dozens of victims of the alien attack. Each cluster of victims was rendered separately, and their shadows combined into a large single shadow plate.

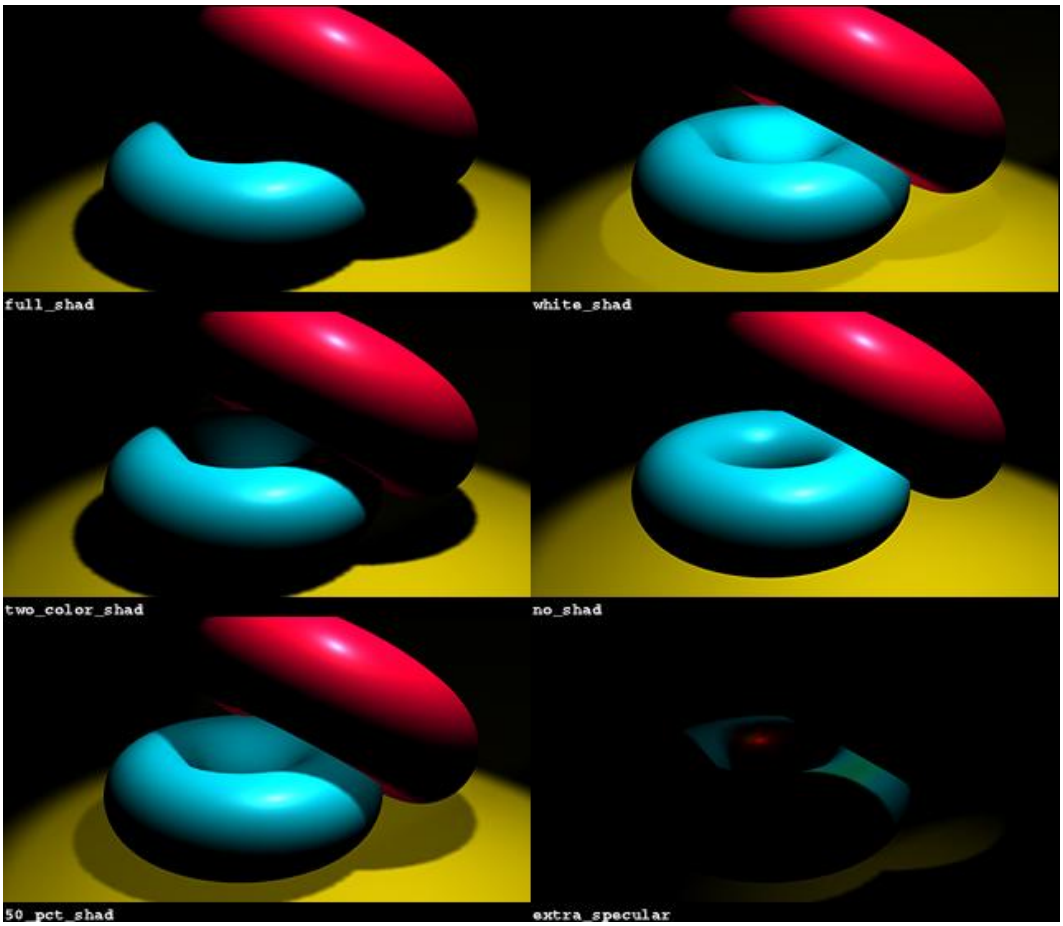


Figure 6.35: Altering shadow density and comparing the results to a kagee shadow-matte image. SqLight (like uberLight) suppresses the specular component of light within a shadow, so even if the shadow color is set to white, the specular is still diminished. Comparing the results of an unshadowed image altered by a shadow matte versus an image rendered with shadows, the difference is revealed as "extra specular."

This approach not only let the director and compositor have greater freedom when combining the background and many victim elements, but it also was useful for scheduling, since the background and foreground images were prepared by different people who followed disjoint schedules.

False-Color Rendering

A number of shaders were written to provide "false color" images, which were subsequently used by the compositors to build the final image. Examples include sparkling sprites, smoke and flame layers, the beams from some light sources such as the Deep Eyes goggles, and the alien Phantoms themselves, which were rendered in multiple layers to provide data for image distortion and internal details.

Figure 6.37: Shadows on floor (and escalator) using “superKagee” in shot”BNA3” — a surface shader that could import multiple shadow maps.

6.3.6 Volumes

Since compositing could add most depth effects, almost no volume shading was done for *Final Fantasy.* The exceptions were special volume shading passes done in the VFX group, using volume effects tied to specific geometric objects, as described by Mach Tony Kobayashis “Stupid RenderMan Trick” from the 2000 Siggraph. This trick and its variations found application in numerous guises, including not only explosions and smoke but also spotlights and laser beams.

