

Advanced RenderMan: Beyond the Companion

Siggraph 1998 Course 11

Monday, July 20, 1998

Tony Apodaca

Larry Gritz

Course Chairs

ABSTRACT

RenderMan has been used by many large and small animation production studios to create high-quality, often photorealistic, imagery for television and motion pictures. Its ability to render extremely complex scenes with motion blur, depth-of-field, and user-programmable shaders has made it the industry leader in feature film CGI, and in recognition of this, it received an Academy Award in 1993. However, much has changed since the “RenderMan Interface Specification” and *RenderMan Companion* were written nearly 10 years ago. This course will teach the modern paradigms of how to use RenderMan. We will discuss generating data to pump into RenderMan renderers, and programming the RenderMan Shading Language to generate special effects. We will examine the production of some famous computer animations made with RenderMan, to show what it really takes to make most effective use of the tools RenderMan provides. We intend to make this both entertaining and informative.

Copyright © 1998 Pixar. All rights reserved.

RenderMan[®] is a registered trademark of Pixar.

Desired Background

This course is for graphics programmers and technical directors. Thorough knowledge of 3-D image synthesis, computer graphics illumination models and previous experience with the RenderMan Shading Language is a must. Students should 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, 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 that it is in great need of an updated edition.

“The RenderMan Interface Specification, Version 3.1,” Pixar

The bible. The first half of this book is the Technical Reference for RenderMan's geometric interface, both C and RIB versions. The second half of this handbook is the Technical Reference for the RenderMan Shading Language.

“Writing RenderMan Shaders,” Siggraph 1992 Course #21 Course Notes

These are the notes to the first advanced course on RenderMan. This is the heavy-duty, very technical course specifically focused on writing high-quality effects in the Shading Language, including topics such as noise synthesis, lighting and antialiasing. The best source available for this topic.

Lecturers

Tony Apodaca

is Director of Graphics R&D at Pixar Animation Studios. Tony is also the lead engineer of Pixar's PhotoRealistic RenderMan image synthesis product, and was co-architect of the RenderMan Interface Specification. In March, 1993, Tony and five other engineers received a Scientific and Technical Academy Award from the Academy of Motion Picture Arts and Sciences for work on this system. Tony has chaired 4 previous Siggraph courses on the use of RenderMan in the production of animation and special effects for motion pictures. Tony received his Master's degree in Computer and Systems Engineering at Rensselaer Polytechnic Institute in 1986, where his thesis topic was interactive image synthesis. His film credits include *Tin Toy*, *knickknack* and *Toy Story*.

Larry Gritz

has been a software engineer in the Graphics R&D group at Pixar Animation Studios since 1995, where for he causes trouble by developing new rendering techniques and systems, and periodically serves as a technical director. His film credits include *Toy Story* and *Geri's Game*, and is currently contributing to Pixar's upcoming film, *A Bug's Life*. Larry's research interests include global illumination, shading languages and systems, and rendering of hideously complex scenes. Prior to joining Pixar, Larry authored the popular "Blue Moon Rendering Tools", a shareware RenderMan-compliant renderer which supports ray tracing and radiosity. Larry has a BS from Cornell University and an MS from the George Washington University. He swears that he really will turn in his dissertation and get his PhD from GWU this year, even though he said the same thing when he taught this course in 1995.

Ronen Barzel

works on tools development at Pixar Animation Studios, where he worked on modeling, lighting, and tools development for the 1995 Disney motion picture *Toy Story*. He holds a BS in math and physics and an MS in computer science from Brown University and obtained a PhD in computer science from the California Institute of Technology. He authored *Physically Based Modeling for Computer Graphics: A Structured Approach* (Academic Press, 1992) and is editor-in-chief of the *Journal of Graphics Tools*.

Antoine Durr

is a Digital Supervisor at BlueSky/VIFX, a Los Angeles, CA based visual effects facility. In the last eight years at BlueSky/VIFX, he helped create the visual effects for feature films including *Alien Resurrection*, *Volcano*, *The Relic*, *Time Cop*, *From Dusk 'til Dawn*, and *Batman Returns*. When between shows, he teaches computer graphics and RenderMan classes to his BlueSky/VIFX coworkers and runs the Los Angeles PRISMS users group.

Clint Hanson

received his B. Mathematics from the University Of Waterloo in 1993. After that he worked at Vertigo Technologies Inc. for 3 years. While there he developed 3D software tools which were used in conjunction with Pixar's RenderMan renderer. He is currently working at Sony

Pictures Imageworks. His list of movie credits include *Contact*, *Escape From L.A.*, *The Postman*, *Mortal Kombat* and *The Santa Clause*.

Scott Johnston

founded Fleeting Image Animation, Inc. in 1997, to develop and produce animation integrating traditional and computer generated techniques. Prior to this, Johnston worked at Walt Disney Feature Animation where he was a principal designer of the ballroom sequence in *Beauty and the Beast* and was CGI Supervisor for *The Lion King*. Johnston is currently serving as Artistic Coordinator for Warner Bros. Feature Animation's upcoming film, *The Iron Giant*.

Schedule

Welcome and Introduction Tony Apodaca Page 1	8:30 AM
Modern Scene Description Paradigms Tony Apodaca Page 3	8:45
<i>Break</i>	10:00
Advanced Shader Writing Techniques Larry Gritz Page 62	10:15
<i>Lunch</i>	12:00 PM
Advanced Techniques for CG Lighting Ronen Barzel Page 96	1:30
RenderMan as an Element in the Production Pipeline Antoine Durr	2:15
<i>Break</i>	3:00
Volume Rendering Effects in <i>Contact</i> Clint Hanson	3:15
Non-photorealistic Rendering Scott Johnston Page 122	4:00

Course Notes Table of Contents

Introduction	Page 1
Modern RenderMan Interface Features	Page 3
Writing Surface Shaders	Page 30
RenderMan Tricks Everyone Should Know	Page 51
Basic Antialiasing in Shading Language	Page 62
Ray Tracing in PRMan	Page 81
Lighting Controls for Computer Cinematography	Page 96
Mock Media	Page 113

CD-ROM Contents

index.htm

Master HTML file with links to all CD-ROM contents and various interesting RenderMan-related resources on the Web.

course11.pdf

These course notes.

barzel

Hi-res color images and shader source code from Ronen Barzel's paper "Lighting Controls for Computer Cinematography."

gritz

Hi-res color images, shader source from Larry Gritz' papers, and a 4/98 distribution of BMRT.

hanson

Shader source code from Clint Hanson's talk on "Rendering Effects Used In *Contact*."

johnston

Images and Quicktime movies referred to in Scott Johnston's paper "Mock Media."

Advanced RenderMan: Beyond the Companion

Tony Apodaca
Larry Gritz
Pixar Animation Studios

Welcome to *Advanced RenderMan: Beyond the Companion*. This course covers the theory and practice of using RenderMan to do the highest quality computer graphics animation production. This is the fourth course that we have taught on the use of RenderMan, and it is the most advanced. Students are expected to understand all of the basics of using RenderMan. In this course, we will explore more advanced details on how to define scenes, how to write truly excellent shaders, and how to integrate RenderMan into the studio production pipeline, which always includes a lot more tools than simply a renderer. Most importantly, we will explore topics that are beyond the scope of the only available textbook — *The RenderMan Companion*.

In 1988, Pixar developed and published the RenderMan Interface Specification with the goal that it would contain enough descriptive power to accommodate advances in modeling, animation and rendering technologies for years to come. It has done a remarkably good job, and even after 10 years, it is still the only open specification for scene description which includes concepts such as motion-blur and user-definable appearance characteristics. This means that RenderMan renderers will generate images of the highest quality, both in their simulation of objects in their environment and in their simulation of the camera.

The result of this strength, both the strength of the specification and the strength of the renderer which carries its name, is that studios throughout the world have made “RenderMan” their rendering platform of choice for doing work which demands images of the highest quality. Most visibly, major players in the motion picture industry have chosen RenderMan to generate 3D computer graphics special effects for films as diverse as *Jurassic Park*, *Mulan* and, of course, *Toy Story*. The striking results that these and other films have achieved through the use of RenderMan is why we are here today.

So how should a new TD learn to use RenderMan? How should an experienced TD keep up to date on the state-of-the-art tricks? How should writers of competitive renderers (yes, we know you’re out there!) learn what features they should be emulating in their software? There is only one published source. Written in 1990, *The RenderMan Companion* is still fabulous as a first course in using RenderMan, but it has at least two limitations. First, it is only a first course. There are a lot of topics of technical interest (and technical necessity) which are simply not covered, such as advanced shader antialiasing techniques. Second, it is 8 years old. The fact that it doesn’t discuss RIB is only the first of its anachronisms. While the things that it does say are generally still true (although not always), some of the most interesting features of the modern RenderMan renderers are simply not there.

Of course, the past 10 years has seen a lot of changes as well. The original dream of the RenderMan Interface was that it would be an open standard that all renderer vendors would embrace, giving the industry a family of compatible renderers that each excelled at different effects. This never occurred, and now the name RenderMan colloquially refers specifically to Pixar’s *PhotoRealistic RenderMan* product, and to the single widely-used compatible renderer *BMRT*. Freed from the shackles of abso-

lute specification conformance, new and very interesting features have been added to these RenderMan renderers that greatly increase its power and utility.

The industry has changed, as well. We've seen the rise of CGI from a curiosity used on a few unpopular movies to the dominant paradigm for generating special effects. Despite the naysayers and doubters that learned the wrong lesson from *TRON*, the first fully computer generated movie was a huge hit, and now there are at least 6 more in production. During this period of explosive growth, we've all learned a tremendous amount about how to create CGI, how to use CGI, how to run studios and how to write resumes. The tricks we thought were so clever in 1992 are high-school stuff nowadays. The beast needs new food, and that's really why we are here today.

So, welcome to *Advanced RenderMan: Beyond the Companion*. We've got several very interesting speakers who will teach you a little of what they know about making beautiful images, and hopefully inspire you to make even more beautiful ones.

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

Modern RenderMan Interface Features

Tony Apodaca
Pixar Animation Studios

New RenderMan Interface Features

Modern versions of *PhotoRealistic RenderMan* and *BMRT* have various extensions, which add large and small features that were not provided for in the original RenderMan Interface Specification. In addition, the newest versions of these renderers implement features which have always been in the Spec, but are probably not widely known because they were never available in implementations. This section of this talk will discuss some of the more important extensions to RenderMan that are found in modern renderers.

Including RIB Archives

The RenderMan Interface describes a subroutine to write arbitrary user data into a RIB “archive file”, `RiArchiveRecord`. Appendices even went so far as to describe a recommended syntax for placing particular objects into their own individual RIB archive files, for later assembly by some “Render Manager” program. However, it never described any way for the renderer read such a file (actually, this was intentional, but the weight of public opinion was that this was a poor decision). Therefore, a new call has been added to read an existing RIB archive into the renderer at an arbitrary point in the RI command stream.

```
RiReadArchive ( RtToken filename, RtFunc callbackfunc, parameterlist );  
ReadArchive filename
```

will read the named filename. Each RIB command in the archive will be parsed and executed exactly as if it had been called by the application program directly, or been in-line in the calling RIB file, as the case may be. This is essentially a RIB-file *include* mechanism.

In the C version, the second parameter is a callback function which will be called for any RIB user data record or structure comment which is found in the file. This routine has the same prototype as `RiArchiveRecord`, and allows the application routine to notice user data records and then execute special behavior based on them as the file is being read into the program.

Restore a Coordinate System

The RenderMan Interface Specification allows the user to set the current coordinate system to any particular value if you know the appropriate transformation matrix (`RiTransform`), and allows the user to name coordinate systems for future use in shaders (`RiCoordinateSystem`), but has always lacked the ability to use those named coordinate systems later in the same RIB file. A new call has been added to set the current transformation matrix to be that of a previous named coordinate system.

```
RiCoordSysTransform ( RtString coordinatesystem );  
CoordSysTransform coordinatesystem
```

will set the current transformation matrix to be the matrix for `coordinatesystem`. `coordinatesystem` can be the name of a user defined coordinate system named using the `RiCoordinateSystem` call, or it can be one of the predefined coordinate systems: "raster", "NDC", "screen", "camera", "world", "object". Note that:

- "raster" is resolution dependent.
- "raster", "NDC" and "screen" use z values that vary between 0.0 at the near clipping plane to 1.0 at the far clipping plane.
- "object" is the coordinate system in current use anyway.

Declaring Parameterlist Variables

The RenderMan Interface Specification identifies four valid *types* of parameterlist variables, `float`, `point`, `color` and `string`, which behave in the obvious way and have direct analogues in the Shading Language.

Modern parsers accept several other types, which have been found to be useful over the past few years. The list of valid declaration types now includes `vector`, `normal` and `hpoint`. These data types represent direction vectors, normal vectors and homogeneous points, respectively. As with geometric parameters of type `point`, parameters of these types are specified in object space and are transformed by the current transformation matrix. However, just as mathematical points, homogeneous points, direction vectors and normal vectors each transform slightly differently, so the declared type indicates which version of the matrix transformation should be used.

The list of valid declaration types has also been extended to include a new `matrix` type, which obviously describes a transformation matrix. As with points, the matrix is described relative to object space.

`point`, `vector` and `normal` parameters each contain three floating point numbers per entry, whereas homogeneous `hpoint` parameters contain four floating point numbers per entry, and `matrix` parameters contain 16 float values per entry. The predefined position variable "Pw" can now be described as a vertex `hpoint`, and the predefined variable "N" as a `varying normal`. As we will see below, the `vector`, `normal` and `matrix` types correspond to new datatypes in the Shading Language. `hpoint`, on the other hand, does not correspond to a new Shading Language type. Variables of type `hpoint` will undergo a homogeneous divide in the renderer, and appear in shaders as `normal points`.

The RenderMan Interface Specification also identifies valid *storage classes*, which specify the number of entries that occur on primitives of various types. The standard storage classes are `uniform` and `varying`. By way of review, `uniform` variables occur once "per facet", and `varying` variables occur once per "parametric corner". The actual number of facets and parametric corners that a particular primitive has is dependent on the primitive, and is often a somewhat complex formula.

The modern RenderMan renderers accept two other storage classes, in order to provide more flexibility in the modeling task. The first new storage class is `constant`, which specifies that variables occur exactly once per primitive. This is different from `uniform` on polyhedra and meshes which have multiple facets per primitive.

The second new storage class is `vertex`. This class specifies that variables occur once per control vertex (that is, exactly as often as "P" occurs). This class is mentioned in the Spec, where it is used in the RIB binding to define certain built-in variables such as "P". Now it is officially sanctioned for use by arbitrary user variables.

It is important for programmers to note that while there are now four storage classes defined by RIB, there are still only two storage classes in the Shading Language. RIB `constant` and RIB `uniform` parameters will both appear to the Shading Language as `uniform` variables. The only difference is that RIB `constant` variables are the same everywhere on a primitive, whereas RIB `uniform` parameters are the same everywhere on a facet, but may be different on different facets.

Similarly, RIB `varying` and RIB `vertex` parameters both appear to the Shading Language as `varying` variables. The difference is that `varying` parameters are bilinearly interpolated in parametric space of each primitive. `vertex` parameters are interpolated using the same basis functions as the position parameter of the primitive. They will literally be interpolated with the vertices.

Multisegment Motion-Blur

Previous versions of *PhotoRealistic RenderMan* handled motion-blur by linearly interpolating the position of primitives from their position at the shutter open to their position at the shutter close. In situations where the object is rotating rapidly, this approximation is often obviously incorrect (for example, spinning tires and flapping wings).

Newer versions of *PhotoRealistic RenderMan* break this limitation by permitting a piecewise linear interpolation of positions within a single frame. The RIB file syntax is exactly as the original RenderMan Interface Specification described. Each motion-block specifies a sequence of times, and there is one version of the transformation (or object specification) that corresponds to each time value. The shutter specifies the range of these times that will appear in the frame, and the motion is piecewise linearly interpolated between the knots at the given times.

For example:

```
Shutter [0.0 1.0]
...
MotionBegin [0.0 0.25 0.75 1.0]
Translate 0.0 10.0 0.0
Translate 10.0 0.0 0.0
Translate 0.0 0.0 20.0
Translate 0.0 0.0 0.0
MotionEnd
Sphere 1 -1 1 360
```

describes a ball which is moving around violently in 3-D.

As with previous versions of motion-blur, transformations and object deformations are both legal inside the motion block.

The shutter open and close times are not required to match any of the specified time values, and if they do not, the objects and transformations will be correctly clipped to the shutter's range. For example, the shutter in the above example could run from 0.1 to 0.9, and the "right thing" would happen. If the shutter extends outside the range specified in the motion block, two things happen.

Transformations are clamped to their endpoints. In the example above, the ball is considered stationary at (0,10,0) prior to time 0.0, and stationary again at (0,0,0) after time 1.0. Geometry, on the other hand, is non-existent outside of the motion-block's time range. Therefore, one may have a geometry appear in the middle of a frame, or similarly vanish intraframe.

Rotations are not divided into segments automatically. That is, spinning tires will not simply start to work better. Rotations that need more temporal resolution must be specified by the model (or the RIB generation program) as follows:

```
Shutter [0.0 1.0]
...
MotionBegin [0.0 0.25 0.50 0.75 1.0]
Rotate 0.0 0.0 1.0 0.0
Rotate 10.0 0.0 1.0 0.0
Rotate 20.0 0.0 1.0 0.0
Rotate 30.0 0.0 1.0 0.0
Rotate 40.0 0.0 1.0 0.0
MotionEnd
```

In the current implementation of *PhotoRealistic RenderMan*, shading of each segment of the object's motion is calculated independently, at the time value which represents the beginning of that segment. However, shading parameters can not yet be interpolated through time. Therefore, any shading differences which are due only to position will be visible on the various segments of the motion path (for example, specular highlight location). Be aware that this will probably interact poorly with shadow generation, since it is not yet possible to specify a separate shadowmap at each individual shading time.

Objects

In the RenderMan Interface Specification, objects (as defined by `RiObjectBegin/End`) have always been extremely limited in that they could only be a flat, attributeless list of geometric primitives of the same type. These restrictions has made objects essentially useless (*PhotoRealistic RenderMan* has never enforced the “of the same type” restriction, but this doesn't really help).

In recent versions of *PhotoRealistic RenderMan* objects have been significantly enhanced. While not all restrictions are being removed, the new objects should have significantly more utility.

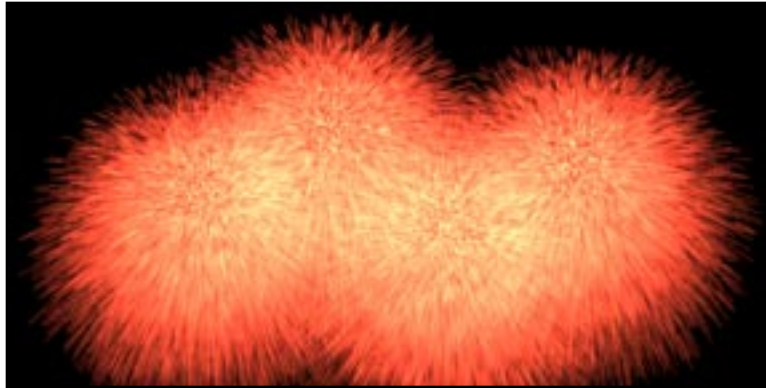
- The restriction that objects cannot have a transformation hierarchy is lifted. `RiTransformBegin/End` and all of the transformation routines (`RiTranslate`, `RiRotate`, etc.) may be used inside an Object block.
- The restriction that objects cannot have motion-blur is lifted. `RiMotionBegin/End` can be used in its full generality to generate both transformation blur and deformation blur of primitives in an object.
- Moving objects can be instanced into moving coordinate systems, and the “right thing happens” even if the motion sample times don't match.
- The renderer will accept `RiBasis` calls inside an object.
- Objects still may not contain other attributes (e.g., shading attributes).
- Objects still may not be instanced inside a Motion block

New RenderMan Interface Primitives

The new versions of *PhotoRealistic RenderMan* have several new geometric primitives which are specifically designed to solve particular problems in modeling scenes for feature film special effects and animation.

Point Clouds

The new **RiPoints** primitive is intended to be used for small particles such as sparks, snowflakes, or raindrops. They are optimized for the case where the intended particle is small, taking up at most a couple of pixels on the screen. Points are nearly ideal for generating star fields and fireworks.



`Points` are much cheaper in terms of memory and in terms of shading time than small bilinear patches or small spheres. Each point is only shaded once, and generates only two micropolygons. You can efficiently render very large point clouds using the point primitive.

There are some disadvantages of this light-weight primitive, however. Since points are only shaded at a single point, they have no access to derivative information. If you want your particle to have differing shading across its surface, or if the particle takes up multiple pixels, you might want to use bilinear patches or spheres. Points may be motion blurred, however one must be aware that when a very small object is motion-blurred, the process of stochastic sampling tends to make individual objects vanish into a sea of dim dots.

The syntax for `RiPoints` is:

```
RiPoints ( RtInt npoints, parameterlist );
Points "P"  particle-locations parameterlist
```

The width of the points is defined in the current object coordinate system, and defaults to 1.0. You can specify either a single width value for all points using the "constantwidth" parameter, which is defined as a uniform float, or give individual points separate widths by using the varying float variable "width". Notice that the rendered size of points is *not* specified in raster coordinates. This choice was made because `Points` were intended to be a rendering primitive, not an annotation primitive. If widths were specified in raster space, they would be the same size independent of distance from the camera, and independent of resolution. This would look very wrong in animation, particular in situations where the points are moving rapidly past the camera.

Points will often be used to create fire or explosion effects. For example, consider the original particle systems of the *Star Trek Genesis Sequence* done by Bill Reeves in 1984. In those renderings, particle opacities were set to 0.0, so that in each pixel, by virtue of the compositing algebra, the intensity of all of the particles in the pixel would add rather than having the front particles obscure underlying ones. At the center of an explosion where many particles overlap this gives the illusion of a glowing hot center.

In situations where point primitives are not appropriate, such as when stochastic sampled motion-blur doesn't give the correct look, but enormously large quantities of particles are still necessary, special-purpose particle rendering programs may be required.