6.4 Credits

A large project needs a large number of people to support it. Final Fantasy had RenderMan support from both Pixar and our Japanese reseller, CTC Create. Alias/Wavefront provided on-site support for Maya on every revision. Tadashi Endo originated the Maya shading network for skin and made the original hair shaders. Kaveh Kardeh & Mike Fu originated the idea of shadow simplification. Other RenderMan-intensive staff included Satomi Uchida, Shinya Yarimizo, Kelly Cowles, Mach Tony Kobayashi, Dave Seager, Steven Preeg, Motohisa Adachi, Takumi Kimura, Kevin Ochs, and many others who were users or made suggestions about the design and pipeline. Id also like to recognize the producers of Final Fantasy for letting us share some of these techniques with the CG community at Siggraph.

6.5 Conclusion

Using Maya with RenderMan, we were able to bridge between the art-intensive skills of our staff, who had previously been using packaged software, and still tap into the high-end possibilities inherent in the programmable RenderMan environment. By enhancing MtoR via scripting, we were

Figure 6.38: Goggle light cones were rendered as geometry in a separate false-color pass, then added to the final image in compositing. Shadow maps could be included in the false-color cones as transparency masks, so cones did not appear in shadowed areas.

able to gain key efficiency and project-management aspects of the pipeline that are often ignored or overlooked by smaller-scale projects. By careful control of lighting and more than a few rendering cheats, we were able to bring a strong, realistic look to the sets, effects, and characters of *Final Fantasy.*

6.6 Listings

Listing 6.1: `WPrefix.tcl`

```
From lg Sat Apr 21 11:45:49 2001
Received: from hawaii.rr.com (hnlmail1.hawaii.rr.com [24.25.227.33])
        by barney.sfrn.dnai.com (8.11.2/8.11.2) with ESMTP id f3LIH1K65445
        for <lg@exluna.com>; Sat, 21 Apr 2001 11:17:01 -0700 (PDT)
Received: from [192.168.123.25] ([24.161.143.117]) by hawaii.rr.com  with Microsoft SMTPSVC(5.5.1877.517.51);
         Sat, 21 Apr 2001 08:17:03 -1000
Mime-Version: 1.0
X-Sender: bjorkek001@pop-server
Message-Id: <a05010406b7077d6b5624@[192.168.123.25]>
Date: Sat, 21 Apr 2001 08:15:03 -1000
To: Larry Gritz <lg@exluna.com>
From: Kevin Bjorke <bjorkek001@hawaii.rr.com>
Subject: wpx.tcl
Content-Type: text/plain; charset="us-ascii"
X-Envelope-To: <lg@exluna.com>
X-UIDL: _s-P+d6c46.barney.sfrn.dnai.com
Status: RO
Content-Length: 2613
Lines: 113

#
# Simplified [wPrefix]
#
#
# when $SHOTINFO is not empty, it will contain the prefix of the currently-executing
#    referenced shotinfo.tcl attribute, if such shotinfo commands are executing.
#
global SHOTINFO
set SHOTINFO ""

#
# does the current scene have the named node?
#
proc has_node { nodeName } {
    if {"$nodeName" == ""} {
        return 0
    }
    if {[catch {set fnd [mel "objExists $nodeName"]} msg]} {
        sqLog Problem with $nodeName msg $msg
        return 0
    }
    return $fnd
}


#
# get prefix from $OBJPATH and attach to coordsys names
#
# nodeName - original name
# args - optional list of prefered prefixes, to permit
#       multiple referencing of a shared coordsys
#
proc wPrefix { nodeName args } {
    global OBJPATH SHOTINFO
    #
    # If we are executing within a Glimpse palette, Maya nodes are invisible,
    #   so just return the node name
    #
    if {("$nodeName" == "world") ||
        ("$nodeName" == "glimpse")||
        ("$nodeName" == "")||
        ("$OBJPATH" == "glimpse")||
        ("$OBJPATH" == "world")||
        ("$OBJPATH" == "")} {
            return $nodeName
    }
    #
    # if name is already valid node, do nothing
    #
    if {[has_node $nodeName]} {
        return $nodeName
    }
    #
    # See if the prefered prefixes (if any) are existing Maya namespaces
    #
    regsub -all {[\{\}]} $args "" aa
    foreach path [split $aa] {
```

```
        set qualified "${path}:$nodeName"
        if {[has_node $qualified]} {
            return $qualified
        }
        set qualified "${path}_$nodeName"
        if {[has_node $qualified]} {
            return $qualified
        }
    }
    set path {}
    #
    # see if we are executing a referenced file's shotinfo data
    #
    if {"$SHOTINFO" != ""} {
        set qualified "${SHOTINFO}:$nodeName"
        if {[has_node $qualified]} {
            return $qualified
        }
        set qualified "${SHOTINFO}_$nodeName"
        if {[has_node $qualified]} {
            return $qualified
        }
    }
    if {[string first "|" $OBJPATH] < 0 } {
        #
        # rare case -- no "|" in the name
        #
        if {[regsub {([^:]+):.*} $OBJPATH {\1} path] < 1 } {
            # no namespace, maybe a prefix
            if {[regsub {([^_]+)_.*} $OBJPATH {\1} path] < 1 } {
                return $nodeName
            }
        } else {
            # found a namespace
            return "${path}:$nodeName"
        }
    } else {
        #
        # if there's no prefix in the current $OBJPATH, do nothing
        #
        set q [string range $OBJPATH [string last "|" $OBJPATH] end]
        if {[regsub {\|([^:]+):.*} $q {\1} path] < 1 } {
            # no namespace, maybe a prefix
            if {[regsub {\|([^_]+)_.*} $q {\1} path] < 1 } {
                return $nodeName
            }
        } else {
            # found a namespace
            return "${path}:$nodeName"
        }
    }
    # tried everything else....
    return ${path}_$nodeName
}
```