Curves

PhotoRealistic RenderMan has an `RiCurves` primitive, which provides a mechanism for specifying 3-D curves. One may think of this primitive as a ribbon or a string of spaghetti. It is efficient to render large numbers of these primitives, so it is ideally suited to modeling such things as hair and grass.

`RiCurves` are specified as follows:

```
RiCurves( RtToken type, RtInt ncurves, RtInt nvertices[],
           RtToken wrap, parameterlist );
Curves type nvertices wrap parameterlist
```

Each `Curves` primitive specifies one or more 3-D ribbons of specified width through a set of control vertices. Notice that multiple disconnected individual curves may be specified using one call to `RiCurves`. The format of the `nvertices` array and `parameterlist` data is very similar to that of `RiGeneralPolygon`, in that `nvertices` is an array of integers specifying the number of vertices in each of the individual curves. Curves can either be "linear" or "cubic" (notice, *not* "bilinear" or "bicubic"), and can wrap around, like a `PatchMesh`. Cubic curves interpolate using the v basis matrix and step size set by `RiBasis`. In either case, the u parameter changes across the width of the curve, while the v parameter changes across the length of the curve (i.e. the direction specified by the control vertices).

The width along the curve may be specified with either a "width" parameter which is a varying float argument, or a "constantwidth" parameter which is a constant float (one value for the entire `RiCurves`). Widths are specified in object space units of the curve. If no "width" vector or "constantwidth" value is given, the default width is 1.0 units in object space.

Each curve generates a flat ribbon, but the control vertices only specify the direction of the "spine", the rotation of the flat ribbon about the spine is ambiguous. In the standard case, the ribbon will always rotate so that it is as parallel to the view plane as possible. In other words, it will twist to face the camera. This is a good way to simulate a thin tube, since the silhouette of the ribbon will match that of the tube.

However, if "N" values are supplied, the normals will be used to guide the ribbon so that it stays perpendicular to the supplied normals, thus allowing user-controlled rotation of the ribbon. To summarize, if you supply "N" values, you will get something like grass, and if you do not supply "N" values, you will get something like hair.

The number of data items required for uniform and varying parameters for curves are reminiscent of those for patch meshes.

- constant: one single value for the entire `RiCurves` primitive.
- uniform: $\text{sum}(n\text{segs}_i)$ values
- varying: $\text{sum}(n\text{segs}_i+1)$ values for nonperiodic curves, $\text{sum}(n\text{segs}_i)$ values for periodic curves.
- vertex: $\text{sum}(n\text{vertices}[i])$

where $\text{sum}()$ indicates summing over all the individual curves specified by the `RiCurves` statement, and:

$nsegs_i$ is the number of segments in curve # i
 = $nvertices[i]-1$ for linear, nonperiodic curves
 = $nvertices[i]$ for linear, periodic curves
 = $(nvertices[i]-4)/vstep+1$ for cubic, nonperiodic curves
 = $nvertices[i]/vstep$ for cubic, periodic curves

Using RiCurves for Hair

PhotoRealistic RenderMan is able to render large numbers (hundreds of thousands or more) of curves very efficiently, making this primitive particularly well suited for modeling hair. Since the default behavior (when normals are not supplied) is to generate ribbons that keep their flat side facing the camera, we are creating geometry which has the same silhouette as a hair, but is perhaps hundreds of times more efficient to render (in both time and memory) than an equivalent surface of circular cross section.

But the hairs are just ribbons, not really round tubes. If we light them using a standard illumination model (like plastic.sl), they will not look like hair. Flat ribbons don't reflect light the same way that a tube does. But we can make the illusion complete by writing a shader for use with the curves which reacts to light *as if* it was a thin cylinder.

The Siggraph literature has several descriptions of nonisotropic illumination functions that simulate extremely thin cylinders. Here is such a shader, with comments that should be self-explanatory:

```
surface hair (float Ka = 1, Kd = .6, Ks = .35, roughness = .15;
  color rootcolor = color (.109, .037, .007);
  color tipcolor = color (.519, .325, .125);
  color specularcolor = (color(1) + tipcolor) / 2; )
{
  vector T = normalize (dPdv); /* tangent along length of hair */
  vector V = -normalize(I); /* V is the view vector */
  color Cspec = 0, Cdiff = 0; /* collect specular & diffuse light */
  float cosang;
  /* Loop over lights, catch highlights as if this was a thin cylinder */
  illuminance (P) {
    cosang = cos (abs (acos (T.normalize(L)) - acos (-T.V)));
    Cspec += Cl * v * pow (cosang, 1/roughness);
    Cdiff += Cl * v;
    /* We multiplied by v to make it darker at the roots. This
     * assumes v=0 at the root, v=1 at the tip.
     */
  }
  Oi = Os;
  Ci = Oi * (mix(rootcolor, tipcolor, v) * (Ka*ambient() + Kd*Cdiff) +
    (Ks * Cspec * specularcolor));
}
```

Here is an example showing what RiCurve primitives look like, in combination with the shader above:



New NURB Parameterization

The RenderMan Interface Specification explicitly details, for each type of primitive, the number of vertex, varying and uniform parameters which occur on that primitive. In general, there are four varying and one uniform variables on any parametric (non-polygon) primitives, with the exception of `RiPatchMesh`, which has a number of varying and uniform variables based on the number of subpatches in the mesh.

However, the specification is silent on the matter of `RiNuPatch`, and in early versions of *PhotoRealistic RenderMan* (prior to version 3.6), the default four/one scheme was used for lack of more explicit direction, in a manner parallel to that of a single B-spline patch.

It turns out that this was a poor choice, and modern versions of *PhotoRealistic RenderMan* make a better choice. The number of varying and uniform variables on an `NuPatch` is now officially computed as if the `NuPatch` is a nonperiodic uniform B-spline *mesh*, rather than a single B-spline patch. The method of computation is as follows:

- A `NuPatch` is defined to have $(1+nu-uorder)$ segments in the u parametric direction,
- and $(1+nv-vorder)$ segments in the v parametric direction.
- A `NuPatch` is defined to have one uniform value per segment,
- and one varying value per segment corner.
- The number of uniform variables is therefore $nusegments*nvsegments$,
- and the number of varying variables is therefore $(nusegments+1)*(nvsegments+1)$.

Users with significant experience with NURBs will probably notice that this results in redundant parameter values corresponding to repeated knot values, for instance when the knot vector indicates

the NuPatch is in Bezier form, however the benefit of the flexibility far outweighs any burden due to the redundancy.

Subdivision Mesh

Perhaps the most interesting new primitive which has been added to the RenderMan family is **subdivision mesh** (a.k.a. subdivision surface). The great motivation for using subdivision surfaces is that it simultaneously captures the most desirable properties of several of the different surface types that we currently use. A subdivision surface, as with parametric surfaces, is described by its control mesh of points. The surface itself can approximate or interpolate this control mesh, while being piecewise smooth. However, unlike NURB or patch mesh surfaces, its control mesh is not confined to be rectangular, a major limitation of bi-parametric patches. It can have arbitrary topology, and arbitrary vertex valence. In this respect, the control mesh is analogous to a polyhedral description. No longer do you have to painstakingly model a smooth surface as a quilt of rectangular patches.

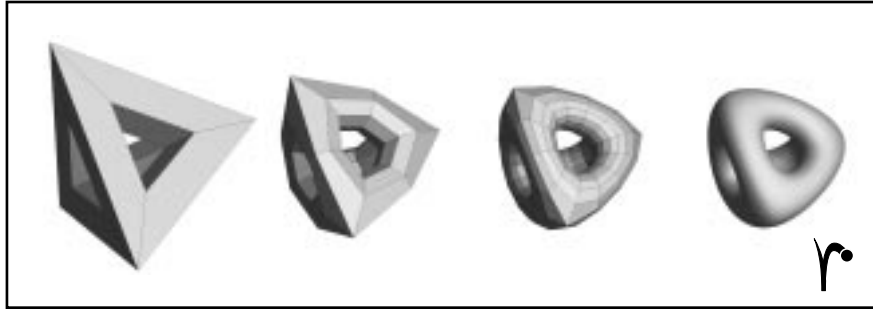
But where a polyhedral surface requires a large number of data points to approximate a smooth surface, a subdivision mesh is always smooth, and needs far fewer points to produce the same quality of smooth surface fit. This significantly cuts down the time necessary to create the model, the amount of data necessary to hold the model, and gives the model significantly more flexibility in terms of the range of model distances and image resolutions at which the model will look good. There is no point at which the rendering of the model belies its underlying polygonal digitization.

For animation, the subdivision surface also shares another important feature with polyhedra. Adding detail is a local operation. There is no need to add large “isoparams” that extend far beyond the area of interest. And because there are no topology restrictions, edges can follow character features (such as skin folds) more directly. This leads to better isolation of one part of the mesh from another, gives direct muscle control over lines which traverse the surface in arbitrary directions, and hence leads to fewer problems with unintentional muscle effects on distant parts of the surface.

Another advantage of the flexible topology is that there has been some pretty nifty work already done on *automatic* surface reconstruction from digitized points. In addition, there have been several Siggraph papers published on multiresolution and wavelet versions of subdivision meshes, which imply that automatic level-of-detail model generation is not out of the question.

Subdivision surfaces were first discussed in the computer graphics literature in 1978, in a paper by Catmull and Clark. However, they were generally ignored until the mid-1990s, when a group of researchers at University of Washington started developing a mathematical formalism for the surfaces, and started demonstrating the advantages they simultaneously had over the more popular NURB and polyhedral models. An important feature was added to subdivision surface description were to make them more generally useful as a modeling primitive: the ability to specify creases along a surface.

The RI interface for subdivision meshes is extremely powerful, and permits the specification of semisharp creases, holes, and other enhancements which are not described in previous literature. This gives subdivision meshes the ability to model fillets, rounds and blends, and to do other modeling tasks often associated with trim curves. The paper in this year’s Siggraph Technical Session, *Subdivision Surfaces for Computer Animation*, by DeRose, Kass and Truong, describe the mathematics behind, and use of, these features.



Subdivision meshes are described by the following interface:

```
RtSubdivisionMesh ( RtToken scheme; RtInt nfaces; RtInt nvertices[];
                    RtInt vertices[]; RtInt ntags; RtToken tags[];
                    RtInt nargs[]; RtInt intargs[]; RtFloat floatargs[];
                    parameterlist )
SubdivisionMesh scheme nvertices vertices tags
                    nargs intargs floatargs parameterlist
```

The token `scheme` specifies the mathematical subdivision scheme to be used by the primitive. There are several such schemes defined in literature, but currently the renderer only implements "cat-mull-clark", specifying the Catmull-Clark subdivision method. The subdivision mesh itself is made up of faces, very similar to those in `RtPointsPolygons`, with an `nvertices` array which contains the number of vertices in each face, a `vertices` array containing, for each vertex, an index into the vertex primitive variable arrays.

A *component* of a subdivision mesh is either a face, a vertex, or a chain of edges. Components of the subdivision mesh may be *tagged* to have various user-defined properties. Each tag has an `RtToken` identifier, and zero or more integer arguments and zero or more floating-point arguments, specified in the `nargs` array. Several tags are currently defined, and more may be added over time:

- The *sharpness* of a vertex or chain of edges is a floating-point number, ranging from completely smooth to C1-discontinuous.
- The "hole" tag specifies that certain faces are holes. This tag has n integer arguments, one for each face which is a hole, and zero floating-point arguments.
- The "corner" tag specifies that certain vertices are sharp corners. This tag has n integer arguments, one for each vertex which is a sharp corner, and n floating-point arguments, specifying a sharpness at each such corner.
- The "crease" tag specifies that a certain chain of edges should be a crease. This tag has n integer arguments, specifying the number of vertices which make up the chain of edges, and one floating-point argument, specifying a sharpness for the edges in the crease. There may be multiple "crease" tags on a subdivision mesh which form independent creases on the primitive. It is an error to specify two sequential vertices in an edge chain which do not form an edge of one or more of the mesh faces.
- The "interpolateboundary" tag specifies that the subdivision mesh should interpolate all boundary faces to their edges. This tag has zero integer arguments and zero floating-point arguments.

The parameterlist must include at least position ("P") information. If a primitive variable in the parameterlist is *varying*, there should be one value of the associated type per vertex, just as with polygon primitives. If the variable is *uniform*, there should be one value of the associated type per face.

Procedural Primitives

RenderMan procedural primitives are user-provided subroutines which manipulate private data structures containing geometric primitives that the renderer knows nothing about. There are two entry points for each procedural primitive, the *subdivide* method, and the *free* method. Each instance of a procedural primitive contains pointers to the appropriate methods, a blind (to the renderer) data pointer which points at data allocated by (and meaningful to) the methods, and a bounding box which completely contains any geometry of the primitive.

When the renderer reaches the bounding box of a particular procedural primitive instance, it calls the subdivide method, passing in the blind data pointer and a floating point number which indicates the number of pixels which the bounding box covers (called the *detail*). The subdivide method splits the primitive into smaller primitives. It can either generate standard RenderMan primitives, or it can generate more instances of procedural primitives with their own blind data pointers and (presumably smaller) bounding boxes, or some combination.

At some point, the renderer knows that it will not need a particular blind pointer ever again. It then calls the free routine to destroy that blind pointer.

The RenderMan Interface Specification provides for linking a renderer into a modeling program, and thus providing the two required subroutine pointers directly. In addition, *PhotoRealistic RenderMan* provides three built-in procedural primitives which can be used from the stand-alone renderer executable via simple RIB directives.

Delayed Read Archive

The simplest of the new RIB procedural primitives is *Delayed Read Archive*. The existing interface for "including" one RIB file into another is `RiReadArchive`. *Delayed Read Archive* operates exactly like `RiReadArchive`, except that the reading is delayed until the procedural primitive bounding box is reached, unlike `RiReadArchive` which reads RIB files immediately during parsing. The advantage of the new interface is that since the reading is delayed, memory for the read primitives is not used until the bounding box is actually reached. In addition, if the bounding box proves to be off-screen, the parsing time of the entire RIB file is saved. The disadvantage is that an accurate bounding box for the contents of the RIB file is required, which was never needed before.

In RIB, the syntax for the *Delayed Read Archive* primitive is:

```
Procedural "DelayedReadArchive" [ "filename" ] [ bound ]
```

As with all RIB parameters which are bounding boxes, the bound is an array of six floating point numbers which is *xmin, xmax, ymin, ymax, zmin, zmax* in the current object space.

Run Program

A more dynamic method of generating procedural primitives is to call a helper program which generates geometry on-the-fly in response to procedural primitive requests in the RIB stream. As will be seen below, each generated procedural primitive is described by a request to the helper program, in the form of an ASCII datablock which describes the primitive to be generated. This datablock can be anything which is meaningful and adequate to the helper program, such as a sequence of a few

floating point numbers, a filename, or a snippet of code in a interpreted modeling language. In addition, as above, the renderer supplies the *detail* of the primitive's bounding box, so that the generating program can make decisions on what to generate based on how large the object will appear on-screen.

The generation program reads requests on its standard input stream, and emits RIB streams on its standard output stream. These RIB streams are read into the renderer as though they were read from a file (as with `ReadArchive` above), and may include any standard RenderMan attributes and primitives (including procedural primitive calls to itself or other helper programs). As long as any procedural primitives exist in the rendering database which require the identical helper program for processing, the socket connection to the program will remain open. This means that the program should be written with a loop which accepts any number of requests and generates a RIB "snippet" for each one.

In RIB, the syntax for specifying a RIB-generating program procedural primitive is:

```
Procedural "RunProgram" [ "program" "datablock" ] [ bound ]
```

`program` is the name of the helper program to execute, and may include command line options. `datablock` is the generation request data block. It is an ASCII string which is meaningful to `program`, and adequately describes the children which are to be generated. Notice that `program` is a quoted string in the RIB file, so if it contains quote marks or other special characters, these must be escaped in the standard way. The `bound` is an array of six floating point numbers which is `xmin`, `xmax`, `ymin`, `ymax`, `zmin`, `zmax` in the current object space.

Procedural Primitive DSO

A more efficient method for accessing subdivision routines is to write them as dynamic shared objects (DSOs), and *Dynamic Load* them into the renderer executable at run-time. In this case, you write your subdivision and free routines exactly as you would if you were writing them to be statically linked with the renderer. They are compiled with special compiler options to make them run-time loadable, and you specify the name of the shared `.so` file in the RIB file. The renderer loads the DSO the first time it is needed to subdivide a primitive, and from then on, it is called as if (and executes as fast as if) it were statically linked.

When writing a procedural primitive DSO, you must create three specific public subroutine entry points, named `Subdivide`, `Free` and `ConvertParameters`. `Subdivide` is a standard RI procedural primitive subdivision routine, taking a blind data pointer to be subdivided, and a floating-point detail to estimate screen size. `Free` is a standard RI procedural primitive free routine, taking a blind data pointer which is to be released. `ConvertParameters` is a special routine which takes a string and returns a blind data pointer. It will be called for each Dynamic Load procedural primitive in the RIB file, and its job is to convert a printable string version of the progenator's blind data (which must be in ASCII in the RIB file), into something that the `Subdivide` routine will accept.

In RIB, the syntax for specifying a dynamically loadable procedural primitive is:

```
Procedural "DynamicLoad" [ "dsoname" "initial" ] [ bound ]
```

`dsoname` is the name of the `.so` file which contains the three required entry-points, and has been compiled and prelinked as described above. `initial` is the ASCII printable string which represents the initial data to be sent to the `ConvertParameters` routine. The `bound` is an array of six floating point numbers which is `xmin`, `xmax`, `ymin`, `ymax`, `zmin`, `zmax` in the current object space.

Note that when the `Subdivide` routine wants to create a child procedural primitives of the same type as itself, it should call `RiProcedural (data, bound, Subdivide, Free)`, *not* call for itself as a `DynamicLoad` procedural.

New RenderMan Shading Language Features

Since the publication of the RenderMan Interface 3.1 Specification in 1990, various extensions have been made to the Shading Language. Each of these have been implemented in Pixar's *PhotoRealistic RenderMan*, and many have been simultaneously implemented in *BMRT*. This section of the talk will highlight the Shading Language extensions in *PhotoRealistic RenderMan* which are not covered in the RenderMan 3.1 Specification.

Enhanced Shading Language Syntax

The most obvious, and in certain cases not backward-compatible, change to the Shading Language is that we have modernized the syntax and added certain syntactical structures which make it easier to write shaders without unintentional errors. The latest compilers take advantage of these new syntactical structures to provide some *lint*-like functionality.

Scoping of Variables and User Functions

The RenderMan Interface Specification describes a Shading Language which is very much like K&R C, when it comes to issues such as variable declarations, function declarations and name scoping. Now ten years later, the Shading Language syntax and semantics have been modified in minor but useful ways, taking "good ideas" from both C++ and Pascal (yes, there are a few...) to enhance the programmer's model.

As in C++, you can now declare local variables anyplace where a statement is valid. You may also generate a new scope anyplace that a statement is valid, using braces `{ }`. Variables are accessible in the scope in which they are declared (or any nested inner scopes) after the declaration appears. Variables are not visible outside their scopes, nor prior to the line on which they are declared.

One of the biggest limitations of the original compiler was that it had trouble compiling functions in Shading Language. There was a funny issue about separate compilation, there were severe restrictions on what you could do inside functions (e.g. you could not call `texture()`), and sometimes the compiler just compiled functions incorrectly.

The new Shading Language compiler does not support separate compilation of functions. Instead, you must declare your functions either in the same source file as your shader, or put the function definition in a header file and use `#include` to tell the compiler to include the source for the function. The new compiler allows any syntactically legal constructs to be used inside functions, though it may issue an error if you then try to use the function from within a shader where such a construct is not allowed.

You may now define a function local to any scope, anyplace that a statement is valid. The function may be called within its scope (or inside any nested scopes) after it is defined. It may not be called from outer scopes, or before (in the textual sense) it is defined. Functions can also have their own locally defined functions, private to their scope. You may recognize these rules as being somewhat reminiscent of the way Pascal allows locally defined functions.

Here is an example of a function which is local to a shader:

```

surface mysurf (float Kd = 0.5;) /* parameter */
{
    color shadernoise (point p) {
        return color noise (transform ("shader", p));
    }
    float q; /* local variable of the surface shader */
    q = 1;
    Ci = shadernoise(P) * diffuse(faceforward(normalize(N),I));
}

```

Functions may still be declared outside of shaders, just as they always were before. Such functions may not access any variables other than their parameters and locals.

New declaration syntax

According to the Specification, function parameters are passed by reference, and a function can overwrite any of its parameters, thus changing its value. In the new syntax, it is illegal to write to a function parameter, unless it is declared using the **output** keyword. For example, consider the following Shading Language code:

```

float test (float foo, output float bar) {
    foo = 1;    /* Illegal! Cannot write to parameter. */
    bar = 2;    /* OK! */
}

```

This same rule applies to shader instance variables. All shader instance variables are now considered “read-only” unless they are explicitly declared with the output keyword.

```

surface test ( float foo = 0; output float bar = 1 ) {
    foo = 5;    /* Illegal! Cannot write to an instance variable. */
    bar = 6;    /* OK! */
}

```

The only variables visible to a function are its parameters and local variables, i.e. it may not access variables of outer scopes, unless you declare the variable as a local using the **extern** keyword.

Here’s an example:

```

surface mysurf (float Kd = 0.5;) /* parameter */ {
    float q; /* local variable of the surface shader */
    color foo (color C) {
        extern float Kd; /* access outer scope: param to function */
        extern point P;  /* access outer scope: global P */
        extern float q;  /* access outer scope: local variable q */
        return Kd * q * C * color noise (P);
    }
    q = 1;
    Ci = foo (Cs) * diffuse(faceforward(normalize(N),I));
}

```

In the preceding example, local function `foo` declared `extern` variables `Kd`, `P`, and `q`, as a way of accessing shader parameters, graphics state variables, and shader local variables, respectively. If `foo()`

had tried to access any of these without declaring them using the `extern` keyword, the compiler would have reported an “unknown variable” error.

New Variable Types

The RenderMan Interface Specification Version 3.1 specifies that the Shading Language has exactly four data types: `float`, `point`, `color`, and `string`. This set has been extended to include two variants on point types, vector and normal, a 4x4 matrix type, and arrays.

New point-like types

Previously, there was only one `point` type which was used to represent three different geometric concepts: (1) spatial positions, (2) directions, and (3) surface normals. However, these three geometric entities represent different concepts and have subtly different usage properties. Now there are three separate types corresponding to these concepts: `point`, `vector`, and `normal`, respectively.

Variables declared as `vector` and `normal` behave similarly to variables declared as `point` in most situations. `point`, `vector` and `normal` variables *can* be arbitrarily assigned from one to another and can be used identically in expressions and function arguments, however mixing these types in ways which are not geometrically appropriate (for example, taking the cross product of two points) will generate a compiler warning.

Keeping these types separate will help to curtail common errors which result in buggy shaders. Note in particular that the transformation functions (described below) are specifically designed to work on particular types, and using them on incorrect types will result in incorrect transformations.

When a shader parameter is declared with **Declare** in a RIB file (or with **RiDeclare** in an RI program) to be a vector or normal, any value provided for that variable in any RI call will be transformed into current space using the appropriate transformation function. Note that the declaration of the variable in the Shading Language code should match the declaration in the RI stream. If the two declarations do not match, the renderer will issue a warning (and may not transform the data correctly).

When a triple of floating point values is assigned to a `point`, `vector` or `normal` variable using the type cast operator, an optional coordinate system name can be specified, such as

```
foo = point "object" (0, 0, 0);
bar = vector "world" (s, t, u);
baz = normalize( normal "world" (xcomp(N), 0.5, zcomp(N)) );
```

These operators now transform the triples of floats from the named coordinate system to the current coordinate system using the appropriate transformation function.

The matrix type

A `matrix` represents the transformation matrix required to transform points and vectors between one coordinate system and another. `matrix` is a first-class data type in the language, so it can be used to declare shader instance parameters, local variables and function parameters. Beware if you declare `matrix` of type `varying`. That’s going to be a lot of data!

A `matrix` is, internally, a 4x4 homogeneous premultiplication transformation matrix. However, our intent was to maintain the programmer’s model that the `matrix` is an atomic data type, not a 2-dimensional array. Individual rows, columns or elements will not be accessible through any form of array notation.

The syntax of matrix specification may be subtly confusing the first time it is seen, but it is consistent with the semantics of point specification. A `matrix` constant can be specified like this:

```
matrix (a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p)
matrix space 1
matrix space (a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p)
```

The first form is the easiest to understand. It generates a standard transformation matrix which transforms points (or vectors). For example, this could be a rotation matrix, which spins points around the origin of whatever coordinate system they happen to be in.

The second form creates the matrix which transforms points from the “current” coordinate system to the *space* coordinate system. Transforming points by this matrix is similar to calling `transform(“current”, space, ...)`.

The last form combines the two. It preconcatenates the “current”-to-*space* transformation matrix onto the specified 4x4 matrix.

`matrix` variables can be tested for equality and inequality with the `==` and `!=` boolean operators. Various new and old functions will accept `matrix` variables as arguments. In addition, the values 0 and 1, when promoted to type `matrix`, generate the obvious zero matrix (all elements zero), and identity matrix (elements on the diagonal one, the rest zero).

When used as a shader instance parameter, `matrix` is used in cooperation with the matching RIB declaration type `matrix` to pass 4x4 transformation matrices into the shader. As with the `point` data type, a `matrix` value in the RIB stream represents a quantity relative to the then-current object space, while a `matrix` in the Shading Language represents a quantity relative to the renderer’s “current” space. Thus, there will be an implicit transformation between the two representations.

There are several operators and functions which operate on `matrix` variables. We tried to provide a complete and orthogonal set. If there are “obvious” `matrix` functions that are missing, or necessary in your work, we are open to all suggestions.

```
matrix * matrix
1.0 / matrix2
matrix1 / matrix2
float determinant( matrix m )
```

These functions calculate the standard matrix functions of matrix multiplication, matrix inverse, multiplying a matrix by the inverse of another, and calculating the determinant of a matrix.

```
point transform( matrix m, point p )
point transform( string fromspace, matrix m, point p )
```

These functions transform points through the matrix. The second form transforms points from *fromspace* space, first to “current” space, then through the matrix. Naturally, there are similar `vtransform` and `ntransform` functions.

Arrays

Shading Language now supports arrays of all the basic data types. Because of the nature of the language, as well as realistic implementation constraints, we were not at liberty to create a full C-like semantics for arrays, which are based heavily on pointer arithmetic. Instead, we chose a more limited

but usefully-expressive simple array semantics, similar to that of strongly-typed languages. Shading Language arrays have the following features:

- arrays can be of any basic data type, and may be either uniform or varying;
- all arrays are one-dimensional, zero-based and of fixed, predeclared (compile-time) lengths; except that
- arrays can be of indeterminate length for function formal parameter declarations;
- there are no pointers, and no array subrange notation;
- strict array over/underrun checking is done at run-time.

As in C, the syntax for declaring an array of any data type uses square brackets, as *datatype*[*length*]. *length* is a compile-time uniform float constant, which is rounded down to generate an integer length. Zero and negative-length arrays are not permitted. Also as in C, the syntax for specifying the data of a constant array uses curly braces, as {*value1*, *value2*, ...}.

As in C, individual array elements may be referenced using the familiar square bracket notation. An array element index is a uniform float expression, which is rounded down to generate an integer index. An array element reference may be used in any expression where a variable of the same base type is legal. An array element reference on the left side of an assignment statement is an *lvalue*, and sets that single element to the result of the right side expression.

You may not assign or compare entire arrays (just like you cannot in C). However, like C, you may initialize an array with constant values. For example, the following declares a shader which takes an array as a parameter, and specifies defaults for the argument:

```
surface mysurf (float Kd = 0.5; ) {
    float vals[5] = { 0, 2, 3, 4, PI };
    float pts[10] = { 0 }; )
{
    ...
}
```

In the above example, default values are given for the entire *vals* array, in the event that an array of values is not specified in the RIB stream. The array *pts* is also initialized, but only the value for *pts*[0] is specified.

There are two other minor syntactic difficulties with arrays, due to the fact that the original language syntax was developed without arrays in mind. First, the existing square bracket notation in the *texture* call for channel number identification is syntactically ambiguous with the use of string array elements for texture names. This is handled by defining that if an element of a string array is used as a texture name, the string array element dereference must be contained within parenthesis. Any square bracket notation outside parenthesis is syntactically defined to be the channel number. Second, if an element of an array is used as an actual parameter to a function which returns a value in that parameter, the Spec seems to imply (due to its mention of call-by-reference) that this should set that element to the return value. This behavior may be extremely difficult to implement in the current run-time system, however, so programmers should not depend on it.

Lighting Control

It is often the case in the simulation of realistic scenes that the limitations of “computer graphics” lighting is one of the biggest barriers to success. A lot of work has been done to increase the amount of control that the Shading Language programmer has over the renderer’s lighting calculations.

Theatrical Lighting

Renderers often have a great deal of difficulty simulating the types of stage and natural lighting that film directors take for granted. Effects such as soft spill lighting, bounce cards, and eye spots can be approximated by particular algorithms (consider radiosity), but we want to simulate and control them directly in other renderers.

The standard technique for faking these types of effects are non-specular lights for soft spills, and non-diffuse lights for mirror reflections and eye spots. These techniques are accommodated by RenderMan through the use of special parameters on light shaders.

The specular and diffuse shadeops now respond to two special parameters of the light shader. Any light shader which contains the following parameter

```
float __nondiffuse = 1;
```

will be ignored by `diffuse` (note that there are two underbars). Similarly, any light which contains the following parameter

```
float __nonspecular = 1;
```

will be ignored by both `specular` and `phong`. Note that it is not the existence of this parameter that matters, it is the value of this parameter. If the value of the parameter is 0.0, the light is not ignored, so this behavior can be controlled both from the RIB file, or procedurally from inside the light shader, if desired. These parameters may be either `uniform` or `varying`. These special parameters can also be accessed within `illuminance` statements, as described in the section on Message Passing.

Light Source Categories

The RenderMan Specification doesn’t actually specify anything about the timing of the evaluation of the light sources, other than stating that the `illuminance` loop will iterate for each light source that is attached to the primitive (and satisfies the cone constraints). As a result, various renderers evaluate light sources at different times in the shading pipeline. Early versions of *PhotoRealistic RenderMan* evaluated all lights immediately after the displacement shader, before the surface shader started to run. The latest versions of *PhotoRealistic RenderMan* evaluate all lights simultaneously as soon as some shader needs the value of any of the lights (at the first call to `diffuse` or `specular`, for example, or at the first occurrence of `illuminance`). During its ray-tracing phase, *BMRT* evaluates each light individually during each iteration of `illuminance`, although the actual distinction between these two implementations is subtle for the Shading Language programmer to detect.

In the RenderMan Specification, however, there is no provision for evaluating only a subset of the lights which are attached to a primitive. It might be valuable for shaders to evaluate the contribution of certain lights before other lights, or independently of other lights. For example, it may be useful to have a special `illuminance` loop which only examines the contribution of ultraviolet light sources. A new parameter to the `illuminance` loop construct gives us this control.

```
illuminance( [string category; ] point P; [normal N; [float angle]] )
```

The optional parameter *category* specifies which shader category should be queried within the body of the illuminance loop. This category identifier is matched against the value of the special light-source string parameter `__category` (note the two underbars) using the following rules:

- The lightsource shader provides a comma-separated list of categories of which the lightsource is considered to be a member;
- The `__category` parameter may be overridden by values in the RIB file, but may *not* be computed by the light as an output parameter;
- The illuminance *category* matches if the string matches any of the values in the comma-separated list; except that
- If the first character of *category* is a - (a minus sign), the category matches if the string is not one of the values in the list.

For example, given the following lightsource shader:

```
light test ( float intensity = 1; )
    string __category = "distant,tony,pixar" ) {
    solar(vector "world" (0,0,-1)) { C1 = intensity; }
}
```

This shader would match, and subsequently be run and be available inside both of the following illuminance loops:

```
illuminance( "tony", P ) { C += C1; }

illuminance( "-larry", P ) { C += C1; }
```

Participating Media

One powerful feature that *BMRT* has always had, and which users of *PhotoRealistic RenderMan* have often clamored for, is the ability to simulate participating media. In order to simulate the way that lights illuminate and scatter off of volumes of small particles (such as smoke, fog banks, particulates in water, etc.), it is first necessary to be able to calculate lighting at any point in the volume. More specifically, it is necessary to be able to call lighting functions from inside volume shaders. As mentioned above, early versions of *PhotoRealistic RenderMan* did not permit this, as they evaluated light sources exactly once at P, prior to the surface shader execution. *BMRT* never had this restriction, and modern versions of *PhotoRealistic RenderMan* do not either.

The standard participating-media technique is to evaluate lighting at multiple locations along the ray, scattering some of this light forward onto the ray path. For example, consider the following volume shader:

```
volume glowingfog (float nsteps = 10; float albido = .03;)
{
    vector delta = I / nsteps;
    vector V = normalize(I);
    uniform float i;
    color C;
```

```

float scatter(float a; vector l;){
    extern vector V;
    return a * abs(V.l);
}

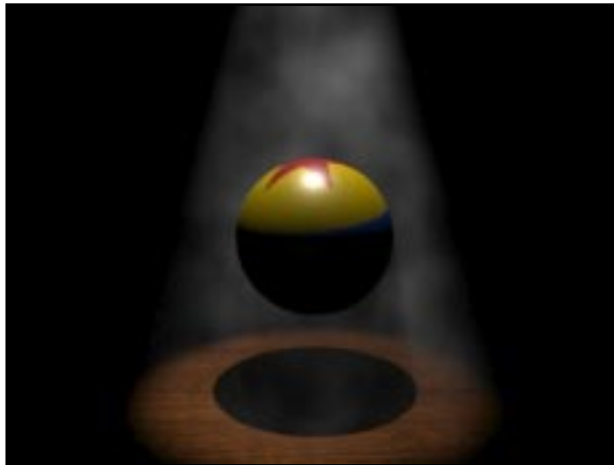
for(i = 1; i < nsteps; i += 1) {
    C = 0;
    illuminance(P - i*delta) {
        C += Cl * scatter(albido, normalize(L));
    }
}
Ci += C;
}

```

This shader integrates over a few locations along the ray from P back toward the viewer (skipping P and the viewer itself), scattering light from all the lightsources toward the viewer based on the angle from the light to the incident ray. (Actually this is an atrocious scattering function, but you get the idea.) Notice that if the light sources have any positional variation such as spotlight cones, luminar functions, or even shadows, these will be accounted for by the illuminance loop.

In a full ray-tracer, this technique does not need to be restricted to the primary rays (from the eye to the first visible object). In *PhotoRealistic RenderMan*, however, where the only available volume shader is the atmosphere shader, this restriction applies in practice.

Here's one example of a spotlight shining through a procedural fog, where the effect of the internal shadowing of the volume is quite apparent.



Soft Shadows

The newest version of *PhotoRealistic RenderMan* has an enhanced shadow shadeop supporting a new method of generating soft shadows with true penumbral fade-out. This is a simulation of the affect of area light sources on shadows. This method uses multiple rendered shadow maps to infer the visibility information from a light source whose extended geometry is also specified in the shadeop.

The enhanced `shadow` call is fully backwards-compatible, but supports a new form to allow specification of multiple shadow maps in a single call:

```
float shadow( string maplist, texture coordinates[, parameterlist] )
```

where `maplist` is a comma-separated list of shadowmap filenames.

This new method of generating soft shadows requires multiple shadow maps to infer geometry between the surface being shaded and the light source. For best results, the views should be placed on or near the light source. Specifying more views gives the renderer more information to work with, and thus can improve the quality of the shadows rendered. However, too many views can be unnecessarily expensive, so caution is advised. For most scenes, three to five shadow maps will be sufficient to capture the geometry sufficiently.

In addition to specifying multiple shadow maps, the user must give parameters to `shadow` to indicate the size and shape of the intended area light source. In the past, the location of the point light source (as far as shadows were concerned) was implicit in the shadow map matrices. The current system is able to generate area lights of various sizes from the same set of shadow-maps, hence the need for giving more information in the `shadow` call itself. The system can simulate both linear lights and triangular and quadrilateral area lights.

Soft shadows are not cheap. In particular, they will generally be half as fast as having n point shadow maps in the light source. (This is a rough average. Various geometric considerations may make shadows somewhat cheaper or somewhat more expensive.) However, it will also generally be the case that more samples are necessary to create a high-quality shadow, which will slow the process even more.

However, the result can be stunning. Consider the following example image:



Message Passing

There are four new Shading Language functions which permit shaders to peek at the parameters of the other shaders that are being executed on the same piece of geometry. This permits shaders to customize their behavior based on their knowledge of what the other shaders will do with those parameter settings.

```

float atmosphere ( string param; {any type} var )
float displacement ( string param; {any type} var )
float lightsource ( string param; {any type} var )
float surface ( string param; {any type} var )

```

These functions access the value of the parameter named *param* of one of the shaders attached to the geometric primitive that is currently being shaded. If the appropriate shader exists, and a parameter named *param* exists in that shader, and the parameter is the same type as the variable *var*, the value of that parameter is stored in *var* and the function returns 1; otherwise, *var* is unchanged and the function returns 0. `lightsource` is only available inside illuminance blocks.

Note that if the parameter named *param* is uniform, but the variable *var* is varying, the appropriate uniform-to-varying conversion will take place, however, the reverse case is considered failure.

For example, the new diffuse shadeop, which notices the special `__nondiffuse` light parameter, can now be written as follows:

```

color diffuse( point N ) {
    color C = 0;
    point Nn, Ln;
    uniform float nondif;
    Nn = normalize( N );
    illuminance( P, Nn, PI/2 ) {
        if ( lightsource( "__nondiffuse", nondif ) == 0 )
            nondif = 0;
        if ( nondif == 0 ) {
            Ln = normalize( L );
            C += Cl * Ln.Nn;
        }
    }
    return C;
}

```

Shaders can set flags and send them as “messages” to other shaders, by placing them in this globally-visible “blackboard” of output parameters, where they can be received (using the parameter-reading functions above) by any other shader that is savvy to their existence.

A more powerful variation of this theme is to use output *varying* parameters. This way, shaders can compute arbitrary values that vary across the surface, and pass them to other shaders. For example, consider the following pair of shaders, which shade a displaced surface based on the original shape (if that information is available).

```

displacement wavy ( float height = 1;
                    output varying point oldP = point "object" 0; )
{
    point Po = transform("object", P);
    point No = ntransform("object", N);
    oldP = Po;
    Po += height * normalize(No) * sin(2*PI*s);
    P = transform("object", "current", Po);
}

```



```

surface prepainted (float Kd = 0.5) {
    point Po;
    if (displacement("oldP", Po)==0)
        Po = transform("object", P);
    Ci = Kd * texture("monasmile.tx", xcomp(Po), ycomp(Po));
}

```

Notice that `prepainted` will work correctly if there is no displacement shader, and do the best it can if the displacement does not set `oldP`, but will have the special effect if `oldP` does exist.

The wily programmer may ask: At what point in the shading pipeline do output parameters get set so that I can read them? (Since there is no restriction on the use of the reading functions, one might imagine that the displacement shader might try read an output of the atmosphere shader before the atmosphere has ever been run.) The answer is complicated, but the basic rules are:

- If a value is set by its default, or by an override in the RIB file, it will be valid at all times.
- Values computed and output by the shader are handled during the normal execution of the shader, and so will be available and valid if the sender is run first, and will be bogus if the receiver is run first. For reference, shaders are executed on a primitive in the following order: first, displacement; then surface; lights are evaluated as subroutines of the surface; finally, atmosphere.

Calling C Functions from Shading Language

The latest release of *PhotoRealistic RenderMan* allows you to write new built-in SL functions in C or C++, and link them to the renderer as dynamic shared objects (DSOs). Such functions overcome many of the limitations of Shading Language user-defined functions. Most importantly, while functions implemented in Shading Language are restricted to operations and data structures available in the Shading Language, DSO shadeops can do anything you might normally do in a C program. Examples include creating complex data structures or reading external files (other than textures and shadows). For example, implementing an alternative `noise()` function, which needs a stored table to be efficient, would be exceptionally difficult in SL, but very easy as a DSO shadeop.

The process of writing and compiling these DSOs is extensively documented in the product documentation of the *PhotoRealistic RenderMan* renderer, and is likely to be different in differing renderer implementations. However, generally, the process is relatively simple.

The C file implementing a DSO shadeop contain a table describing the shadeop contained in the DSO — essentially mapping a Shading Language function template (with arguments) to the C implementations of those functions. Since functions in the Shading Language are commonly polymorphic (in other words, there are several versions of the function with the same name, which are distinguished by the types of the arguments passed to them), this is provided for by the data provided in this table.

The function implementations themselves get pointers to the function arguments, and a pointer to the location to store the function result. All Shading Language data types are available to the functions, and the type mapping is relatively obvious. Be very careful when writing your functions. If the DSO should have some bad behavior, like an infinite loop, runaway memory allocation or causes a core dump, the renderer cannot protect you!

When compiling your shader, if the Shading Language compiler comes across a function reference which is neither a known built-in nor a function already defined in the shader, it will search for a DSO with that name. The compiler will type-check your function call and issue a warning if you pass arguments which do not match any of the entries in the function table of your DSO.

When running your shader, the renderer will open the same DSO, and call the appropriate C function, with the specified arguments, once for each point which is being shaded.

DSO shadeops have several limitations that you should be aware of:

- DSO shadeops only have access to information passed to them as parameters. They have no knowledge of “global shaders variables” such as `P`, parameters to the shader, or any other renderer state. If you need to access global variables or shader parameters or locals, you must pass them as parameters to your DSO shadeop.
- DSO shadeops act as strictly point processes. They possess no knowledge of the topology of the surface, derivatives, or the nature of surface grids (in the case of a REYES renderer like *Photo-Realistic RenderMan*). If you want to take derivatives, for example, you need to take them in the shader and pass them as parameters to your DSO shadeop.
- DSO shadeops cannot call other built-in shadeops or any other internal entry points to the renderer itself.

New and Improved Built-in Functions

There are many new built-in functions which have been added to the Shading Language over the years, and new functions almost certainly continue to be added in the future. I will highlight a couple that I think are of particular interest.

Transformations

As has been intimated earlier, there are several new transformation functions, and several new flavors of existing transformation functions.

```
vector vtransform ( string [fromspace,] tospace; vector v )
vector vtransform ( [string fromspace;] matrix m; vector v )
normal ntransform ( string [fromspace,] tospace; normal n )
normal ntransform ( [string fromspace;] matrix m; normal n )
```

`vtransform` will transform the direction vector `v` from the coordinate system *fromspace* to the coordinate system *tospace*. `ntransform` does the same with a normal vector `n`. As always, if *fromspace* is omitted, it is assumed to be “current” space. The versions of these routines which take matrix arguments work identically to those taking string arguments, but use the 4x4 matrix instead of looking up a named coordinate system.

The `vtransform` and `ntransform` functions are useful because points, vectors and normals transform through different transformation matrices. Care should be taken that the correct transformation routine be used with each data type.

```
matrix translate ( matrix m, point t )
matrix rotate ( matrix m, float angle, vector axis )
matrix scale ( matrix m, point s )
```

These functions concatenate simple RI-like transformations onto existing matrices. Of course, you can always use the identity matrix as the starting matrix, if you need to generate a matrix from scratch.

```
color ctransform ( string [fromspace,] tospace; color C )
```

Transform the color *C* from the color representation *fromspace* to the color representation *tospace*. If *fromspace* is omitted, it is assumed to be "rgb".

String Manipulation

The Shading Language now has several interesting string functions which permit runtime manipulation of strings. This gives the programmer the ability to, for example, create texture map names on the fly, or recognize particular names by virtue of identifying substrings.

```
string concat ( string a, b, ... )
string format ( string pattern, ... )
float match ( string pattern, subject )
```

`concat` simply concatenates two or more strings into a single string. `format` does a more complicated formatted string creation under the control of *pattern*. This function is similar to the C function `printf`.