Listing 6.2: `isShadowPass.h`

```
/*********************************************************/
/*** Return 1 if we're pretty sure this is a shadow pass ***/
/*********************************************************/

float is_shadow_pass()
{
    float fmt[3];
    string sc;
    float is = 0;
    option("Format",fmt);
    attribute("identifier:name",sc);
        /* square pic - can't be a regular camera */
    /* not a glimpse editor window? */
    if ((sc != "<unnamed>") &&
        ((fmt[0] == fmt[1]) && (fmt[2] == 1.0)) )  {
            is = 1;
    }
    return(is);
}
```

Listing 6.3: `Makefile`: Generic shader Makefile caption.

```
#
# Generic Shader Makefile with M4 and BMRT support
#
#   Uses GNU make
#

SHELL = /bin/sh

.SILENT :
.SUFFIXES :
.SUFFIXES : .sl .slo .slc .slm4
.PRECIOUS : %.slo %.slc %.sli

INCLUDES = -I${RATTREE}/lib/shaders/src -I${TOOLINC}
TOOLINC = /gaia/tools/renderman/shadersrc/include

PRMANDIR = ${RMANTREE}/bin
PRMANVERS = prman3.9.1
SHADER = /app/${PRMANVERS}/bin/shader ${INCLUDES}

BMRTDIR = /sqhnl/app/BMRT/bin
SLC = ${BMRTDIR}/slc ${INCLUDES}

M4 = /usr/bin/m4

#
# rules for making shaders and sli files
#

%.slo : %.sl
        ${SHADER} $<

%.slc : %.sl
        ${SLC} $<

%.sl : %.slm4
        -${M4} $< > $@

SOURCES = $(wildcard *.sl)

SLOTARGS = $(patsubst %.sl,%.slo,${SOURCES})
SLCTARGS = $(patsubst %.sl,%.slc,${SOURCES})
SLITARGS = $(patsubst %.sl,%.sli,${SOURCES})

all: slo sli

slo : ${SLOTARGS}

slc : ${SLCTARGS}

sli : ${SLITARGS}

#
# danger -- what if we don't have the source for these .slo/.slc files?
#
.PHONY : clean
clean :
        -rm ${SLODIR}/*.slo
        -rm ${SLCDIR}/*.slc
```