`match` does a string pattern match on *subject*. Returns 1.0 if the pattern exists anywhere within *subject*, and 0.0 if the pattern does not exist within *subject*. *pattern* is a regular expression, as is familiar to most Unix programmers. See the renderer documentation for more details. Note that the pattern does not need to start in the first character of the subject string, unless the pattern begins with the `^` (beginning of string) character.

Noise Functions

It is clear that noise functions play a critical role in the development of interesting, natural looking surface appearances. So, we have added several variations on `noise()` that makes it easier to develop noisy looks.

```
{float|color|point} noise ( point pt, float t )
```

The noise function now comes in a variety which takes a point and a float as arguments. This 4-dimensional domain may be interpreted as time-varying noise. Thus, you may now compute noise in 3-space which varies with time.

```
{float|color|point} cellnoise ( float x )
{float|color|point} cellnoise ( float s, t )
{float|color|point} cellnoise ( point p )
{float|color|point} cellnoise ( point p, float t )
```

`cellnoise` returns a value which is a random function of its arguments. Like `noise`, its domain is 1-D (one float), 2-D (two floats), 3-D (one point), or 4-D (one point and one float). Its return value is uniformly distributed between 0 and 1. Unlike `noise`, however, the return value has a constant value between integer lattice points, and is discontinuous at integer locations.

This feature is deceptively simple, and yet very interesting. In earlier versions of Siggraph course notes, the authors described a technique called *bombing*, where various patterns have particular set properties everywhere within a "cell". In other neighboring cells, the property has a different value, but nonetheless constant within that cell. For example, polka-dots were generated where each cell had a particular dot radius and dot center.

These bombs were implemented by calling `noise` with some function of the center of the cell, such as the `mod(ncells*s, 1.0)`. This construct was so common, and tweaking the function of `mod` to get a good, well distributed set of values was so subtle, that we created `cellnoise` to replace that construct entirely. Even better, `cellnoise` is considerably cheaper than calling `noise`.

```
{float|color|point} pnoise ( v, period )
```

`pnoise` computes periodic noise. That is, it returns a value similar to `noise` with the same arguments, however, the noise values returned by `pnoise` satisfy the relationship `pnoise(v, p) == pnoise(v+p, p)`. This is clearly useful if you wish to apply a noise pattern to an object which wraps around in one of the dimensions that noise is being applied. For example, `noise(u)` looks fine on a patch, but has a seam when applied to a cylinder. `pnoise(u, 1.0)` does not have a seam at the parametric discontinuity.

Every form of `noise` has a equivalent form of `pnoise` with one or two extra arguments (as necessary) to identify the length of the period. *period* must have an integer value (if it is a float expression), or lie on the integer lattice (if it is a point expression).

Time Derivatives

Two new global variables have been added to allow shaders to take into account the way that the surface changes with time (motion blur).

```
float dtime
```

The amount of time covered by this shading sample. This is similar to the way that `du` and `dv` are the amount of *u* and *v* being sampled by the shader at this point.

```
vector dPdttime
```

The vector indicating the way that the surface position *P* is changing per unit time, as described by motion blur in the scene. This is similar to the way that `dPdu` and `dPdv` give the change in *P* per unit of *u* and *v*, respectively.

Renderer State Queries

The latest release of *PhotoRealistic RenderMan* has new shadeops which provide access to global rendering internal state from inside the Shading Language. In each case, the function is prototypically like the shader variable access functions `surface`, `lightsource`, etc. Each function takes a string which is the name of a piece of global data, and a variable to contain the result. The functions return non-zero if the data exists and the variable is of the right type to contain the result, in which case the variable will be filled with the data requested. The functions return zero if any error prevents returning the requested data, including unrecognized requests, unobtainable data and variable type mismatches.

```
float attribute ( string, variable )
```

`attribute` returns data which is part to the primitive's RenderMan Interface attribute state, either from individual RI calls which set attributes, or from the `RiAttribute` call. *string* specifies the piece of RenderMan Interface attribute state which will be returned in *variable*. For example, a string "Sides" will return a float specifying the value of `RiSides` on the primitive. See *renderer* documentation for a complete list of attributes accessible.

```
float option ( string, variable )
```

`option` returns data which is part of the image's RenderMan Interface global option state, either from individual RI calls which set options, or from the `RiOption` call. `string` specifies the piece of RenderMan Interface option state which will be returned in `variable`. For example, a string "Format" will return three floats specifying the value of the parameters to `RiFormat`. See *renderer* documentation for a complete list of options accessible.

```
float textureinfo ( filename, string, variable )
```

`textureinfo` returns data which describes the format of a texture file. `string` specifies the information about the texture file named `filename` which will be returned in `variable`. See *renderer* documentation for a complete list of texture information accessible.

Miscellaneous Functions

Here are a few important odds-and-ends.

```
color specularbrdf ( vector L, normal N, vector V, float rough )
```

Returns the specular attenuation of light coming from the direction `L`, reflecting toward direction `V`, with surface normal `N` and roughness `rough`. This is the same *nonstandard* reflection model found in *PhotoRealistic RenderMan*'s `specular()` function.

This is useful if you write your own custom local illumination models using `illuminance`. Previously, it was impossible to write an illuminance loop that exactly matched the specular highlight behavior because *PhotoRealistic RenderMan*'s implementation of `specular` is proprietary and does not exactly match the implementation described in the *RenderMan Interface 3.1 Specification*. Now, using `specularbrdf`, you can exactly write illuminance loops which exactly match *PhotoRealistic RenderMan*'s proprietary specular highlight behavior.

```
float filterstep ( float edge, s1 [, s2]; [parameterlist] )
```

This new Shading Language function provides an analytically antialiased step function. In its simplest form, with two arguments, it takes parameters identical to `step`, but returns a result which is filtered over the area of the surface element being shaded. If the optional `s2` parameter is provided, the step function is filtered in the range between the two values. This low-pass filtering is similar to that done for texture maps (for reference, see page 129 of the *RenderMan Interface Specification*, *Texture Mapping Functions*). The *parameterlist* provides control over the filter function, and may include the following parameters:

- "width" (a.k.a. "swidth"), the amount to "overfilter" is `s`;
- "filter", the name of the filter kernel to apply. The filter may be any of the following: "box", "triangle", "catmull-rom", or "gaussian". The default filter is "catmull-rom".

Writing Surface Shaders

Tom Porter
Pixar

Opening comments

Let's leave the textbook examples behind and consider shader writing in everyday life. I want to give you a candid view of a couple of days of shader writing in the animation group at Pixar. I hope that you'll find that the process is straightforward, that there are a few simple principles to follow in putting shaders together, and that it is fairly easy to achieve *some* success. Look in the 1992 course notes if you want to see how hard it may be in achieving *complete* success.

Overview

- **Let's get beyond the textbook**
- **Consider some real objects**
- **What do you need to know?**
- **What do you need to do?**
- **What should you watch out for?**



I am going to talk only about surface shaders. *Surface* shaders comprise at least 90% of the shaders that get written. Production teams that we spend on a 30-second TV spot. We expect to write perhaps 1500 shaders for the movie we are undertaking.

I am going to talk only about simple shaders. *Simple* shaders comprise at least 90% of the shaders that get written. Others in this course may show some really clever stuff; my point is to give you a glimpse of the process.

What do you need to know before writing a shader? First, read the section on the shading language in the RenderMan book. Second, I found the notes to the 1991 and 1992 SIGGRAPH courses to be terrific. Third, consult all example shaders in the book, and in the notes, and those written by your colleagues. Finally, a high school math education helps, especially all that stuff about sines and cosines and other smoothly varying functions.

What do you need to do in writing a shader? That's the major portion of my talk. The quick and obvious answer is:

- Make sure you have good geometry.

- Know what you're trying to achieve.
- Start with a shader closest to what you are aiming for.
- Work on various aspects of the look one layer at a time.

What should you watch out for? Slow rendering times, aliased images, wasted effort. I will speak to these points throughout the talk. The overall principle here is to create a shader only as good as it needs to be.

Axioms of shader writing

I am not going to tell you much that your mother didn't tell you:

Time Honored Principles

- **Stop, Look, and Listen**
- **One step at a time**
- **Everything in moderation**
- **Lie, cheat, and steal**

r

- *Stop, Look, and Listen.* Don't set foot until the street is clear and you have an idea how to get to the other side.
- *One Step at a Time.* Build up your shader one layer at a time, mimicking the manner in which the simulated object achieves its real surface characteristics, whether grown or manufactured.
- *Everything in Moderation.* There are (Nyquist) limits beyond which your shader should not go. Take care.
- *Lie, Cheat, and Steal.* This is only computer graphics, a science of simulation. The shader needs to be accurate only within a certain range of conditions. Steal code from your previous shaders and patch together something that works.

Know what you're after!

Pattern Generation: Divide & Conquer

- **Bowling pins have color, scratches, labels**
- **Soccer balls have color, bumps, dirt, decals**
- **Leaves have color, blemished, striation**



As with many programming tasks, you should adopt a strategy of divide and conquer in planning the shader. Bowling pins have color, scratches, and imprinted labels. Soccer balls have a color pattern, bumps and dirt. Leaves have a color pattern, blemishes, and striations.

You will need such a plan of action in creating the shader. Break down the surface features into distinct characteristics and concentrate on the most significant ones. Failure to do this at the outset will hamper you throughout the process. Resist the urge to hack a shader from the very start -- there will be excellent opportunities for hacks as the shader progresses!

Know What You're After

- **Get physical models, photos, renditions**
- **Understand how the real object is made**
- **Decide on the accuracy needed**



You must understand what you are trying to achieve with the shader. Get as many physical examples of comparable surfaces as you can. Rip pages out of picture books and post them on your wall.

Consider the mango. We accepted a job to animate fruit for multiple Tropicana juice ads. I had the task of putting together shaders for several of the fruit. I am going to walk you through the process of writing such shaders. This will be a fairly complete look at the process, showing the shader code and the test pictures at several stages of the process.

The first order of business is to buy mangoes. The ads called for a realistic depiction of the fruit, so I purchased some realistic fruit. Of course, television realism differs from grocery store realism, but I use nature as a starting point nonetheless. In fact, this point about realism needs to be made again. Very often, a realistic simulation of a physical object evolves into a realistic simulation of the shared and glorified human expectation of the look of an object. The man in the street expects a mango to look *different* than it really does, but we *still* use nature as a starting point.

Consider The Mango

- **Buy a few mangoes**
- **Notice a base color, little spots, and larger marks**
- **Cylindrical growth; singularities at stem and base**



Next, contemplate your mango. Stare at it. Understand how nature grows it. Theorize on what causes variations among mangoes. What is its color? What is its reflectivity? What is its bumpiness? Take as much time as you need to understand the 4 or 5 principal features of the surface.

An hour with a mango will convince you that it has significant low frequency color variation, lots of tiny dark spots, a minor amount of wrinkling, and a stem at the top. Some people can deduce this sort of information in a matter of minutes, but the serious shader writer will walk the hallways with mango in hand for at least an hour.

Now you need to find out the range over which your mango will be seen in the animation. Allocate your time for writing this shader based on the maximum number of pixels covered in any frame. If your mango is just making a cameo appearance in the back of a large crowd scene, stop now and use the plastic shader with an orange/green tone. If your fruit is to be rendered full screen, get out the magnifying glass and spend another hour meditating about your mango.

Pattern & illumination

Shaders Involve Pattern and Illumination

- **Start with plastic.sl**
- **Copy the illumination function**
- **Invent the pattern**
- **Your job: create some clever function for Cs**
- **...or maybe Os, Ks, Kd**



A shader involves pattern generation and illumination. Look at any shader in the RenderMan book, and you will find a fairly standard illumination function at the end, with contributions for ambient, diffuse, and specular light. Everything else is a pattern that you devise, usually from scratch. Your shader will probably have a standard illumination function. In fact, a good starting point is to copy the code from `plastic.sl` and start from there.

Render a picture with `plastic.sl`.

But first, a digression on test rendering

I have the luxury of fast workstations, and you may conclude that I employ lots of test renderings because I can afford to abuse CPU time. This is true. But no matter what machines you render on, you will want to render tests quickly. To that end, you should clearly understand the effect of the `RiFormat`, `RiShadingRate` and `RiPixelSamples` calls to RenderMan. These are the best to adjust for the purpose of fast test renderings.

The total rendering time is proportional to the cost of computing each shader at various points over the surface of each possibly-visible object on the screen. Use `RiFormat` to limit the resolution of the target image. Use `RiShadingRate(4)` to ask the renderer not to compute the shader more than once every 2x2 pixel area. Use `RiPixelSamples(2,2)` (or (1,1)) to limit the number of hidden surface determinations to only 4 (or 1) per pixel.

The plastic mango will confirm two things: the geometry and the lighting. Both geometry and lighting should be adequate before proceeding. Do not expect to fix either of these in the shader.

Plastic Mango

- **Plastic illumination function will do**

```
Ci=cs * (ka*ambient() +
          kd*diffuse(Nf)) +
      ks*specular(Nf,V,roughness);
```

How shall we set cs, ka, kd, ks, Nf?



By copying the illumination function from `plastic.sl`, you have completed half the shader. Good work. Now you must invent the pattern. You will notice several coefficients in the plastic illumination function, such as `cs`, `os`, `ks`, `kd`. These are often locked down as parameters to the shader and written as `CS`, `OS`, `KS`, `KD`. I write them in lower case to stress that each of these can be smoothly varying function. Your job in inventing the pattern for the surface is usually to design some sufficiently clever function for `cs`. This is a classic procedural texture map, one that varies the color over the surface. By setting patterns for other variables, you can create opacity maps and bump maps and lots of other effects.

Procedural vs. photographic texture patterns

You will often face the decision of writing a procedure or scanning a photograph to describe a pattern. Both are viable alternatives, with pros and cons. The fact that RenderMan offers a shading lan-

Patterns: Photographic vs. Procedural

- **Procedures scale, can be tuned up**
- **Photographs scan, can be touched up**
- **Both need “wrapping”**
- **Guiding Principle:**
Use photos for artwork, procedures otherwise



guage doesn't mean that you should *not* use a photograph; it just tempts you with greater flexibility in writing procedures. Quite often, you will find that the problem of wrapping the pattern onto the surface is as hard as detailing the pattern itself.

Procedures can scale nicely as the object moves. Procedures are easily adjusted, just a minor matter of reprogramming. Photographs can easily be scanned and touched up, just a minor matter of re-painting. The guiding principle that I live by is to use scanned photographs for artwork, such as product packaging, and to try my hand at programming everything else.

Thus, for bowling pins, scan the Brunswick decal and program the color variations over the surface. For the scratches and dirt, try writing a procedure. In 1989, I painted the scratches and bumps. Today, I might be more tempted to program them. Mango skins should be programmed -- they are too messy on a scanner anyway.

Mango: base color

Mango: The First Attempts

- **Start with `plastic.sl` to confirm geometry**
- **Use `show_st.sl` to understand orientation**
- **Set up lighting for test shot**
- **Try library shaders to gauge color, bumpiness**



After starting with the plastic shader, I tried two shaders from the shader library that we keep. The first was a peach shader, just to look at the nature of color variation over the surface. It is clear that the mango has a far lower frequency variation. The second is a simple bump-mapped shader used for a wall in *Tin Toy*. I was interested in comparing the results with the natural bumpiness of mangoes. I set it up with my first attempt at an orange-green color, and produced a picture vaguely reminiscent of a chocolate leather. I quickly squirreled *that* shader away for some future use.

It was clear that neither of these would lead anywhere, so I returned to `plastic.sl` and set out to design a pattern ranging over the s - t of the mango geometry. I thus ran a test with `show_st.sl` to understand how s and t varied over the surface.

Next, I set out to simulate the distinctive color variation of the mango. I needed a variable which would vary slowly over the surface. Note that I needed the variable i to “wrap” in s , showing no seam as s wraps around from 1 back to 0. Thus, I use $(\sin(2\pi s), t)$ instead of (s, t) as arguments to the noise function. Notice that the (s, t) really is appropriate here; mango skin does expand in a cylindrical fashion, creating singularities at the top and bottom.

Mango: Base Color

- **Compute a smoothly varying index variable**

```
i = noise(.5*sin(2*PI*s),2*t);
```

- **Index into a spectrum of color**

```
spotcolor = spline(i,grn,ylw,  
                  org,red);
```

r

Now I use the spline function to turn the variable *i* into a smoothly varying color variable *cs*. That is what I plug into in the illumination function, to good effect.

Mango: spots and marks

The color of the spots of the mango vary across the surface. The reddish parts of mangoes I buy in Marin County grocery stores have yellow spots, and the yellowish parts have white spots. It is a simple matter to run the spline function through a new color table to compute the spot color. The spots themselves are positioned by computing a new variable *darken*. We take the noise function with parameters $(150*s, 150*t)$ to create a smoothly varying function with at least 150 “cycles” in it. We then square the function as a way of sharpening the peaks of the function and pushing most of the values less than 0.25. We then use this variable in mixing the spot color with *cs*.

Mango: Spot Color

- **Use high frequency noise for position of spots**

```
darken = float noise(150*s,150*t);
```

- **Sharpen the peaks of the noise function**

```
darken = pow(darken,2);
```

- **Mix the spots in with the color**

```
cs = mix(c,spotcolor,darken);
```

r

This has been the standard use of the `noise` function, to create a random and smoothly varying function with some guaranteed frequencies to it. As a further refinement, I decided to darken the tips of the spots. If the `darken` variable is greater than 0.33, I mix in the darker *mark color*.

Stop! A digression on transitions and aliasing

This last statement should set off alarms in the head of any self-respecting shader writer. The use of an `if`-statement in a shader is dangerous! Up until now, we have been dealing with smoothly varying functions. The `if`-statement allows us to go beyond the safety of these functions. Transitions are evil, and `if`-statements cause transitions.

Under no circumstances should you create a discontinuity in the color of the surface. This will almost certainly create visible aliasing. Similarly, you should avoid discontinuities in the reflectivity or any other variable contributing to the illumination function, though the visual effect of such discontinuities may be less pronounced. In fact, we do keep the color function continuous in this code, but we do create a discontinuity in the first derivative of color. This can be the cause of Mach banding, and we should watch for that in looking at the rendered pictures.

Earlier, you saw a discussion of tools such as `smoothstep()` used to keep functions smooth and combat various forms of aliasing. For our purposes here, notice that I keep my functions continuous and don't put too many wobbles in them.

Mango: Mark Color

- **Use lower frequency noise for position**

```
darken = float noise(30*s,30*t);
```

- **Leave very few peaks of the function**

```
darken = pow(darken,5);
```

- **Use spline again to create spectrum**

```
markcolor = spline(i,ylw,bej,brn,
                  bej,ylw);
```

Every mango I looked at had black marks, skin blemishes caused by growth, picking, or handling. I noticed perhaps 5 to 10 marks per mango, the size of your smallest fingernail. These are computed by taking a medium-frequency noise function in `s` and `t`. We then sharpen it by putting to the 5th power, so that very little of it is greater than 0.05. We then run the function through a sin function to round it out a little bit. We then use this variable to control the mixing of a dark mark color.

Mango: generalize, wrinkle, and finalize

Mango: Generalize

- **Introduce “label” argument**

```
darken = noise(150*(s+label),
               150*(t+label));

darken = noise(30*(s+label),
               30*(t+label));
```



We now introduce the `label` argument in the shader, to generalize this code for use on multiple mangoes. We want these characteristic color, spot and mark computations to occur for each mango, without producing clones. We therefore give each mango a label from 0 to 1, and use `label` to access each noise function that we call. This provides the variability we want.

Mango: Bump Map For Wrinkle

- **Define `p2 = (sin(2*PI*s),2*t,cos(2*PI*s))`;**

```
#define AMPLITUDE1/40
#define FREQUENCY 25
turb = noise(FREQUENCY*p2)*AMPLITUDE;
newP = calculatenormal
      (P+turb+normalize(N));
Nf = faceforward(normalize(newP,I));
```



Until now, we have carried along this computation of `Nf` to be a forward facing normal to the surface of the mango. Let's adjust that slightly to give the impression of wrinkles on the surface. This is bump-mapping. We compute a point `p2` in space and run that through a noise function, subject to a

defined frequency and amplitude. This value is used move the surface point to some other spot along the surface normal.

Mango: Align Wrinkle With Spots

- **A different bump map**

```
p2 = transform("shader",P);
turb = noise(FREQUENCY*p2);
if (turb<.8) turb = 0;
else turb = (turb-.8)/.2;
turb = turb*AMPLITUDE;
newP = calculatenormal
      (P+turb*normalize(N));
Nf = faceforward(normalize(newP,I));
darken = turb;
```



In the course of playing around with bump maps, I did take some wrong turns. I had this great idea of aligning the color spots with bumps. I thus ran through the bump mapping code, use the computed turbulence function as the variable `darken`, used for the spot color. I tried it two different ways, with different bump frequencies and amplitudes. As the pictures show, the dimples and pimples are not too pleasing.

Mango: Final Shader

- **plastic illumination; spots and bruises**
- **bump map for nice wrinkle**
- **noise() and spline() and mix() are you friends**
- **fast rendering helps tremendously**



I went back to the wrinkled bump-mapping, adjusted a few numbers and created the final picture. The final mango shader is listed in the first appendix.

Here are few items to keep in mind about the process:

- `noise()` and `spline()` and `mix()` are your friends
- use `plastic.sl` and `show_st.sl` to check geometry and lighting
- fast rendering of lots of test images helps tremendously

As a minor point, I keep open two text windows and a bunch of image windows on the workstation screen. One text window is for editing; when you get rolling you are able to edit the next shader change as the last shader is being rendered. The other text window is merely for initiating the shader compiler and renderer. The rest of the screen is commonly littered with versions of the rendered picture.

Consider the banana

(The lecture will include a quick discussion of a similar process for developing a banana shader.) The final shader is listed in the second appendix.

Conclusions about fruit

We see a number of similarities between bananas and mangoes. It seems that nature has a surprisingly small repertoire of tricks that it uses to differentiate fruit! Get the basic color right. Use the noise function and some high school math to create smoothly varying functions across the surface. Proceed one layer at a time, and create a small number of adjustable parameters to allow for tuning the images. Use a label parameter to create largely similar functions for different pieces of fruit.