Listing 6.4: softBoxes.slm4

```
divert(-1)
#
############ This is the M4 File and is okay to edit ##############################
#
# to get VIM to act like this is an .sl file, try
#
# :source $VIM/syntax/cpp.vim
#
divert
/*****************************************************************************/
/*****************************************************************************/
/*** DO NOT EDIT THE .SL FILE FOR THIS SHADER ********************************/
/*** INSTEAD EDIT THE .SLM4 FILE WHICH GENERATES THE .SL FILE ****************/
/*****************************************************************************/
/*****************************************************************************/

/*
* To do: accomodate negative widths. twosided/onesided cards.
*/

/*
** $Id: softBoxes.slm4,v 1.1 2001/04/21 21:50:06 lg Exp $
**
** Derived from softbox3 v1.2 - Author Bjorke for all
**
** Full RCS log in .slm4 file
divert(-1)
**
** $Log: softBoxes.slm4,v $
** Revision 1.1  2001/04/21 21:50:06  lg
** *** empty log message ***
**
**
**
divert**
**
*/

#define BOOL float
#define ENUM float

#define SHAD_BOX_FILT 0
#define SHAD_GAUSSIAN_FILT 1

#define TEX_GAUSSIAN_FILT 0
#define TEX_BOX_FILT 1
#define TEX_RADIAL_FILT 2
#define TEX_DISK_FILT 3

#define HALFSIZE

#define OBEY_THIS "Reflectivity"

#define CLASSIC 0
#define SPHERICAL 1

#define CUBEFACE 0
#define LATLONG 1

divert(-1)

/*****************************************************************/
/*** M4 DECLARATIONS *********************************************/
/*****************************************************************/

define(M4_SOFTBOX_DECLARE,`string       boxCoord$1      = "";
    color       boxColor$1      = color (1,1,1),        /* multiplied by lightcolor&intensity */
                boxOpacity$1    = color (1,1,1);
    float       boxWidth$1      = 1,
                boxWEdge$1      = -0.1,
                boxHeight$1     = 1,
                boxHEdge$1      = -0.1,
                boxRoundness$1  = 0;
    string      boxTex$1        = "";
    float       boxFilter$1     = TEX_GAUSSIAN_FILT,
                boxTexStr$1     = 1,
                boxTexBlur$1    = 0;')

define(M4_DO_SOFTBOX,`if (boxCoord$1 != "") {
```

```
            softbox_contrib2(Ps,rv,
boxCoord$1,boxTex$1,boxTexStr$1,
(boxTexBlur$1+materiaBlur),boxFilter$1,
                    boxWidth$1,boxHeight$1,boxWEdge$1,boxHEdge$1,
                    boxRoundness$1,boxColor$1,boxOpacity$1,decayRate,
                    thisDist,thisColor,thisOpac);
            sortedDist[boxCt]  = thisDist;
            sortedColor[boxCt] = thisColor;
            sortedOpac[boxCt]  = thisOpac;
            boxCt += 1;
        }')

define(M4_SHAD_DECLARE,`string shadowname$1 = "";
    ENUM        shadowfilt$1 = SHAD_BOX_FILT;
    float   shadowblur$1 = 0.01,
            shadowbias$1 = 0,
            shadowsamples$1 = 16;')

define(M4_DO_SHAD,`if (shadowname$1 != "") {
            shadowed = sbShadow(shadowname$1,Ps,shadowfilt$1,
                        shadowblur$1,shadowsamples$1,shadowbias$1);
            fullShad = max(fullShad,shadowed);
        }')

define(REALLY_FAR,1000000.0)
divert

/*************************************************************************/
/* Superellipse soft clipping - straight out of "uberlight"             ***/
/* Input:                                                               ***/
/*   - point Q on the x-y plane                                         ***/
/*   - the equations of two superellipses (with major/minor axes given  ***/
/*      by  a,b and A,B for the inner and outer ellipses, respectively) ***/
/* Returns:                                                             ***/
/*   - 0 if Q was inside the inner ellipse                              ***/
/*   - 1 if Q was outside the outer ellipse                             ***/
/*   - smoothly varying from 0 to 1 in between                          ***/
/*************************************************************************/

/* this is the identical function used by sqLight etc */

float clipSuperellipse (
    point Q;          /* Test point on the x-y plane */
    uniform float a, b;       /* Inner superellipse */
    uniform float A, B;       /* Outer superellipse */
    uniform float roundness;  /* Same roundness for both ellipses */
) {
    varying float result;
    varying float x = abs(xcomp(Q)), y = abs(ycomp(Q));
    if (roundness < 1.0e-6) {
        /* Simpler case of a square */
        result = 1 - (1-smoothstep(a,A,x)) * (1-smoothstep(b,B,y));
    } else {
        /* more-difficult rounded corner case */
        varying float re = 2/roundness;         /* roundness exponent */
        varying float q = a * b * pow (pow(b*x, re) + pow(a*y, re), -1/re);
        varying float r = A * B * pow (pow(B*x, re) + pow(A*y, re), -1/re);
        result = smoothstep (q, r, 1);
    }
    return result;
}

/*****************************************************************/
/*** Given info on a softbox, ************************************/
/*****************************************************************/

void softbox_contrib2(
    varying point       surfPt;
    varying vector      reflVect;
    uniform string      boxCoords;
    uniform string      boxTexture;
    uniform float       boxTexStr,
                        boxTexBlur,
                        boxFilter,
                        boxWidth,
                        boxHeight,
                        boxWEdge,
                        boxHEdge,
                        boxRoundness;
    uniform color       boxColor,
                        boxOpac;
```

```
    uniform float       decayExp;
    output float theDist;
    output color theColor;
    output color theOpac;
) {
    uniform string filtTypes[4] = {"gaussian","box","radial-bspline","disk"};
    uniform string theFilterName = filtTypes[clamp(boxFilter,0,3)];
    varying float contrib;
    varying color ct = 1;
    varying float ot = 1;
    /* Get the surface position */
    varying point Pb1 = transform (boxCoords, surfPt);
    varying vector Vlight = vtransform (boxCoords, reflVect);
    varying float zv = zcomp(Vlight);
    varying point Pplane = Pb1 - Vlight*(zcomp(Pb1)/zcomp(Vlight));
#ifdef HALFSIZE
    uniform float bw2 = boxWidth/2;
    uniform float bh2 = boxHeight/2;
#else
#define bw2 boxWidth
#define bh2 boxHeight
#endif
    uniform float we = max(boxWEdge,-bw2);
    uniform float he = max(boxHEdge,-bh2);
    uniform float bW = bw2+we;
    uniform float bH = bh2+he;
    uniform float iW = min(bW,bw2);
    uniform float iH = min(bH,bh2);
    uniform float oW = max(bW,bw2);
    uniform float oH = max(bH,bh2);
    if (sign(zcomp(Pb1)) == sign(zcomp(Vlight))) {
        contrib = 0;
    } else if (abs(zv) < 0.0001) {
        contrib = 0;
    } else {
        contrib = 1 - clipSuperellipse (Pplane, iW, iH, oW,oH, boxRoundness);
        if (boxTexture != "") {
            uniform float nChans;
            textureinfo(boxTexture,"channels",nChans);
            varying float theS = (oW+xcomp(Pplane))/(oW*2);
            varying float theT = (oH-ycomp(Pplane))/(oH*2);
            theS = min(2,max(-1,theS));
            theT = min(2,max(-1,theT));
            if (nChans>1) {
                ct = texture(boxTexture,theS,theT,
                            "filter",   theFilterName,
                            "blur",     boxTexBlur);
                if (boxTexStr != 1) {
                    ct = (ct*boxTexStr)+(1-boxTexStr);
                }
            }
            if ((nChans==1)||(nChans>3)) {
                uniform float alphaChan;
                if (nChans==1) {
                    alphaChan = 0;
                } else {
                    alphaChan = 3;
                }
                ot = float texture(boxTexture[alphaChan],theS,theT,
                            "filter",   theFilterName,
                            "blur",     boxTexBlur);
            }
        }
    }
#pragma nolint
    varying point ppC = transform(boxCoords,"world",Pplane);
    varying point spw = transform("world",surfPt);
    varying float pDist = length(ppC - spw); /* in "world" coords */
    theDist = pDist;
    theOpac = contrib * boxOpac * ot;
    theColor = contrib * ct * boxColor / pow(pDist,decayExp);   /* premultiplied!!!!! */
}

/*************************************************/
/*** SHADOW ******************************************/
/*************************************************/

float sbShadow(
    uniform string      theName;
    varying point       thePoint;
    uniform ENUM        theFilt;
    uniform float       theBlur,
```

```
                        theSamples,
                        theBias;
) {
    uniform string filtTypes[2] = {"box", "gaussian"};
    uniform string theFilterName = filtTypes[clamp(theFilt,0,1)];
    varying float inShadow = shadow (theName,
                             thePoint,
                             "filter",   theFilterName,
                             "blur",     theBlur,
                             "samples",  theSamples,
                             "bias",     theBias);
    return(inShadow);
}

/*****************************************************************/
/*** MAIN SHADER *************************************************/
/*****************************************************************/

light softboxes(
    string      Comment = "";
#ifdef BMRT
    string      Tcl = "";
    string      NodeName = "";
#else /* !BMRT -- slc compiler doesn't like these definitions */
    string      Tcl = "[addL]";
    string      NodeName = "$OBJNAME";
#endif /* BMRT */
    float       intensity       = 1;
    color       lightcolor      = color (1,1,1);
    float       decayRate       = 0;
    float       meterDistance = 1;
    string      meterSpace = "";
    float       edgeRolloff = 0,
                edgeAngle = 90,
                edgeExp = 1;
    M4_SOFTBOX_DECLARE(1)
    M4_SOFTBOX_DECLARE(2)
    M4_SOFTBOX_DECLARE(3)
    M4_SOFTBOX_DECLARE(4)
    uniform string      envTexName = "";
    uniform float       EnvType = CLASSIC;
    uniform float       MapType = CUBEFACE;
    uniform float envTexIntensity       = 1;
    uniform float envTexBlur    = 0;
    uniform float envTexStr     = 1;
    uniform float envTexFilter  = 0;
    uniform string envReflSpace = "";
    color shadowcolor = 0;
    float shadowintensity = 1;
    M4_SHAD_DECLARE()
    M4_SHAD_DECLARE(b)
    M4_SHAD_DECLARE(c)
    M4_SHAD_DECLARE(d)
    BOOL        NonDiffuse      = 1;
    BOOL        NonSpecular     = 0;
    BOOL        ObeyMateria     = 0;
    BOOL        UseMateriaBlur  = 0;
    output varying float __nondiffuse = 1;
    output varying float __nonspecular = 0;