Random walks through n-D space

I started by insisting on Divide-and-Conquer as an approach to shader writing. The whole point of this strategy is to tease apart the different layers of surface characteristics and set up adjustable variables for tuning each layer independently. This is the optimal strategy.

Random Walks Through n-D Space

- **Avoid the random walk with Divide & Conquer**
- **But events may conspire against you**
- **Guidelines**

Make sure lighting is right and monitor is good

Wedge it (sparsely sample the n-D space)

Start with most significant parameter



There will be times, however, when the strategy breaks down. We have had occasions where the shader writer can make an object brighter or bumpier or browner, but the Art Director wants it “better”. You will find yourself a few hours from a deadline and without a clue. Here are some quick thoughts to keep in mind when you find yourself in such a predicament.

Make sure once again that the lighting for your shader is similar to the lighting used for the entire scene. For the same reasons, use a monitor that you trust. All too often, I have attempted to optimize a shader for lighting and monitor settings other than those to be used in the real scene.

Determine the most significant feature of the surface and start by getting that right. Usually, this is color variation, but there will be times when it is really the quality of the highlight or reflectivity which turns out to be the most important feature.

When all else fails, you will find yourself with a shader with n adjustable parameters and no idea how to set them. You are now wandering around in an n -dimensional space, a space skewed because of interdependencies of some of the dimensions. I suggest that you take a scattershot approach. Sample this n -D space very sparsely by choosing random values for each variable. Compute a picture for each selection and take a look at the results. Visually interpolate the images and average together one final setting for each the variables.

Fast shaders

Shaders That Execute Quickly

- **There is no magic; `sqrt()` and `noise()` take time**
- **Avoid calling noise more than a few times**
- **Avoid large-scale hit-testing in the shader**



There is no magic here. If you include lots of floating point computations in your shaders, your rendering will run more slowly. Certain functions take longer to compute than others on your computer. Square roots might be optimized on some machines, but you will find that matrix inversions almost always take time. I find the `noise` function indispensable, but it also takes some time to compute. I try to limit myself to four well chosen calls to the `noise` function per shader.

The efficiency consideration which I face most often is *hit* testing. The surface pattern is generated implicitly rather than explicitly. A shader answers the renderer’s question: What color is the surface at point P? The easiest shaders to compute are those which create an analytic function to describe the surface. When the renderer asks for the color at a point, the shader samples the function.

Explicit descriptions of surfaces are much harder to deal with. Consider a PostScript file, an explicit list of drawing primitives, as a surface description. The only way to compute the color at point P is to test whether each and every drawn primitive hits the point P. If the list is long, the rendering will take forever. The only reasonable way I know to handle a PostScript surface is to create a texture map at a sufficiently high resolution and map it on.

Hit-Testing in the Shader

- **Surface pattern is generated implicitly**
- **Shader's job: What color is surface at point P?**
- **Can afford to create function and sample it**
- **Cannot afford to hit-test entire PostScript file**
- **Basketball example**



So this is the *hit testing* problem: What is the most efficient scheme for computing the color at point P when you have a list of primitives which might hit point P? As you can imagine, if the list is long or if the primitives are unwieldy, this process can involve quite a bit of computation. *Avoid this problem at every opportunity!* Here is an example of a shader which successfully avoids it.

A basketball has little circular nibs covering its surface. The circles are not in any regular pattern; they are densely packed but generally random. After much thought about clever ways to decide whether a point on the surface was inside one of the nibs, I gave up and used a texture map. I felt that I had a much better chance of producing a pleasing distribution of nibs by using a C program to create a texture than with a shader. In fact, I used a dart throwing algorithm suggested by Don Mitchell in a SIGGRAPH 1991 paper to space the nibs. The final basketball shader merely wraps the texture map around the patches making up the sphere.

Concluding comments about our axioms

So what have we learned from all this?

- Make sure the input geometry is adequate.
- Know what you're trying to compute.
- Steal code from previous shaders.
- Divide and Conquer.

But these axioms are no different from those for any programming effort! Edit, compile, run, debug. Just be thankful it's graphics and not accounting. Laugh at the buggy intermediate results and enjoy the final pictures.

Appendix 1: Mango Shader

```

surface
mango(
    float label = 0.0;
    float Ka = 1.0;
    float Kd = .6;
    float Ks = 0.1;
    float roughness = .1;)
{
    point Nf,V;
    color cs ;
    float darken;
    varying float i;
    color spotcolor,markcolor;
    point turb,p2;
    point newP;

    color whitey = color (1.0,0.70,0.1);
    color yellow = color (0.9,0.50,0.0);
    color yelora = color (0.9,0.40,0.0);
    color orange = color (0.9,0.30,0.0);
    color redora = color (0.9,0.20,0.0);
    color red = color (1.0,0.10,0.0);
    color green = color (0.25,0.50,0.0);
    color greora = color (0.47,0.43,0.0);
    color oragre = color (0.69,0.37,0.0);
    color brown = color (0.1,0.05,0.0);
    color lbrown = color (0.5,0.25,0.05);

    /* Nf = faceforward(normalize(N),I);*/

    setxcomp(p2,sin(2*PI*s));
    setycomp(p2,2*t);
    setzcomp(p2,cos(2*PI*s));
#define BUMP_AMPLITUDE (1/40)
#define BUMP_FREQUENCY (25)
    turb = noise(BUMP_FREQUENCY * p2)*BUMP_AMPLITUDE ;
    newP = calculatenormal(P + turb * normalize(N));
    Nf = faceforward(normalize(newP), I);

    V = -normalize(I);

    /* Now let's get the basic color of the mango. */

```

```

#define SFACTOR .5
#define TFACTOR 2
    i = float noise(SFACTOR*sin(2*PI*s)+PI+
                    label,TFACTOR*t+1.414+label);
    cs = spline(i,green,green,greora,
               oragre,redora,red,red );
    spotcolor=spline(i,green,yellow,whitey,
                    yellow,whitey,yellow,yelora);
    markcolor=spline(i,yellow,lbrown,brown ,
                    brown,brown,lbrown,yellow);

/*
Now let's look at the spots on the mango by looking up a noise
value.
We will be setting the darkening coefficient "darken".
*/
    darken = float noise(150*(s+label),150*(t+label));
    darken = pow(darken,2);
    cs = mix(cs,spotcolor,darken);
    if (darken > .33)
        cs = mix(cs,markcolor,(darken-.33)*1.5);

/*
Now let's put some bruises on the mango by looking up a noise
value. We will be setting the darkening coefficient "darken".
*/
    darken = float noise(30*(s+label),30*(t+label));
    darken = pow(darken,5); /* looks ok between 4 and 6 */
    darken = .5+.5*sin(PI*(darken-1/2)); /* to sharpen the func-
tion */
    cs = mix(cs,markcolor,darken);

    Oi = 1.0;
    Ci = cs * (Ka*ambient() + Kd*diffuse(Nf))
        + Ks*specular(Nf,V,roughness);
}

```

Appendix 2: Banana Shader

```

surface
banana(
    float label = 0.0; /* for each banana, choose random in [0,1) */
    float spotted = 0.5; /* 0 for perfect yellow; 1 for bruised */
    float lined = 0.7; /* 0 for perfect yellow; 1 for lined */
    float Ksmix = 0.8; /* 0 for white specular color; 1 for yellow */
    float Ka = 1.0;
    float Kd = .7;
    float Ks = 0.2;
    float roughness = .1;)
{
    float Scale,lineScale;
    float level,level2,level3;
    float angle,dangle;
    point Nf,V;
    point Nfd,Nfs;
    point p1;
    float x,y,z;
    color cs ;
    float labels;
    float darken;
    varying point turb;
    color specularcolor;
    point p2,newP;
    float ks;

    color yellow = color (1.0,0.55,.0);
    color white = color (1.0,1.0,1.0);
    color green = color (0.3,0.25,.0);
    color grelow = color (0.50,0.35,.0);
    color lbrown = color (.5,.25,.0);
    color dbrown = color (.07,.05,.025);

    color bruiseColor = color (0.2,0.05,.025);

    Scale = 8-5*spotted;
    lineScale = 8-7*lined;
    ks = Ks;

    setxcomp(p2,sin(2*PI*s));
    setycomp(p2,3*t);
    setzcomp(p2,cos(2*PI*s));
    Nf = faceforward(normalize(N),I);
    Nfd = Nf;

```

```

/* We'll use a bump map just to disperse the specular highlight */
#define BUMP_AMPLITUDE (1/40)
#define BUMP_FREQUENCY (25)
    turb = noise(BUMP_FREQUENCY * p2)*BUMP_AMPLITUDE ;
    newP = calculatenormal(P + turb * normalize(N));
    Nfs = faceforward(normalize(newP), I);

    V = -normalize(I);

/*
Now let's get the basic color of the banana, by looking primarily
at t. We will also affect the specular coefficient.
*/

    labels = s + label;
    level = t+.02*sin(2*PI*labels) +.01*cos(2*PI*labels)+
        .03*noise(s);
#define L1 .13
#define L2 .16
#define L3 .45
#define L4 .70
#define L5 .85
    if (level <= L1) {
        cs = dbrown;
        ks = 0;
    } else
    if (level <= L2) {
        cs = mix(dbrown, yellow, (level-L1)/(L2-L1));
        ks *= (level-L1)/(L2-L1);
    } else
    if (level <= L3) cs = yellow;
    else {
    if (level <= L4) {
        cs = mix(yellow, green, (level-L3)/(L4-L3));
        ks *= (level-L4)/(L3-L4);
    } else {
        cs = spline((level-L4)/(1-L4),
            grelow,green ,green ,green ,grelow,
            dbrown,lbrown,lbrown,lbrown,lbrown);
        ks = 0;
    }
    }
}

```

```

/*
Now let's look at the bruises on the banana by looking up a noise
value. We will be setting the darkening coefficient "darken".
There is one calculation for the bruises along the seams; there
is another for the bumps on the side.
*/
#define AA .2
    angle = mod(s, AA);

#define B0 .015
#define B1 .025
#define B4 .18
#define B5 .19

    if ((level < L5) && (level > L1)) {
#define SF 1
#define tF 5
        dangle = angle-AA/2;
        if (dangle < 0) dangle += AA;
        if (angle <= B0)
            darken = float noise(sF*sin(2*PI*(s+.1234))+
                                sF*dangle/AA,tF*t);
        else
            if (angle <= B1)
                darken = float noise(sF*sin(2*PI*(s+.1234))+
                                    sF*dangle/AA,tF*t)*(B1-angle)/(B1-B0);
            else
                if (angle >= B5)
                    darken = float noise(sF*sin(2*PI*(s+.1234))+
                                        sF*dangle/AA,tF*t);
                else
                    if (angle >= B4)
                        darken = float noise(sF*sin(2*PI*(s+.1234))+
                                            sF*dangle/AA,tF*t)*(angle-B4)/(B5-B4);
                    else
                        darken = 0;

        if (darken > 0) {
            darken = sin((PI/2)*darken);
            darken = darken*lineScale;
            darken = darken-lineScale+1;
            if (darken < 0) darken = 0;
            cs = mix(cs,bruiseColor,darken);
        }
    }

```



```

#define A0 .005
#define A1 .08
#define A2 .09
#define A3 .11
#define A4 .12
#define A5 .185

#define SF 10
#define TF 50
    if (angle <= A0)
        darken = float noise(SF*(s+angle/AA),TF*t)*(angle/A0);
    else
        if (angle <= A1)
            darken = float noise(SF*(s+angle/AA),TF*t);
        else
            if (angle <= A2)
                darken = float noise(SF*(s+angle/AA),TF*t)*(A2-angle)/
                    (A2-A1);
            else
                if (angle >= A5)
                    darken = float noise(SF*(s+angle/AA),TF*t)*(AA-angle)/
                        (AA-A5);
                else
                    if (angle >= A4)
                        darken = float noise(SF*(s+angle/AA),TF*t);
                    else
                        if (angle >= A3)
                            darken = float noise(SF*(s+angle/AA),TF*t)*(angle-A3)/
                                (A4-A3);
                        else
                            darken = 0;

/*
Now we have the basic darkening coefficient.
The following is a little math to scale the coefficient
appropriately.
*/
    if (darken > 0) {
        darken = sin((PI/2)*darken);
        darken = darken*darken;
        darken = darken*Scale;
        darken = darken-Scale+1;
        if (darken < 0) darken = 0;
        cs = mix(cs,bruiseColor,darken);
    }
}

```

```
/*  
One of our parameters is the extent to which the banana reflects  
specular highlights as white or yellow.  
*/  
    specularcolor = mix(white,yellow,Ksmix);  
  
    Oi = 1.0;  
    Ci = cs * (Ka*ambient() + Kd*diffuse(Nfd))  
        + ks*specularcolor*specular(Nfs,V,roughness);  
}
```

RenderMan Tricks Everyone Should Know

Tony Apodaca
Pixar Animation Studios

In this section of the talk, I will outline a few “basic” advanced tricks that everyone should already know about. Some of these topics will be discussed in more detail in the afternoon sessions, so I will only touch on them briefly. The more of these that are old news to you, the more advanced you are. Don’t worry. If you know *all* of these already, then you are all prepared for Larry’s talk. I will proceed in the order of most pragmatic to most unusual.

Using Level-of-Detail

The traditional model-based special effects industry have a long tradition of building multiple models of objects to satisfy various shot requirements. Long shots can utilize relatively small and undetailed models while close-ups often use very large and detailed models. In computer graphics, we can benefit by adopting a similar methodology, because simpler models require less time and memory to render versus their large and complex versions.

PhotoRealistic RenderMan implements the *Level of Detail* Optional Capability of the RenderMan Interface Specification. The beneficial features are

- the user may specify multiple versions of a particular object in a RIB file;
- the renderer automatically selects the version appropriate for final size of the object on the screen;
- *PhotoRealistic RenderMan* can smoothly transition between multiple versions of a model;
- this transition can be made to be smooth despite large-scale changes in topology, shading model, or even color of the different versions;
- memory and time can be drastically cut compared to not using level of detail;
- appropriate complexity models are far easier to antialias, leading to fewer artifacts in animation.

Specifying Level of Detail

Two quantities are maintained in the hierarchical graphics state: the *current detail* and the *current detail range*. The *current detail* is a size metric, specified by giving an axis-aligned bounding volume in the current coordinate system:

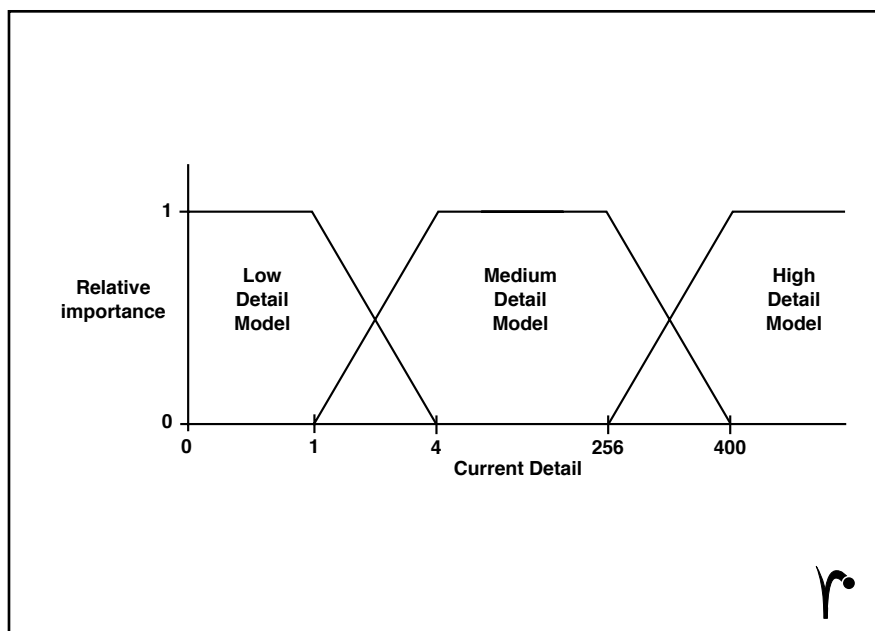
```
Detail [ xmin xmax ymin ymax zmin zmax ]
```

The RIB file provides the *current detail* in object coordinates, and the renderer then computes the raster space area of the projection of the specified bounding volume onto the screen. The *current detail range* specifies the range of raster-space *current detail* over which the particular model representation is to be used. The detail range is given by four numbers:

```
DetailRange[ minvisible lowertransition uppertransition maxvisible ]
```

The four detail parameters trace out a graph describing the relative importance of each representation of the model. Each geometric primitive in the representation will be used, or not used, depending on the following rules:

- If the *current detail* is outside the range, importance is 0, so all primitives in that representation will be discarded without rendering them;
- If the *current detail* is in the center region, importance is 1, and the primitives will be rendered normally;
- If the *current detail* lies in the rising or falling regions, the model smoothly transitions between this representation and representations specified for other detail ranges, based on their relative importance.



```
AttributeBegin                # push the graphics state
Detail [-1 1 -2 2 0 8]       # setup a "ruler" for detail calcs
DetailRange [0 0 1 4]
    # ...primitives for the low detail model (< 2 pixels)
DetailRange [1 4 256 400]
    # ...primitives for the medium detail model
DetailRange [256 400 1e38 1e38]
    # ...primitives for the high detail model
AttributeEnd                  # pop the graphics state, LOD object done
```

One may use as few or as many levels as desired, but the sum of all model importances should be 1.0 over the entire range of potential detail values, in order to keep objects from being accidentally over- or underrepresented in any particular image. In practice, this means that each representation's transition regions should exactly match the next higher and next lower representation, as shown in

the figure above. The sole exception to this rule is that it may be valuable to underrepresent the lowest complexity representation, to allow the object to fade out as it falls below some minimum size.

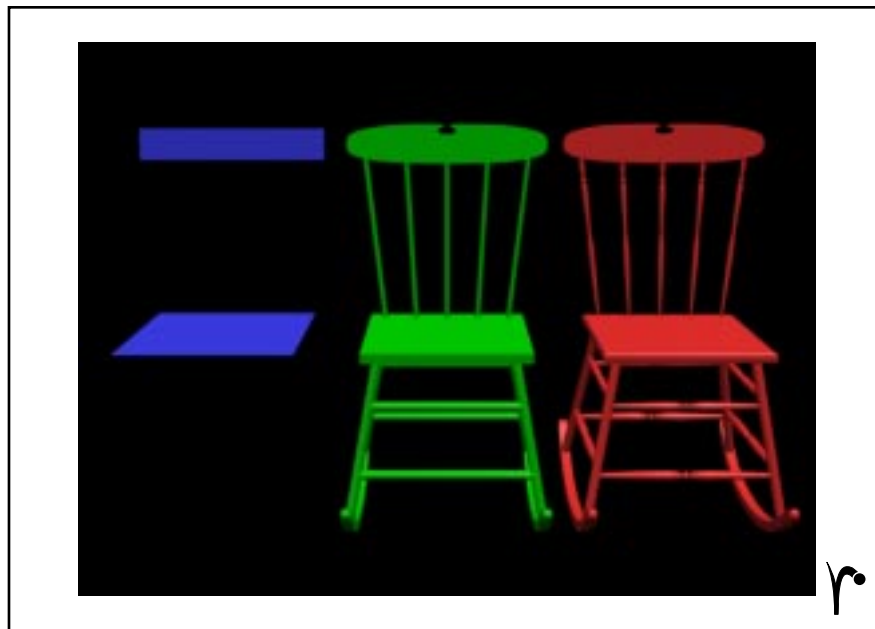
While it may be useful to fade between representations of a model in a still frame, it is far more interesting to use level-of-detail in animation. In animation, the raster size of a model will change as the model or camera move, and therefore each frame will typically have different contributions by the various representations.

Interestingly, the specified *detail* box which controls the selection of model representations does not have to be the bounding box surrounding the entire object. Rather, it is often better to use a box around a feature of a particular interesting size within the model. For example, the *detail* of an entire tree may be most easily quantified by the box around a leaf. The detail box should be considered to be a “ruler” in the scene, not a containing box.

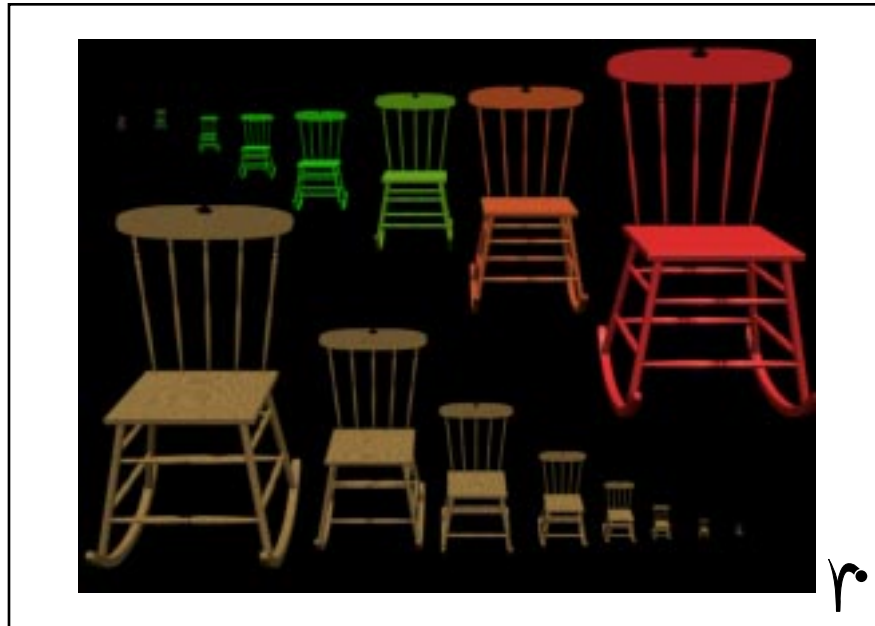
Choosing detail ranges which appropriately remove complexity when it is too small to see is a bit of an art. In addition to the obvious replacement of complex (high vertex count) surfaces with simpler (smoother, low vertex count) versions, it can also include shader tricks such as replacing displacements with bumps and replacing complex procedural shaders with texturemap approximations, or even eliminating features when they become too thin or too small for the renderer to reasonably antialias. The renderer does an excellent job of blending based on relative importance, but nevertheless, the smoothness of transitions in animation is largely dependent on how carefully the models are constructed. If silhouettes change radically, or surface coloration changes drastically, it may be possible to see the changes, even if they occur smoothly.