    string      __category = "reflection";
    output varying float __inShadow = 0;
) {
    uniform string rcsInfo = "$Id: softBoxes.slm4,v 1.1 2001/04/21 21:50:06 lg Exp $";
    uniform string filtTypes[4] = {"gaussian","box","radial-bspline","disk"};
    uniform string theFilterName = filtTypes[clamp(envTexFilter,0,3)];
    normal Nf = faceforward(normalize(N),I);
    vector rv = reflect(I,Nf);
    uniform float edgeLimVal = cos(radians(90-clamp(edgeAngle,0,90)));
    uniform string theEnvSpace;
    if (envReflSpace == "") {
        theEnvSpace = "shader";
    } else {
        theEnvSpace = envReflSpace;
    }
    uniform float adjMeterDistance;
    if (meterSpace == "") {
        adjMeterDistance = meterDistance;
    } else {
        uniform point metP = transform(meterSpace,"shader",point (0,0,0));
        adjMeterDistance = length(metP) + meterDistance;
    }
    uniform float adjIntensity = pow(adjMeterDistance,decayRate)*intensity;
```

```
    varying float fullShad = 0;
    uniform float materiaRefl = 1;
    uniform float materiaBlur = 0;

    Cl = 0;
    __nondiffuse = NonDiffuse;
    __nonspecular = NonSpecular;
    if (ObeyMateria > 0) {
        if (surface("Reflectivity",materiaRefl) == 0) {
            if (surface("abReflectivity",materiaRefl) == 0) {
                if (surface("reflectivity",materiaRefl) == 0) {
                    if (surface("Kr",materiaRefl) == 0) {
                        materiaRefl = 0;
                    }
                }
            }
        }
    }
    if (UseMateriaBlur > 0) {
        if (surface("ReflectionMapBlur",materiaBlur) == 0) {
            if (surface("abReflectionMapBlur",materiaBlur) == 0) {
                if (surface("reflectionMapBlur",materiaBlur) == 0) {
                    materiaBlur = 0;
                }
            }
        }
    }
    solar() {
        if (materiaRefl != 0) {
            uniform float boxCt=0;
            varying float thisDist;
            varying color thisOpac;
            varying color thisColor;
            varying float sortedDist[4];
            varying color sortedOpac[4];
            varying color sortedColor[4];
            M4_DO_SOFTBOX(1)
            M4_DO_SOFTBOX(2)
            M4_DO_SOFTBOX(3)
            M4_DO_SOFTBOX(4)
            if (envTexName != "") {
#pragma nolint
                varying vector Rs = normalize (vtransform (theEnvSpace, normalize(-L)));
                if (EnvType == SPHERICAL) {
#pragma nolint
                    varying point PShd = transform (theEnvSpace, Ps);
                    varying float pl = vector(PShd).vector(PShd);
                    varying float pdotv = -vector(PShd).Rs;
                    Rs = vector( PShd + (pdotv + sqrt (abs (1 - pl + ((pdotv)*(pdotv)))))*Rs );
                }
                if( MapType == LATLONG ) {      /* latlong */
                    Rs = vector (-zcomp (Rs), xcomp (Rs), ycomp (Rs));
                }
                Cl = color environment (envTexName, Rs,
                                        "filter", theFilterName,
                                        "blur", (envTexBlur+materiaBlur));
                if (envTexStr != 1) {
                    Cl = (Cl*envTexStr)+(1-envTexStr);
                }
                Cl *= envTexIntensity;
            }
            if (boxCt > 0) {
                uniform float i, j, k;
                for(k=0; k<(boxCt-1); k+=1) {
                    j = 0;
                    for(i=1; i<boxCt; i+=1) {
                        if (sortedDist[i]>sortedDist[j]) { /* farthest first */
                            thisDist = sortedDist[j];
                            thisOpac = sortedOpac[j];
                            thisColor = sortedColor[j];
                            sortedDist[j] = sortedDist[i];
                            sortedOpac[j] = sortedOpac[i];
                            sortedColor[j] = sortedColor[i];
                            sortedDist[i] = thisDist;
                            sortedOpac[i] = thisOpac;
                            sortedColor[i] = thisColor;
                        }
                        j = j+1;
                    }
                }
```

```
                color ttc;
                for(k=0; k<boxCt; k+=1) {
                    ttc = Cl;
                    Cl = sortedColor[k]+(Cl*(1-sortedOpac[k]));
                }
            }
            /* Apply shadow mapped shadows */
            varying vector Ln = normalize(L);
            varying vector Nn = normalize(N);
            varying vector In = normalize(I);
            float theDot = Ln.Nn;
            if (edgeRolloff > 0) {
                float q;
                q = In.Nn/edgeLimVal;
                q = 1 - clamp(edgeRolloff*pow(clamp(abs(q),0,1),1/max(edgeExp,0.001)),0,1);
                Cl = mix(color(0,0,0),Cl,q);
            }
            varying float shadowed;
            M4_DO_SHAD()
            M4_DO_SHAD(b)
            M4_DO_SHAD(c)
            M4_DO_SHAD(d)
            __inShadow = fullShad;
        }
    }
    if (materiaRefl > 0) {
        Cl *= (lightcolor * adjIntensity * materiaRefl);
        Cl = mix(Cl, (shadowcolor*shadowintensity*adjIntensity), fullShad);
    } else {
        Cl = 0;
    }
}

/*************************************************/
/************************************** eof ***/
/*************************************************/
```