Level-of-Detail Examples

Three different models of a chair were created. The simplest is only two bilinear patches. The second has several patches, and some simple cylinders for the spindles in the back. The most detailed model has nicely sculpted surfaces for all of the spindles.



A RIB file was created specifying appropriate detail levels for each representation, and chairs at different distances were rendered. Across the top, representations were color coded to show how the various representations are smoothly blended. The chairs across the bottom illustrate a more realistic application. Even when animated, the transitions between these models occur smoothly.



Using Level of Detail with Procedural Primitives

In some productions which have attempted to use models with varying complexity, but without the advantage of this Level-of-Detail feature, it has been incumbent on some (poor) TD to painstakingly determine for every shot (or even for every frame of every shot) which representation of each model was the best to use for that particular rendering. It is easy to see that this is a tedious, error-prone and unglorious task. With level-of-detail, the renderer is able to determine with very little computation which representations will be used in a particular image, and which will have zero importance and can be trivially rejected. This means that there is often no reason not to put all representations of the model in the RIB file of every frame.

However, in those situations in which it is undesirable for any of the too-complex representations to exist in the RIB file (for example, when RIB file size or RIB file generation time are serious issues), this can be ameliorated by combining level-of-detail with procedural primitives. When used in combination with the procedural primitive functions, the detailed object descriptions need never be loaded, or even created, if they aren't used. As an example, suppose we have three versions of a particular object, stored in three rib files named `small.rib`, `medium.rib` and `big.rib` that represent the underlying object at various levels of detail.

```
AttributeBegin                # push the graphics state
Detail [-1 1 -1 1 -1 1]      # setup a "ruler" for detail calcs
DetailRange [0 0 1 4]
Procedural "DelayedReadArchive" [ "small.rib" ] [-1 1 -1 1 -1 1]
```

```

DetailRange [1 4 256 400]
Procedural "DelayedReadArchive" [ "medium.rib" ] [-1 1 -1 1 -1 1]
DetailRange [256 400 1e38 1e38]
Procedural "DelayedReadArchive" [ "big.rib" ] [-1 1 -1 1 -1 1]
AttributeEnd

```

The Procedural "DelayedReadArchive" geometric primitive acts as a placeholder. The corresponding rib file is only read when the bounding box given is determined to be potentially visible. If the box is determined to be outside the current detail range (or otherwise invisible), it will never be read at all. This means that you can create RIB files which include large amounts of geometry without knowing for sure what level of detail they will be drawn.

Similarly, one could create a modeling system which created RIB representations of objects "on demand", where Procedural "RunProgram" primitives are executed to generate the RIB only when those primitives are discovered to have some relevant importance in the current frame.

Simulated Ray Tracing

Everyone knows that *PhotoRealistic RenderMan* is not a ray tracer. *BMRT* is a ray tracer, and for scenes that absolutely positively must have ray tracing in them, using *BMRT* is an excellent choice. For scenes that don't particularly need exact ray tracing, but do want to have reflection or refraction effects, there are always environment maps and reflection maps (for round and planar surfaces, respectively).

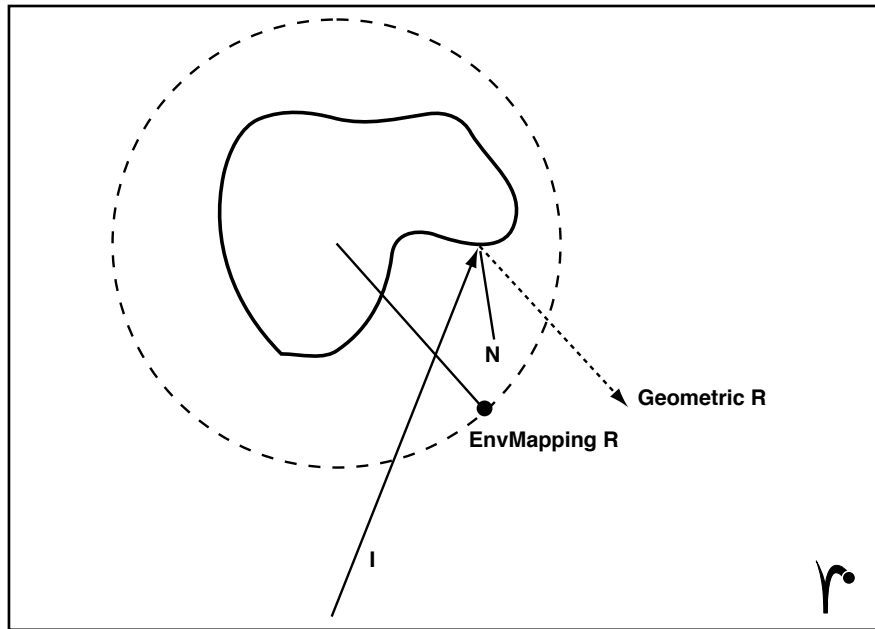
However, there are situations in the middle, where environment maps don't do a very good job, but ray tracing per se isn't really needed either. Some of those situations might include:

- Reflected objects are too close to be approximated by the "worldsphere" projection of environment maps;
- Slightly curved objects look unconvincing with either environment maps or reflection maps;
- We need CG reflections of objects that can't be modeled, such as live-action plates;
- Complex self-reflection can't be captured by a single central projection.

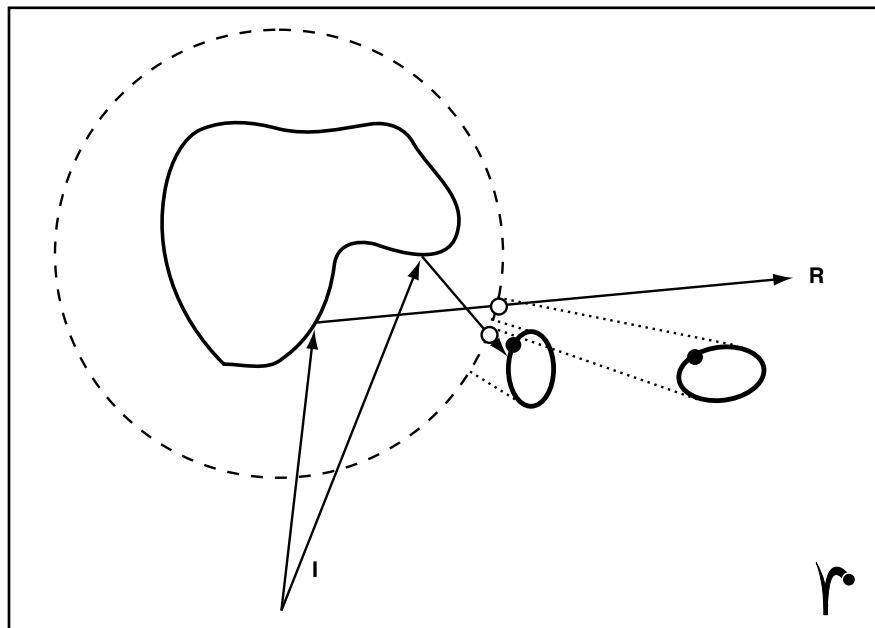
In these situations, we can resort to a concept that is known as "ray tracing in the shader". The idea is deceptively simple. Approximate the world to be reflected by a small number of objects. Objects for which the shader writer can easily write an analytical ray-object intersection test *in Shading Language*. For example, a couple spheres, a cube, or some planes floating in space. Then put texture maps on these objects. The shader fires rays at the objects, computes the appropriate texture coordinates of each hit, and looks up the texture using that texture coordinate.

See? Simple! There are a couple subtle points, however.

First, consider the following diagram which shows a reflection sphere, centered around and seen by some object. In the environment mapping case, the lookup uses direction only query the texture, which is identical to shooting the reflection ray from the center of the sphere (where the "reflection camera" lies). As we all know, the environment mapping case will look fine if the objects in the texture map are supposed to look "far away" (ideally, at infinity, but practically, some distance very much larger than the size of the object).



In the traced case, the true reflection ray is fired which hits the sphere at some entirely different location. It should be easy to see that the traced case will look fine if the objects in the texture map are local, and are very close to the position of the sphere itself. If there is very much distance discrepancy, the illusion will falter. In particular, if there are multiple objects in the texture which are supposed to be any significant distance apart, the lack of parallax on the reflecting surface will belie the local nature of the reflection lookup.



This parallax issue can be solved by having several layers of traced spheres, or even having free-floating polygons populating the traced environment, with pictures of objects on each one. In this case, each reflection map should have an alpha channel, so that the code can do a reasonable composite of the layers. For reference, this was the technique that was used on *The Abyss* and later on *Terminator 2*, to get the appearance of local reflections of live-action objects onto the CG objects.

An interesting twist on this scheme is to run the calculation backwards. That is, instead of having the surface shader probe the environment with outgoing rays, have something else fire rays from each little reflection object, towards the surface. That “something” is, of course, a light source which specifically sends photons along the reflection direction, and only contributes to the specular calculation (a non-diffuse *broad solar* light). The advantage of this scheme is that it modularizes the calculation. Each reflection light only needs to know how to project itself, rather than have every surface shader contain 5 different ray-object intersection subroutines; and you can easily add or delete reflection objects at runtime, rather than at surface shader compile time. The disadvantages are that layering is, well, tricky; and the resulting reflection photons come into the surface through the specular call instead of perhaps having a special path. The sufficiently motivated shader writer can get around these limitations with some work — an exercise left to the reader.

Clipping-Plane Shaders

One problem with trying to get special effects out of *PhotoRealistic RenderMan* is that it only evaluates shaders where there is geometry. If you want a pixel to have some color, but there is no geometry in that pixel, then you have to resort to compositing or image processing with an image made another way. *BMRT* somewhat ameliorates this issue by providing Imager shaders, but the semantics for imagers is pretty weak and the possibilities haven’t really been explored very well.

Nevertheless, we want to have pixels with interesting colors that are not limited to whether there is geometry in them. In particular, artists want lighting effects which bleed off of objects onto adjacent pixels, such as glows, and lighting effects which (in reality) happen entirely inside the camera, such as lens flares. Everybody *loves* lens flares.

There is actually a quite simple way to solve this, and many other similar problems which can be categorized as needing to run a shader in every pixel, but in front of (or behind) all objects in the scene. Effects which literally happen inside the camera lens or inside the eye of the viewer are prime examples. The solution is to have a piece of geometry which covers the screen, is generally clear, and is in front of every other object. There is one place in the scene which is guaranteed to satisfy this requirement is literally on the near clipping plane.

It is quite easy to place a polygon on the near clipping plane, using the `RiCoordSysTransform` function described earlier. To wit:

```
AttributeBegin
ShadingRate 1.0
CoordSysTransform "NDC"
Polygon "P" [0 0 0 1 0 0 1 1 0 0 1 0]
AttributeEnd
```

In NDC space, the entire image runs from 0 to 1, left to right and top to bottom, with near at $z = 0$ and far at $z = 1$. Therefore, this polygon will cover the entire image on the near clipping plane, regardless of resolution or frame aspect ratio.

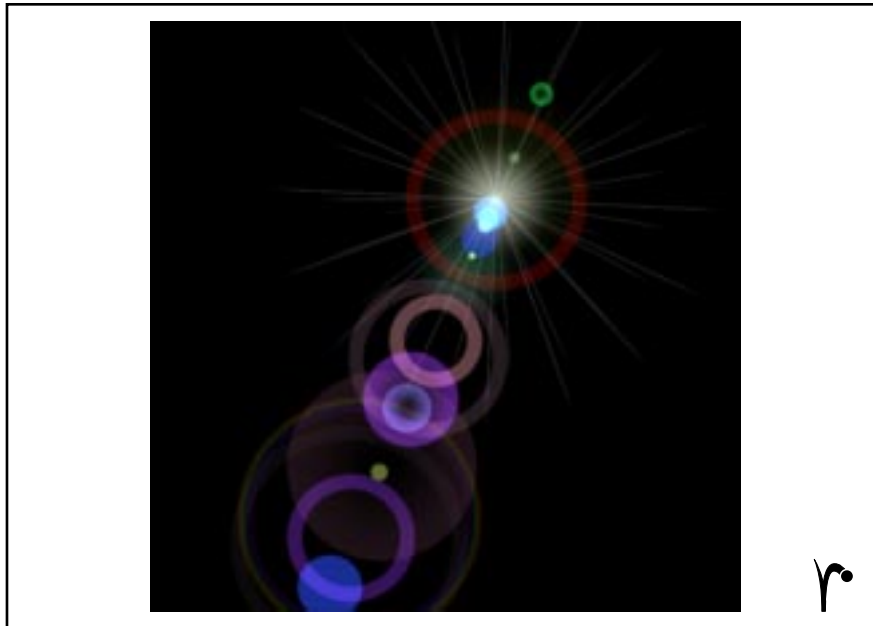
What to do with this bounty? Consider the following shader which places a glowing disk around every light source in the scene.

```

surface glow ( float intensity = 0.5, radius = 0.1; ) {
    point Plight, Pndc;
    float dist;
    Ci = 0;
    Oi = 0;
    Pndc = transform("NDC", P);
    setzcomp(Pndc, 0.0);
    illuminance (P, vector "camera" (0,0,1), PI) {
        Plight = transform("NDC", P+L);
        setzcomp(Plight, 0.0);
        dist = length(Plight - Pndc);
        Ci += (1 - smoothstep(0, radius, dist)) * intensity * Cl;
    }
}

```

This measures the distance from every point on the nearplane object to each light source, measured in NDC units, and puts color in points within a particular radius. The object is transparent everywhere, so the glow is added to every pixel, but doesn't obscure anything that's already there. With a little more work, lens flares can be implemented the same way.



Shading Reference Positions

In computer animation, particularly when done using modern animation tools, it is extremely common for characters to deform. Long gone are the days when objects' shapes were rigid and static. It is also extremely common for objects to be shaded using texture patterns that are intrinsically 3D in

nature (solid noise functions, projected textures, etc.). It must be obvious that these two techniques are fundamentally inconsistent with each other. If the object's shape is changing in 3D, then each frame will have different 3D coordinates to shade, and will get separate texture patterns.

This is solved very simply in RenderMan. Since RenderMan permits users to attach arbitrary variables to primitives, one trick that was discovered very early was to attach 3D texture coordinates to the primitives instead of the usual 2D ones. The 3D noise patterns and projected textures would be looked up using this 3D texture coordinate, which was always the same for any given parametric position on the primitive, regardless of the deformations and transformations that the underlying patch geometry underwent. These 3D coordinates were commonly called “sticky P”, because they made the 3D textures stick to the surface. They would be interpolated using the standard *varying* (parametric bilinear) interpolation on the surface, which was not quite identical to the actual surface itself, but that didn't matter very much.

In the modern RenderMan syntax, we can do even better, since can now apply a *vertex* variable to the surface. This will interpolate with the same patch basis as the underlying geometry. This gives us a 3D position that we can texture with that is exactly identical to the “rest” or “reference” position of the character. What this means, of course, is that every primitive is now generally twice the size in the RIB file, each frame containing the reference position “*Pref*” for texturing, and the deformed position “*P*” for illumination and standard hidden surface evaluation.

But what if...

...we used “*Pref*” for both texturing and illumination, and only used “*P*” for hidden surface evaluation. Then the illumination would stick to the surface from as it appeared in the reference pose, and the object would carry it's illumination around as it deformed. That could look interesting. Or, ...

...“*P*” is, in fact, nothing at all like a deformed pose of the character. Say “*P*” is actually a camera-aligned square, which is placed carefully to fill up the entire viewing frustum. We could generate the texturing and illumination on “*Pref*” in 3D, and write that out into a flat image. One that is appropriate as a texture map. Which we could use later to texture map onto other versions of the geometry, perhaps from other frames in the animation, perhaps simplified level-of-detail versions of the geometry. Or, ...

...we used several reference poses, a few for computing different layers of texture, one for illumination, a couple for shadowing....

Optimizing Area Operator Usage

The Shading Language provides several control-flow constructs, such as conditionals and loops. The existence of these constructs means that some points on the surface will execute a different set of shader instructions than other points on the surface, and that some expressions (and hence some local variables) are therefore not calculated at every point on the surface. Advanced Shading Language programmers know that, for this reason, they must avoid calling certain functions inside *varying* conditional statements. These functions, known generically as *area operators*, are those functions which use information at neighboring shading points in order to compute values.

For example, in order to compute a normal at a point *p0*, the function *calculatenormal* must implicitly examine two adjacent points *p1* and *p2* on the shading grid, in order to estimate the tangents and thereby compute the normal. If, due to the conditional, the value of the points *p1* or *p2* was not computed, then the tangents cannot be generated and the resulting normal at *p0* will be garbage.

The standard way around this dilemma is to compute all such functions outside any conditionals, and then use the conditional to control the application of the result rather than to control the computation of the value. For example, consider the following shader snippet

```
N2 = calculatenormal(transform("shader",P));
if (s > 0.5) {
    N2 = normalize(N2) * amplitude;
    P += ntransform("shader","current",N2);
}
```

Here the `calculatenormal` and its arguments are computed everywhere, and the result is applied only in the area of interest.

However, clearly this call to `calculatenormal` is wasted if the entire grid fails the conditional. In some shaders, the amount of wasted calculation can be quite large, particularly when texture maps are involved. We'd like a way to know ahead of time whether we are ever going to need those values.

Some parallel programming languages have constructs known as `if-all` and `if-any`. They help those languages optimize out wasted computations such as these, as well as manage other dataflow and synchronization issues. However, even though the Shading Language is clearly a parallel programming language, it doesn't have such constructs. Or does it?

Consider the following Shading Language code:

```
uniform float flag = 0;
if (s > 0.5) flag = 1;
if (flag) {
    N2 = calculatenormal(transform("shader",P));
    if (s > 0.5) {
        N2 = normalize(N2) * amplitude;
        P += ntransform("shader","current",N2);
    }
}
```

This is an example of the type of egregious hack known as a *device*. The uniform counter `flag` is set to one if any of the points which are being shaded in parallel pass the varying test. Thus we know if we need to do the expensive calculation, or if we can skip it entirely for this subsection of the primitive being shaded. Because I invented this construct in 1990, it is officially the "Apodaca Device".

Renderer-Driven Modeling

The renderer is a very powerful geometric computation engine. In it can be found half a million lines of code which computes matrix algebra, geometric clipping, arbitrary basis bicubic and NURB curve and patch evaluation, depth sorting, raster size estimation, stochastic sampling of scalar fields, hidden surface evaluation, image file resampling and filtering, noise functions, homogeneous matrix transformations, as well as a not insignificant C-like language interpreter. There is a huge amount of code implementing the heart of tens of years of graphics research which has been run trillions of times, and so it is pretty well debugged. Isn't it a shame that it takes 2 weeks to write and debug a B-spline patch evaluator, when there is one more general than you'll ever need, sitting right there inside that huge hunk of code called the "renderer". Hmm....

There's an old adage which states, "If all you have is a hammer, everything looks like a nail." The inverse adage never gets much press. "If you have is a bucket full of nails, you'll do just fine with one big hammer." RenderMan is one heck of a big hammer.

Consider the following problems. We've got a human character, who is supposed to have a handful of whiskers on his chin. The whiskers sprout out of hair follicles, which we can plainly see in a close-up shot of his face. The whiskers need to be modeled as `Curve` primitives, but the follicles want to be displacement maps generated from a procedural noise field applied to the face reference geometry. How are we going to get the curves to sprout from the follicles when their positions aren't known until the image is shaded?

Or, we have a particle system which is firing particles at a large, amorphous figure modeled out of NURBs with trim curves and is procedurally deformed by the modeler. The particles need to bounce off of the figure (and naturally emit collision particles), but writing a particle/deformed-trimmed-nurb collision routine is a one-month project. And we've got 2 days to make the shot work.

Or, we have hundreds of characters running around in our scene with expensive procedurally-enhanced secondary animation. We'd like to stop executing the dynamics code as soon as the characters and their shadows are no longer visible on-screen. How can we automate this tedious and error-prone process?

Interestingly, the renderer knows (or can know) the answers to these questions, you just need to ask the renderer to spit them out. It knows the position of the center of the procedurally defined follicle (and the surface normal at that position). It can determine when a particle is inside or outside the NURB object, by direct evaluation or by depth comparisons (depending on algorithm). It can tell if any characters are visible from the main camera or any of the shader cameras.

Using renderer prepasses to generate data that is used to shade beauty passes is commonplace. Shadow maps, computed cube-face environment maps, etc. are examples of this. However, it is rare for people to try the even more powerful trick of using the renderer prepasses to generate data that is used to procedurally *model* and *animate* a scene. Simply run the renderer in a prepass to generate modeling data, create a final model from that data, and feed the resulting model back into the renderer for image generation.

The renderer is highly programmable, both in the flexibility of its geometric language, the ease of use of its interpreted Shading Language, as well as the availability of DSOs for both geometry and shading calculations. It is surprisingly easy to create models and shaders that represent abstract concepts such as a force field or the magnitudes of displacement on a character. And it has several ways of emitting information, such as outputting arbitrary computed variables into image files, calling `printf` from the Shading Language, or sending data to a DSO which processes it and logs it to a file. Clever use of object naming, color coding, or just plain piping of renderer printing streams into `sed` scripts can give you access to the results of this powerful computation engine.

This, I think, is the next great frontier for RenderMan hackery. Long live *The Art of Chi' Ting*!