Listing 6.5: superKagee.slm4

```
divert(-1)
#
# Glimpse-compatible surface shader that accepts an arbitrary
#       number of shadows -- match basic form of sqLight shadows
#
# permit M4 flags for offset, inset, and soft shadow capability.
#     define m4inset m4offset and/or m4softshad
# define the number of shadows supported by defining
#     the m4 value ‘m4numshads’

# m4 utility functions
define(alpha,‘.abcdefghijklmnopqrstuvwxyz1234567890’)
define(ORD,‘substr(alpha,$1,1)’)
define(‘forloop’,
    ‘pushdef(‘$1’, ‘$2’)_forloop(‘$1’, ‘$2’, ‘$3’, ‘$4’)popdef(‘$1’)’)
define(‘_forloop’,
    ‘$4‘’ifelse($1, ‘$3’, ,
    ‘define(‘$1’, incr($1))_forloop(‘$1’, ‘$2’, ‘$3’, ‘$4’)’)’)

#
# Shadow definitions
#
define(SHAD_DECLARE,‘string     shadowname$1 = "";
            ENUM        shadowfilt$1 = FILT_BOX;
            float       shadowblur$1 = 0,
                        shadowbias$1 = 0,
                        ifdef(‘m4insetshad’,‘inset$1 = 0,
                        ’)shadowsamples$1 = 16,
                        shadowdensity$1 = 1;
            ifdef(‘m4shadoffset’,‘vector        shadowV$1 = vector "world" (0,0,
0);
            ’)ifdef(‘m4softshad’,‘BOOL  softtype$1 = 0;
            float       softsize$1 = 1,
                        softbias$1 = 0;
            ’)’)

define(DO_SHAD,‘if (shadowname$1 != "") {
        ifdef(‘m4shadoffset’,‘uniform vector shV$1 = vtransform("shader",shadowV
$1);
        uniform float shvx$1 = xcomp(shV$1);
        uniform float shvy$1 = ycomp(shV$1);
        uniform float shvz$1 = zcomp(shV$1);
        uniform vector shadoff$1 = vector "world" (shvx$1,shvy$1,shvz$1);
        /* uniform vector shadoff$1 = vtransform("world","current",shV$1); */
        ’)
        shadowed = clamp(shadowdensity$1,0,1) * do_shadow6(shadowname$1,ifdef(‘m
4shadoffset’,‘P-shadoff$1’,‘P’),
                    shadowfilt$1,shadowblur$1,shadowsamples$1,shadowbias$1
                    ifdef(‘m4softshad’,‘,softtype$1,softsize$1,softbias$1’)
                    ifdef(‘m4insetshad’,‘,inset$1,N’));
        fullShad = max(fullShad,shadowed);
    }’)

divert

#define BOOL    float
#define ENUM    float

#define FILT_BOX 0
#define FILT_GAUSS 1

/*************************************************/
/*** SHADOW ************************************/
/*************************************************/

float do_shadow6(
    uniform string      theName;
    varying point       thePoint;
    uniform ENUM        theFilt;
    uniform float       theBlur,
                        theSamples,
                        theBias;
    ifdef(‘m4softshad’,‘uniform float useSoft, softDiam, softGap;
    ’)ifdef(‘m4insetshad’,‘uniform float theInset;
    varying normal      theNormal;
’)) {
    uniform string filtTypes[2] = {"box", "gaussian"};
    uniform string theFilterName = filtTypes[clamp(theFilt,0,1)];
```

```
    float inShadow;
ifdef(`m4insetshad',`    varying point SHADPOINT;
    if (theInset != 0) {
        normal wN = normalize(ntransform("world",theNormal));
        point wP = transform("world",thePoint);
        point wNew = wP + wN * theInset;

        SHADPOINT = transform("world","current",wNew);
    } else {
        SHADPOINT = thePoint;
    }
',`#define SHADPOINT thePoint')
ifdef(`m4softshad',`    if (useSoft < 1) {
    ')    inShadow = shadow (theName,
                             SHADPOINT,
                             "filter",   theFilterName,
                             "blur",     theBlur,
                             "samples",  theSamples,
                             "bias",     theBias);
ifdef(`m4softshad',`    } else {
        uniform point p0, p1, p2, p3;
        uniform float r = softDiam * .5;
        p0 = point "shader" (-r, r, 0);
        p1 = point "shader" (r, r, 0);
        p2 = point "shader" (r, -r, 0);
        p3 = point "shader" (-r, -r, 0);
        inShadow = shadow (theName,
                             SHADPOINT,
                             "source", p0, p1, p2, p3,
                             "gapbias", softGap,
                             "filter",   theFilterName,
                             "blur",     theBlur,
                             "samples",  theSamples,
                             "bias",     theBias);
    }
    ')return(inShadow);
}

/*****************************************************/
/*** SHADER BEGINS ***********************************/
/*****************************************************/


surface superKagee(
    color shadColor = 0;
    forloop(`ssn',1,m4numshads,`SHAD_DECLARE(ORD(ssn))
    ')
) {
    uniform string rcsInfo =
"$Id: superKagee.slm4,v 1.1 2001/04/21 21:50:06 lg Exp $";
    varying float fullShad = 0;
    varying float shadowed;
    forloop(`ssx',1,m4numshads,`DO_SHAD(ORD(ssx))')
    Oi = Os;
    Ci = Oi * mix(Cs,shadColor,fullShad);
}
```