Basic Antialiasing in Shading Language

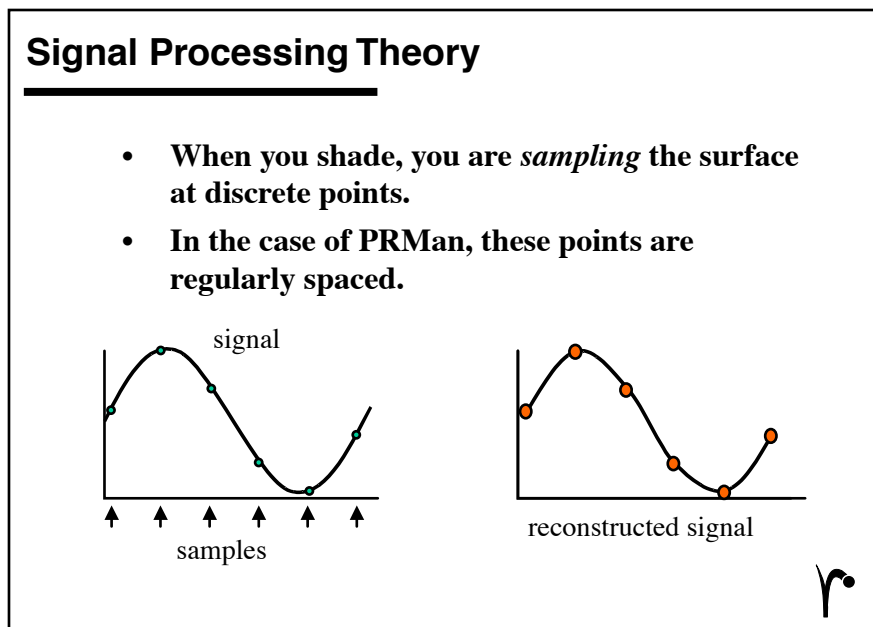
Larry Gritz
Pixar Animation Studios
lg@pixar.com

Abstract: Everybody has experienced it: sharp jaggies, pixelation artifacts, creepy-crawlies as the camera moves, horrible twinkling, or just plain weirdness when you pull back from your favorite procedural texture. It's aliasing, and it's a fact of life when writing shaders.

This talk is designed to make the problem seem less scary, to give you the conceptual tools to attack the problem on your own. It's not the be-all of antialiasing techniques, but it should get you started.

Introduction: Sources of Aliasing in Shading

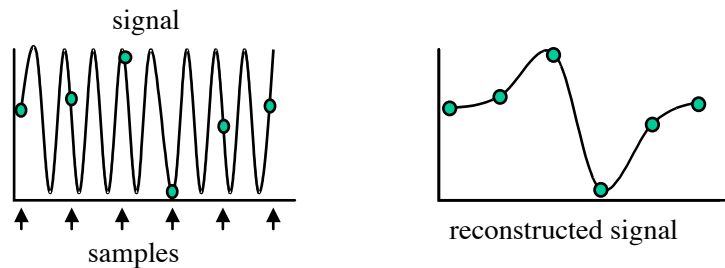
What is aliasing, really? The heart of the problem lies in the domain of signal processing. Fundamentally, our conceptual model is that there is a continuous *image function* in front of the camera (or alternately, impinging on the film plane). But we want discrete pixels in our image, so we must somehow *sample* and *reconstruct* the image function. Conceptually, we want the pixel value to represent some kind of average measure of the image function in the area “behind” the entire pixel. But renderers aren't terribly good at computing that directly, so they tend to use *point sampling* to measure the scene at a finite number of places, then use that information to reconstruct an estimate of the image function. When the samples are spaced much closer than the distance between the “wiggles” of the image function, this reconstructed signal is an accurate representation of the real image function.



But when the samples aren't spaced closely enough, the trouble begins.

More sampling

- But what if the signal gets to be higher frequency?



r

As illustrated by the previous slide, if the wiggles in the signal are spaced too closely (compared to the spacing of the samples), the wrong signal will be reconstructed. Specifically, highest frequency that can be adequately sampled is *half* the frequency of the samples themselves. This is known as the *Nyquist* limit. To put it another way, the samples must be at least twice the highest frequency present in the signal, or it will not adequately sample the signal.

What happened?

- If the frequency of samples is not at least *twice* the highest frequency of the signal, it will not be reconstructed properly.
- Technically, the energy appears in, or *aliases to*, lower frequencies.
- **Nyquist Limit:** the highest signal frequency which can be adequately sampled and reconstructed == $1/2$ the sampling frequency.

r

The renderer executes the shader at discrete points on the surfaces, and this is a sampling problem. In fact, for *PRMan* in particular, the sampling of the shaders happens at *regular intervals* in parameter space. Regular samples are very prone to aliasing. In screen space, most renderers avoid aliasing by sampling geometry in a jittered fashion, rather than with regularly spaced samples. Jittering the samples replaces aliasing with noise, usually regarded as a less objectionable artifact (this is strictly a perceptual effect, a by-product of our visual systems).

The problem with shading

- **Shaders are calculated at (potentially regular) intervals in parametric space, loosely coupled to raster units.**
- **Aliasing in screen space (e.g., for geometric edges) is handled by *stochastic sampling*, but grids are shaded at regular intervals.**
- **How else can we avoid aliasing? If we could *prefilter* the color of the surface, then the samples can be reconstructed.**

There are potentially two different sampling processes going on: the screen space samples and the shading samples in parameter space on the surface (for *PRMan*). Either one could cause aliasing if it is sampling a signal with frequencies above the Nyquist limit. For screen space samples, jittering replaces the aliasing with noise. This is a less objectionable artifact than aliasing, but it is still an artifact that can be avoided if we ensure that the image function itself has limited frequency content. But in *PRMan* the situation is worse, because by the time we are stochastically sampling in screen space, we have already taken *regularly spaced* samples in parametric space. In other words, the aliasing happens before the stochastic sampling can help. But even in a stochastic ray tracer, it would be better to be sampling an image function that didn't have frequencies beyond the Nyquist limit. We would like to *prefilter* the texture.

The key is realizing that the evils are all the result of point sampling the texture space function. Though we are making a single texture query, the reality is that this sample will be used to assign a color to an entire micropolygon or to color an entire pixel (or subpixel). Thus, our task is to estimate the *integral* of the texture function under the filter kernel that the sample represents. In other words, for a REYES-style micropolygon renderer, we want the average of the texture function underneath the entire micropolygon, rather than the exact value at one particular point on the micropolygon. For a ray tracer, we do not want to point sample the texture function, but rather we want the average value underneath the pixel that spawned the ray.

There are two trivial methods for estimating the average texture value under a region. First, you could use brute force to point sample several places underneath the filter kernel, and average those samples. This approach is poor because it merely replaces one point sample with many, which are still likely to alias, albeit at a higher frequency. Also, the cost of shading is multiplied by the number of samples that you take, yet the error decreases only as the *square root* of the number of samples.

The second trivial approach would be to generate the texture as an image map, a stored texture. RenderMan automatically antialiases texture lookups, so once the texture map is made (assuming that the generation of the texture map itself did not have aliasing artifacts), further use of the texture is guaranteed to antialias well with no need for you to worry about it in your shader. The problems with this approach mirror the downsides to texture mapping in general: (1) there is a fixed highest resolution that is evident if you are too close to the surface; (2) image texture maps have limited spatial extent and are prone to seams or obvious periodicity if they are tiled; (3) high resolution stored textures can eat up a lot of disk space.

So assuming that we want to antialias a truly procedural texture, and don't feel that a valid option is to subsample the texture by brute force, we are left searching for more clever options. Sections 3 and 4 will explain two such strategies: analytic solutions to the integral, and frequency clamping methods. But first, Section 2 will review the Shading Language facilities that are needed for these approaches.

Antialiasing shaders

- **Goal:** *prefilter* the texture.
- **Strategy:** rather than *point sampling* the texture function, return the *integral* of the function under the filter.
- **Approaches to antialiasing:**
 - Brute force: sample at a higher rate — not a good solution
 - Utilize the texture system, which antialiases automatically
 - Analytic/symbolic solutions to the integral
 - Simply avoid or fade out frequencies higher than Nyquist



Shading Language Facilities for Filter Estimation

We've now recast the problem as estimating the average value of the texture function over an area represented by a sample (either a micropolygon or a pixel – it hardly matters which from here on, so I will sometimes interchange the terms). Pictorially, we have some function f that we are sampling at location x . But while we are supplied a single x value, we really want to average f over the range of values that x will take on underneath the entire pixel with width w :

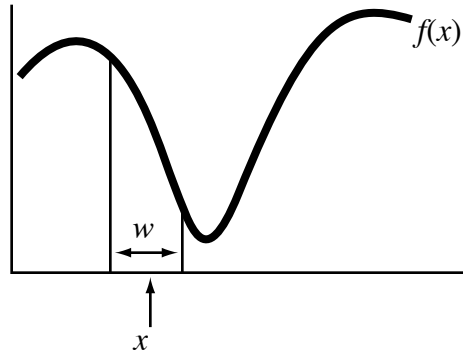


Figure 2.1. The pixel represented by this sample covers a width of w .

The trick is, how big should w be in order to cover a pixel? Luckily, Shading Language provides us with methods for estimating the filter size w . In particular, there are three useful built-in functions:

- **Du**(x) returns the derivative of arbitrary expression x with respect to surface parameter **u**.
- **Dv**(x) returns the derivative of arbitrary expression x with respect to surface parameter **v**.
- **Deriv**(x, p) returns the derivative of expression x with respect to a unit change in expression p .

Okay, maybe only the first two are useful.

So we can take a derivative of an expression with respect to the **u** and/or **v** parameters of the surface, i.e. we know the rate of change, per unit of **u** or **v**, of x , at the spot being shaded. How does this help us estimate how much x changes from pixel to pixel? Two more built-in variables come into play:

- **du** is the change in surface parameter **u** between adjacent samples being shaded.
- **dv** is the change in surface parameter **v** between adjacent samples being shaded.

So if **Du**(x) is the amount that x changes per unit change of **u**, and **du** is the amount that **u** changes between adjacent shading samples, then clearly the amount that x changes between adjacent shading samples (moving in the direction of changing **u**) is given by:

$$\mathbf{Du}(x) * \mathbf{du}$$

And similarly, the amount that x changes between adjacent shading samples (moving in the direction of the **v** parameter) is:

$$\mathbf{Dv}(x) * \mathbf{dv}$$

Not only can we take these derivatives explicitly, but there are also some other built-in Shading Language functions that implicitly make use of derivatives:

- **calculatenormal**(p) returns the cross product of the derivatives in each direction, i.e.

$$\mathbf{Du}(p) \wedge \mathbf{Dv}(p)$$

Derivatives in SL

- You can take the derivatives of *arbitrary expressions*:

$$\mathbf{Du}(x) \equiv dx/du$$

$$\mathbf{Dv}(x) \equiv dx/dv$$

$$\mathbf{Deriv}(x,p) \equiv dx/dp$$

- You also know the spacing between shading samples in the u and v directions: du , dv
- So the change in expression x in adjacent u and v samples is:

$$\mathbf{Du}(x)*du$$

$$\mathbf{Dv}(x)*dv$$



- area**(p) returns the (geometric) mean of the change of p between adjacent samples in each parametric direction,

$$\mathbf{length}(\mathbf{Du}(p)*du \wedge \mathbf{Dv}(p)*dv)$$

This is loosely interpreted as the area of the microfacet, if $p = \mathbf{P}$.

- texture**(filename, s,t) implicitly takes the derivatives of coordinates s and t to decide how large an area of texture to filter.

Functions based on derivatives

- Calculatenormal:**

$$\mathbf{calculatenormal}(p) \equiv \mathbf{Du}(p) \wedge \mathbf{Dv}(p)$$

- Area:**

$$\mathbf{area}(p) \equiv \mathbf{length}(\mathbf{Du}(p)*du \wedge \mathbf{Dv}(p)*dv)$$

- Texture:**

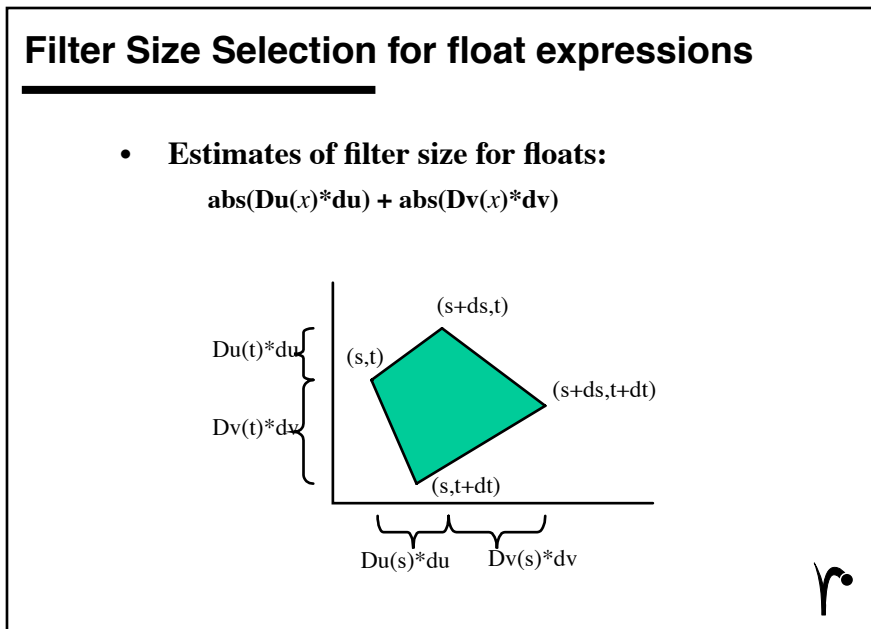
$$\begin{aligned} \mathbf{texture}(\text{file}, s, t) \equiv & \mathbf{texture}(\text{file}, s, t, \\ & s + \mathbf{Du}(s)*du, t + \mathbf{Du}(t)*du, \\ & s + \mathbf{Dv}(s)*dv, t + \mathbf{Dv}(t)*dv, \\ & s + \mathbf{Du}(s)*du + \mathbf{Dv}(s)*dv, t + \mathbf{Du}(t)*du + \mathbf{Dv}(t)*dv) \end{aligned}$$



How can we use this information to estimate filter size? Recalling the expressions above for estimating how much x changes as we move one sample over in either u or v , a fair estimate of the amount that x will change as we move one sample over in any direction might be:

$$\text{abs}(\mathbf{Du}(x) * \mathbf{du}) + \text{abs}(\mathbf{Dv}(x) * \mathbf{dv})$$

Informally, we are just summing the potential changes in each of the parametric directions. We take absolute values because we don't care if x is increasing or decreasing, we only are concerned with *how much* it changes between where we are and the neighboring sample. The geometric interpretation of this formula is shown in the following slide:



That formula works great if x is a **float** expression. But if x is a **point**, its derivative is a vector, and so the rest of the formula doesn't make much sense. But recall the definition of the built-in **area()** function:

$$\text{area}(p) = \text{length}(\mathbf{Du}(p) * \mathbf{du} \wedge \mathbf{Dv}(p) * \mathbf{dv})$$

The cross product of the two tangent vectors ($\mathbf{Du}(p) * \mathbf{du}$ and $\mathbf{Dv}(p) * \mathbf{dv}$) is another vector that is mutually perpendicular to the other two, and has length equal to the “area” of the parallelogram outlined by the tangents. Hence the name of the function. Remember that the length of a cross product is the product of the lengths of the vectors. So the square root of **area** is the geometric mean of the lengths of the tangent vectors. Thus, $\text{sqrt}(\text{area}(p))$ is a decent estimate of the amount that point p changes between adjacent samples (expressed as a scalar).

With this knowledge, we can write macros that are useful for filter width estimates for either float or point functions:

```
#define MINFILTERSIZE 1.0e-6
#define filtersize(x) max (abs(Du(x)*du)+abs(Dv(x)*dv), MINFILTERSIZE)
#define pfiltersize(p) max (sqrt(area(p)), MINFILTERSIZE)
```

The `filtersize` and `pfiltersize` macros can be used to estimate the change of its parameters from sample to sample, for float or point-like arguments, respectively. We impose a minimum filter size in order to avoid math exceptions if we divide by the filter size later. With these macros, we can move on to specific antialiasing techniques.

Filter size estimates for point expressions

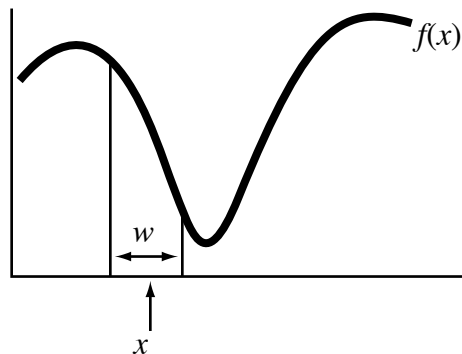
- **Remember that $\text{area}(p) = \text{length}(\mathbf{Du}(p) * \mathbf{du} \wedge \mathbf{Dv}(p) * \mathbf{dv})$**
- **If $\text{area}(p)$ is the area of the surface element that you're shading, then...**
- **$\sqrt{\text{area}(p)}$ is the amount that p changes between adjacent shading samples in either direction (sort of averaged)**
- **Summary of filter sizes, with macro examples:**

```
#define filtersize(x) max(abs(Du(x)*du)+abs(Dv(x)*dv), MIN)
#define pfiltersize(p) max(sqrt(area(p)), MIN)
```

r

Analytic Antialiasing

Recall our earlier figure illustrating the function f , which we are trying to sample at x . We really want some metric of the average value of f in the region surrounding x with width w :



More technically, we want the *convolution* of function f and some filter kernel k having support of width w . Convolution is defined as:

$$(f \otimes g)(x) = \int f(x)g(x + \delta)d\delta$$

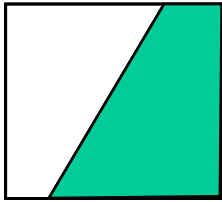
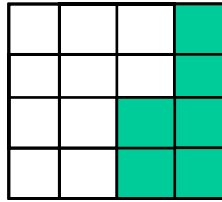
If we could use our knowledge of f to analytically (i.e. exactly, symbolically) derive the convolution $F = f \otimes k$, then we could replace all references to f in our shader with F , and that function would be guaranteed not to alias.

Example: Antialiasing a Step Function

As an example, consider the built-in `step(edge, val)` function. This function returns 0 when $val < edge$, and 1 when $val \geq edge$. If we make such a binary decision when we are assigning colors to a micropolygon, the entire micropolygon will be “on” or “off.” This will tend to produce “jaggies,” a form of aliasing, as shown in the following slide:

Antialiasing by analytic convolution

- Consider a step in abstract feature space:


shade->


- The binary decision of `step()` makes jaggies!

r

Can we construct a function, *filterstep*, which is the convolution of `step` with an appropriate filter kernel? Then we could use *filterstep* in place of `step` in our shaders and eliminate the resulting jaggies.

As usual for these problems, we must choose an appropriate filter kernel. For simplicity, let's just choose a box filter, because it is so easy to analyze. The adventurous reader may wish to derive a version of *filterstep* that uses, say, a Catmull-Rom filter.

Constructing a convolved step function

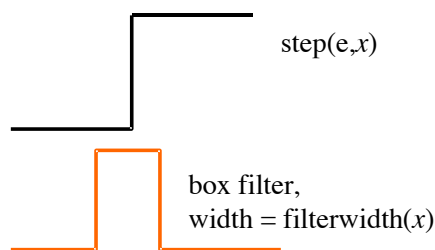
- What we'd really like instead of `step()` is a function that returns the convolution of `step()` with a filter.
- If we had such a function, we could use it in place of `step()`, using the filter size described earlier.
- For simplicity, let's use the example of a box filter (which is often good enough).
- Convolution: $(f \otimes g)(x) = \int f(x)g(x + \delta)d\delta$

r

An intuitive way of looking at this problem is: for a box filter of width w centered at value x , what is the convolution of the filter with `step(e,x)`?

Box filtering step()

- Can we derive a formula for
 $\text{filterstep}(e,x) = \text{step}(e,x) \otimes \text{box}(x,w)$?



r

This problem is easy to analyze. If $(x+w/2) < e$, then the result of `filterstep` is 0, because the box filter only overlaps the part of `step` that is zero. Similarly, if $(x-w/2) > e$, then `filterstep` should return 1, since the filter only overlaps the portion of `step` that is 1. If the filter overlaps the transition e , then `filterstep`

should return the fraction of the filter which is greater than e , in other words, $(x+w/2-e)/w$. As a further optimization, note that we can express this in Shading Language very compactly:

```
float filterstep (float edge, float x, float w)
{
    return clamp ((x+w/2-e)/w, 0, 1);
}
```

(Can you convince yourself that this is correct?) Of course, the w you supply to this function is properly the filter width returned by the macros we discussed earlier.

Note that we made a compromise for simplicity: we generated the antialiased version of **step** by convolving with a box filter. It would be better to use a Catmull-Rom, or other higher quality filter, and for a function as simple as **step**, that wouldn't be too hard. But for more difficult functions, it may be very tricky to come up with an analytic solution to the integral when convolved with a nontrivial kernel. So we can often get away with simplifying the integration by assuming a simple kernel like a box filter.

Constructing a homemade filterstep

$$\text{filterstep}(e,x,w) = \begin{cases} 0 & \text{if } x+w/2 < e \\ (x+w/2-e)/w & \text{if } x-w/2 < e \\ & \text{and } x+w/2 > e \\ 1 & \text{if } x-w/2 > e \end{cases}$$

- **More compactly:**
 $\text{filterstep}(e,x,w) = \text{clamp}((x+w/2-e)/w, 0, 1)$
- **You could use a more complex filter than box if you wanted to solve a harder integral.**
- **The same approach can be applied to other analytic functions.**

As an aside, there's now a built-in SL function, **filterstep**, which does exactly what we've described. It's been in PRMan since release 3.5, and has also been in the past few releases of BMRT. The syntax differs slightly from what we have described above. In particular, you don't need to supply the filter width – **filterstep** will automatically compute it, much as **texture** does with its arguments. Also, the built-in **filterstep** takes optional parameters allowing you to specify exactly which filter to use and how to modulate its width. Please see the PRMan user manual for details.