Bibliography

[Apodaca and Gritz, 1999] Apodaca, A. A. and Gritz, L. (1999). *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan-Kaufmann.

[Cohen et al., 1988] Cohen, M. F., Chen, S. E., Wallace, J. R., and Greenberg, D. P. (1988). A progressive refinement approach to fast radiosity image generation. In Dill, J., editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 75–84.

[Debevec, 1998] Debevec, P. (1998). Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In Cohen, M., editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 189–198. ACM SIGGRAPH, Addison Wesley. ISBN 0-89791-999-8.

[Debevec and Malik, 1997] Debevec, P. E. and Malik, J. (1997). Recovering high dynamic range radiance maps from photographs. In Whitted, T., editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 369–378. ACM SIGGRAPH, Addison Wesley. ISBN 0-89791-896-7.

[Dorsey and Hanrahan, 1996] Dorsey, J. and Hanrahan, P. (1996). Modeling and rendering of metallic patinas. In Rushmeier, H., editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 387–396. ACM SIGGRAPH, Addison Wesley.

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

[Glassner, 1995] Glassner, A. S. (1995). *Principles of digital image synthesis*. Morgan Kaufman, San Francisco.

[Goral et al., 1984] Goral, C. M., Torrance, K. E., Greenberg, D. P., and Battaile, B. (1984). Modelling the interaction of light between diffuse surfaces. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 212–22.

[Haase and Meyer, 1992] Haase, C. S. and Meyer, G. W. (1992). Modeling pigmented materials for realistic image synthesis. *ACM Transactions on Graphics*, 11(4):305–335.

[Hall, 1989] Hall, R. (1989). *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, New York. out of print.

[Hanrahan and Krueger, 1993] 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 and Salzman, 1989] Hanrahan, P. and Salzman, D. (1989). A rapid hierarchical radiosity algorithm for unoccluded environments. In *Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics*.

[Jensen, 1996] Jensen, H. W. (1996). Global illumination using photon maps. In Pueyo, X. and Schröder, P., editors, *Eurographics Rendering Workshop 1996*, pages 21–30, New York City, NY. Eurographics, Springer Wien. ISBN 3-211-82883-4.

[Jensen and Christensen, 1998] Jensen, H. W. and Christensen, P. H. (1998). Efficient simulation of light transport in scenes with participating media using photon maps. In Cohen, M., editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 311–320. ACM SIGGRAPH, Addison Wesley. ISBN 0-89791-999-8.

[Kokhanovsky, 1999] Kokhanovsky, A. A. (1999). *Optics of Light Scattering Media: Problems and Solutions*. Great Britan.

[Kubelka, 1948] Kubelka, P. (1948). New contributions to the optics of intensely light-scattering materials, part i. *Journal of the Optical Society of America*, 38(5).

[Kubelka, 1954] Kubelka, P. (1954). New contributions to the optics of intensely light-scattering materials, part ii: Nonhomogeneous layers. *Journal of the Optical Society of America*, 44(4).

[Kubelka and Munk, 1931] Kubelka, P. and Munk, F. (1931). Ein Beitrag zur Optik der Farbanstriche. *Z. Tech. Physik.*, 12:593. Translation by Steve Westin available at http://graphics.cornell.edu/w̃estin/pubs/kubelka.pdf.

[McCluney, 1994] McCluney, W. R. (1994). *Introduction to radiometry and photometry*. Artech House.

[Nicodemus et al., 1977] Nicodemus, F. E., Richmond, J. C., Hisa, J. J., Ginsberg, I. W., and Limperis, T. (1977). *Geometrical Considerations and Nomenclature for Reflectance*. Monograph number 160. National Bureau of Standards.

[Pixar, 2000] Pixar (2000). *The RenderMan Interface, Version 3.2*. Pixar.

[Porter and Duff, 1984] Porter, T. and Duff, T. (1984). Compositing digital images. In Christiansen, H., editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 253–259.

[Smits et al., 1994] Smits, B., Arvo, J., and Greenberg, D. (1994). A clustering algorithm for radiosity in complex environments. In Glassner, A., editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 435–442. ACM SIGGRAPH, ACM Press. ISBN 0-89791-667-0.

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

[Wallace et al., 1989] Wallace, J. R., Elmquist, K. A., and Haines, E. A. (1989). A ray tracing algorithm for progressive radiosity. In Lane, J., editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 315–324.

[Ward, 1994] Ward, G. J. (1994). The RADIANCE lighting simulation and rendering system. In Glassner, A., editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 459–472. ACM SIGGRAPH, ACM Press. ISBN 0-89791-667-0.

[Ward and Heckbert, 1992] Ward, G. J. and Heckbert, P. (1992). Irradiance gradients. *Third Eurographics Workshop on Rendering*, pages 85–98.

[Ward et al., 1988] Ward, G. J., Rubinstein, F. M., and Clear, R. D. (1988). A ray tracing solution for diffuse interreflection. In Dill, J., editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 85–92.