Aside: there's now a built-in filterstep

- **There is now a filterstep function,**
float filterstep (edge, x), filterstep (edge, x, ...)
- **The filterstep function takes derivatives of x.**
- **Default is Catmull-Rom filter, but you can use others.**
- **Other options:** “filter” “filtername” (“box”, etc.)
“width” filter-width-multiplier
- **Watch out that x is well defined everywhere!**

r

More Complex Examples

As another example, consider the function:

```
float pulse (float edge0, edge1, x)
{
    return step(edge0,x) - step(edge1,x);
}
```

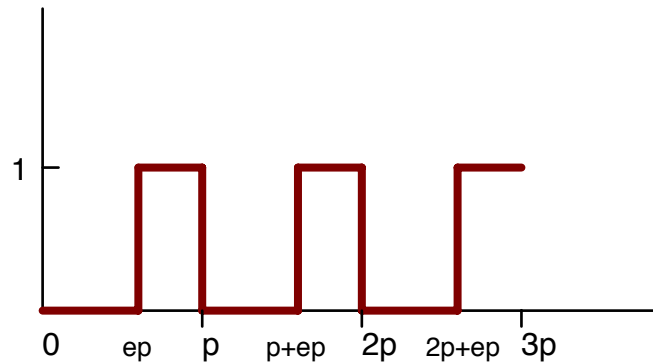
This is a useful function which returns 1 when $edge0 \leq x \leq edge1$, and 0 otherwise. Like **step**, it aliases horribly at any sampling rate, since it has infinite frequency content. And here is its antialiased version:

```
float filteredpulse (float edge0, edge1, x, fwidth)

{
    float x0 = x - fwidth/2;
    float x1 = x0 + fwidth;
    return max (0, (min(x1,edge1)-max(x0,edge0)) / fwidth);
}
```

It is left as an exercise for the reader to verify this derivation.

One last, less trivial, example of analytic antialiasing involves antialiasing a pulse train. Such a function is shown in the following figure:



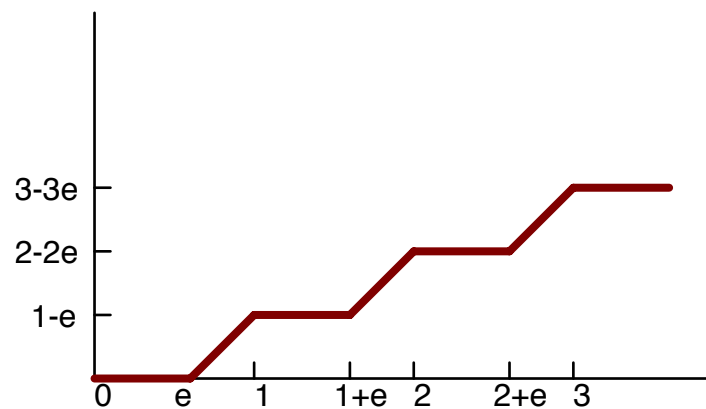
Here is this function expressed in Shading Language:

```
/* A pulse train: a signal that repeats with a given period, and is
 * 0 when 0 <= mod(x/period,1) < edge, and 1 when mod(x/period,1) > edge.
 */
float pulsetrain (float period, edge, x)
{
    return pulse (edge, 1, mod(x/period, 1));
}
```

Attacking this function is more difficult. Again, we will assume a box filter, which means that filteredpulsetrain(period,edge,x,w) is:

$$\frac{1}{w} \int_{(x-w/2)}^{(x+w/2)} \text{pulsetrain}(p,e,y) dy$$

This integral is actually not hard to solve. First, let's divide x and w by *period*, reducing to a simpler case where the period is 1. Here is a graph of the *accumulated* value of the function between 0 and x :



This function is given by $F(x) = (1-e)\text{floor}(x) + \max(0, x - \text{floor}(x) - e)$. Can you convince yourself that this is true? If the accumulated sum of $f(x)$ – in other words, the indefinite integral – is F , then the definite integral

$$\int_{x_0}^{x_1} f(x) = F(x_1) - F(x_0)$$

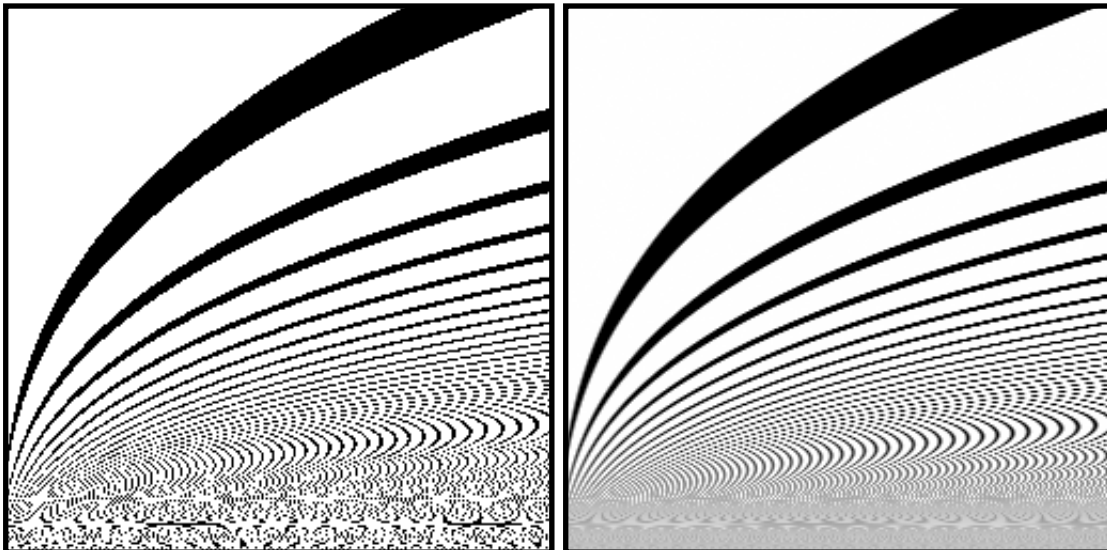
Therefore the following code implements a filtered pulse train:

```
/* Filtered pulse train:  it's not as simple as just returning the mod
 * of filteredpulse -- you have to take into account that the filter may
 * cover multiple pulses in the train.
 * Strategy: consider the function INTFPT, which is the integral of the
 * train from 0 to x.  Just subtract!
 */

float filteredpulsetrain (float period, edge, x, width)

{
    /* First, normalize so period == 1 and our domain of interest is > 0 */
    float w = width/period;
    float x0 = x/period - w/2;
    float x1 = x0+w;
    /* Now we want to integrate the normalized pulsetrain over [x0,x1] */
#define INTFPT(x) ((1-edge)*floor(x) + max(0,x-floor(x)-edge))
    return (INTFPT(x1) - INTFPT(x0)) / (x1-x0);
#undef INTFPT
}
```

This code works well even when the filter size is larger than the pulse period. Here is an example of a pattern based on pulsetrain (left), versus the filtered version using the function above (right):



Note that there is still some bad ringing. This is a result of our use of a box filter – it’s not very good at clamping frequencies. A better filter would have less ringing, but would have a more difficult integral to solve. Nonetheless, the most insidious aliasing is gone.

Antialiasing by Frequency Clamping

You will find that your favorite Shading Language functions fall into three categories: (1) functions for which you can trivially derive the convolution with a filter kernel, probably a box filter; (2) functions that are *really hard* to integrate analytically – lots of work, some integral tables, and maybe a session with Mathematica will help; and (3) functions which don’t have an analytic solution, or whose solution is just too hard to derive.

Many useful functions fall into the third category. Don’t worry, just try another strategy. In general, the next best thing to an analytic solution is frequency clamping. The basis of this approach is to decompose your function into composite functions with known frequency content, and then to only use the frequencies which are low enough to be below the Nyquist limit for the known filter size.

The general strategy is as follows. Suppose you have a function $g(x)$ that you wish to antialias, and your computed filter width is w . Suppose further that you know the following two facts about your function g :

1. Function g has features (the “wiggles” of the function) no smaller than width f .
2. The average value of g , over large ranges of values, is a .

We know that when $w \ll f/2$, we are sampling areas of g much smaller than its smallest wiggles, so we will not alias. If $w \gg f/2$, we are trying to sample at too high a frequency and we will alias. But we know the average value of g , so why not substitute this average value when the filter is too wide compared to the feature size? More formally,

```
#define fadeout(g,g_avg,featuresize,fwidth) \
    mix (g, g_avg, smoothstep(.2,.6,fwidth/featuresize))
```

This simple macro does just what we described. When the filter width is small compared to the feature size, it simply returns g . When the filter width is too large to adequately sample the function with that feature size, it returns the known average value of g . And in between, it fades smoothly between these values.

Another approach: Frequency Clamping

- Some functions are hard to antialias analytically.
- Instead, just don't compute (or fade out) their higher frequency components.
- Example: suppose you had function $g(x)$ with feature size f , and filter width w . Consider:

$$\text{mix}(g(x), g_avg, \text{smoothstep}(.25, .6, w/f))$$
- This yields $g(x)$ when the filter width is small compared to feature size, average of $g(x)$ when the filter width is big, and smoothly blends between them.



As an example, let's look at the **noise** function. We use it in shaders all the time. It will alias if there are fewer than two samples per cycle. We know that it has a limited frequency content – the frequency is approximately 1. We also know that its average value is 0.5.

But **noise** is often less useful than its cousin, **snoise** (for Signed noise). We usually define **snoise** as follows:

```
#define snoise(x) (2*noise(x)-1)
```

Regular noise has a range of (0,1), but **snoise** ranges on (-1,1) with an average value of 0. Consider the following macro:

```
#define filteredsnoise(x,w) fadeout(snoise(x), 0, 1, w)
```

This macro takes a domain value x , and a filter width w , and returns an (approximately) low pass filtered version of **snoise**. Okay, we all know it's not *really* low pass filtered noise – it fades uniformly to the average value of noise, just as the frequency gets high enough that it would tend to alias.

Poor Man's filterednoise

- Consider the following code fragment:

```
#define filterwidth(x) \
    max(MW, abs(Du(s)*du)+abs(Dv(x)*dv))
#define fadeout(x,avg,f,w) \
    mix(avg,x,smoothstep(.25,.75,w/f))
#define snoise(x) (2*noise(x)-1)
#define filteredsnoise(x,w) fadeout \
    (snoise(x),0,1,w)

float fwidth = filterwidth(x);
float n= filteredsnoise(x,fwidth);
```

- Can you convince yourself that this works and will not generate frequencies beyond the Nyquist limit?



We can extend this to make a filtered version of fractional Brownian motion – that's the technical term for summing successively higher octaves of noise at successively lower amplitudes.

Filtered fBm

- Consider:

```
float fBm (point p, float maxoctaves) {
    uniform float i;
    uniform float amp=1, freq=1;
    varying float fw = pfilterwidth(p);
    for (i = 0; i < maxoctaves; i += 1) {
        sum += amp*filterednoise (p*freq, fw);
        freq *= 2; amp *= 0.5; fw *= 2;
    }
    return sum;
}
```

- Watch out: fading to average is not low pass filtering!



As noted earlier, this isn't a truly correct filtered noise. Rather than correctly low pass filtering the noise function, we are simply omitting (or fading out) the octaves which are beyond the Nyquist limit.

it. This can lead to artifacts, particularly when the filter width is so large that even the lowest frequencies in the fBm will fade to their average value of 0 – in other words, the entire function will fade to a flat color with no detail.

Conclusions and Caveats

This chapter has outlined two methods for antialiasing your procedural shaders. Unfortunately, it's still a hard task. And these methods, even when they work fine, are not without their share of problems:

- While analytic integration yields exact antialiased results, only a few (generally simple) functions have analytic solutions that exist and are easy to derive.
- Frequency clamping and global fading are applicable to more complex functions, but they are not really the same as low pass filtering, and artifacts may result. (But those clamping artifacts are probably less objectionable than horrible aliasing.)
- Both methods can suffer from the *fallacy of composition*: for functions f and g and filtering kernel k , if you know the low-pass-filtered versions $F = f \otimes k$ and $G = g \otimes k$,

$$F(G(x)) \neq (f(g) \otimes k)(x) \quad \text{in general}$$

In other words, convolution does not necessarily hold across the composition of functions. What this means is that you cannot blindly replace all your **step** calls with **filterstep**, and your **noise** calls with **filternoise**, and expect thresholded noise to properly antialias.

- Previous versions of *PRMan* had **du** and **dv** estimates that were constant across each grid. That meant that any filter size estimates that depended on derivatives (as all the ones discussed in this paper do) potentially had discontinuities at grid boundaries. This was a particularly evil problem when dealing with displacements, as discontinuities of displacement amounts across grid boundaries can result in cracking. This derivative discontinuity issue was never a problem with BMRT, and is now fixed with *PRMan 3.8* (!), but it's an additional bugaboo to watch out for if you are using an earlier version of *PRMan*.
- We've been restricting our discussion to color, and displacement is even trickier. Consider a surface that is displaced with a noise field. If we use the filtered noise to antialias the bumps, what happens as we approach the Nyquist limit and the bumps fade down? A flat surface catches and reflects light much differently than a rough surface, so you may notice a light or color shift that you didn't want. This is a common problem when antialiasing bumps or displacements, and I have yet to see a satisfactory solution that works in the general case.

Problems with this approach

- **Because PRMan's du, dv are uniform across a grid, there can be discontinuities in the fading at grid boundaries. (But not a problem with BMRT or PRMan 3.8!)**
- **This is especially problematic for displacements, where this may cause cracking.**
- **Fading out bumps or displacements like this may also cause a shift in BRDF where none is desired.**
- **Composition of antialiased functions is tricky!**



These are only the basics, but they're the bread-and-butter of antialiasing, even for complex shaders. Watch out for high frequency content, conditionals, etc., in your shaders. Always try to calculate what's happening under a filter area, rather than at a single point. Don't be afraid to cheat – after all, this is computer graphics. And above all, have fun writing shaders!

Ray Tracing in PRMan*

(*) with a little help from BMRT

Larry Gritz
Pixar Animation Studios

Abstract. This document explains how to render scenes using *PRMan* with ray traced shadows and reflections, using BMRT as an “oracle” to provide answers to computations that *PRMan* cannot solve. We describe a method of actually stitching the two renderers together using a Unix pipe, allowing each renderer to perform the tasks that it is best at.

Introduction

PhotoRealistic RenderMan has a Shading Language function called `trace()`, but since there is no ability in *PRMan* to compute global visibility, the `trace()` function always returns 0.0 (black). This is no way to ask for any other global visibility information in *PRMan*. Though *PRMan* often can fake reflections and shadows with texture mapping, there are limitations:

- Environment mapped reflections are only “correct” from a single point. Environment mapping a large reflective object has errors (which, to be fair, are often very hard to spot). Mutually reflective objects are a big pain in *PRMan*.
- Environment and shadow maps require multiple rendering passes, and require TD time to set up properly.
- Dealing with shadow maps – selecting resolution, bias, blur, etc. – can be time consuming and still show artifacts in the shadows. Also, shadows cannot motion blur in *PRMan*, and cannot correctly handle opacity (or color) changes in the object casting a shadow.
- Refraction is nearly impossible to do correctly, since even when environment mapping is acceptable, *PRMan* cannot tell the direction that a ray *exits* a refractive object, since the “backside” is not available for ray tracing.

The *Blue Moon Rendering Tools* (BMRT) contains a renderer, *rendrib*, which is fully compliant with the RenderMan 3.1 specification and supports ray tracing, radiosity, area lights, volumes, etc. It can compute ray traced reflections, shadows, and so on, but is much slower than *PRMan* for geometry which doesn’t require these special features.

Both renderers share much of their input – by being RenderMan compliant, they both read the same geometry description (RIB) and shader source code files.¹ It’s tempting to want to combine the effects of the two renderers, using each for those effects that it achieves well. Several strategies come to mind:

1. Choosing one renderer or the other based on the project, sequence, or shot. Perhaps a strategy might be to use *PRMan* most of the time, BMRT if you need radiosity or ray tracing.

1. The compatibility is limited to areas dictated by the RMan spec. The two renderers each have different formats for stored texture maps and compiled shaders, and support different feature subsets of the spec.

2. Rendering different objects (or layered elements) with different renderers, then compositing them together to form final frames.
3. Rendering different *lighting* layers with different renderers, then adding them together. For example, one might render base color with PRMan, but do an “area light pass” (or radiosity, or whatever) in BMRT.

All of these approaches have difficulties (though all have been done). Strategy #1 may force you to choose a slow renderer for everything, just because you need a little ray tracing. There may also be problems matching the exact look from shot to shot, if you are liberally switching between the two renderers. Strategies #2 and #3 have potential problems with “registration,” or alignment, of the images computed by the renderers. Also, #3 can be very costly, as it involves renders with each renderer.

The attraction of using the two renderers together, exploiting the respective strengths of both programs while avoiding undue expense, is alluring. I have developed a method of literally stitching the two programs together.

Can we combine them?

- **Both read the same RIB, shaders; tempting to combine**
- **Choose one or the other for an animation, sequence, shot?**
- **Render different elements, comp together?**
- **Render different lighting, add together? (e.g., add a BMRT “area light pass” to a PRMan frame without the area light)**



Background: DSO Shadeops in PRMan

RenderMan Shading Language has always had a rich library of built-in functions (sometimes called “shadeops”), already known to the SL compiler and implemented as part of the runtime shader interpreter in the renderer. This built-in function library included math operations (sin, sqrt, etc.), vector and matrix operations, coordinate transformations, etc. It has also been possible to write SL functions in Shading Language itself (in the case of *PRMan*, this ability was greatly enhanced with the new compiler included with release 3.7). However, defining functions in SL itself has several limitations.

The newest release of *PRMan* (3.8) allows you to write new built-in SL functions in 'C' or 'C++'. Writing new shadeops in C and linking them as DSOs has many advantages over writing functions in SL, including:

- The resulting object code from a DSO shadeop is shared among all its uses in a renderer. In contrast, compiled shader function code is inlined every time the function is called, and thus is not shared among its uses, let alone among separate shaders that call the same function.
- DSO shadeops are compiled to optimized machine code, whereas shader functions are interpreted at runtime. While *PRMan* has a very efficient interpreter, it is definitely slower than native machine code.
- DSO shadeops can call library functions from the standard C library or from other third party libraries.
- Whereas functions implemented in SL are restricted to operations and data structures available in the Shading Language, DSO shadeops can do anything you might normally do in a C program. Examples include creating complex data structures or reading external files (other than textures and shadows). For example, implementing an alternative `noise()` function, which needs a stored table to be efficient, would be exceptionally difficult in SL, but very easy as a DSO shadeop.

DSO shadeops have several limitations that you should be aware of:

- DSO shadeops only have access to information passed to them as parameters. They have no knowledge of “global” shader variables such as `P`, parameters to the shader, or any other renderer state. If you need to access global variables or shader parameters or locals, you must pass them as parameters.
- DSO shadeops act as strictly point processes. They possess no knowledge of the topology of the surface, derivatives, or the nature of surface grids (in the case of a REYES renderer like *PRMan*). If you want to take derivatives, for example, you need to take them in the shader and pass them as parameters to your DSO shadeop.
- DSO shadeops cannot call other built-in shadeops or any other internal entry points to the renderer itself.

Further details about DSO shadeops, including exactly how to write them, are well beyond the scope of these course notes. For more information, please see the *RenderMan Toolkit 3.8 User Manual*.

New backdoor: PRMan 3.8 DSO shadeops

- **RenderMan renderers have lots of useful built-in shadeops**
- **SL compilers let you write new functions in SL.**
 - - Still limited to SL constructs
 - Still interpreted at runtime
- **PRMan 3.8 allows C compiled DSO shadeops**
 - - Compiled to machine code
 - - Can do anything that you could do from C
 - See PRMan 3.8 docs for details



How Much Can We Get Away With?

So *PRMan 3.8* has a magic backdoor to the shading system. One thing it's good for is to make certain common operations much faster, by compiling them to machine code. But it also has the ability to allow us to write functions which would not be expressible in SL at all – for example, file I/O, process control or system calls, constructing complex data structures, etc.

How far can we push this idea? Is there some implementation of `trace()` that we can write as a DSO which will work? Yes! The central idea is to render using *PRMan*, but implement `trace` as a call to BMRT. In this sense, we would be using BMRT as an oracle, or a *ray server*, that could answer the questions that *PRMan* needs help with, but let *PRMan* do the rest of the hard work.

BMRT (release 2.3.6 and later) has a ray server mode, triggered by the command line option `-rayserver`. When in this mode, instead of rendering the frame and writing an image file, BMRT reads the scene file but it just waits for “ray queries” to come over `stdin`. When such queries (specified by a ray server protocol) are received, BMRT computes the results of the query, and returns the value by sending data over `stdout`.

The *PRMan* side is a DSO which, when called, runs *rendrib* and opens a pipe to its process. Thereafter, calls to the new functions make ray queries over the pipe, then wait for the results.

Sneaky Idea: "Ray Server"

- **PRMan's implementation of trace() returns 0, since it cannot ray trace.**
- **Can we write a DSO that implements trace()?**
- **Idea:**
 - **Launch BMRT with a pipe, have it read the same scene**
 - **Don't have BMRT render the picture, just wait for queries**
 - **PRMan renders, but asks the "ray server" when it needs the results of a trace**



New Functionality

This hybrid scheme effectively adds three new functions that you can call from your shaders:

`color trace (point from, vector dir)`

Returns the incoming ray-traced irradiance at point **from** looking in direction **dir**.

`color visibility (point p1, point p2)`

Returns the spectral visibility (1 for completely visible, 0 for completely opaque) for the space between arbitrary points **p1** and **p2**.

`float rayhittest (point from, vector dir,
output point Ph, output vector Nh)`

Probes geometry **from** point from looking in direction **dir**. If no geometry is hit by the ray probe, the return value will be very large (1e38). If geometry is encountered, the position and normal of the geometry hit will be stored in **Ph** and **Nh**, respectively, and the return value will be the distance to the geometry.

Yes, This Actually Works

- **BMRT 2.3.6 is modified to have a "ray server mode"**
- **The PRMan side is implemented as a DSO shadeop (source code included in these course notes)**
- **New functionality:**

```
color trace (point from; vector direction)
color visibility (point p1,p2)
float rayhittest (point from; vector
                  direction; output point Phit;
                  output vector Nhit)
```



How to use it

Using *PRMan* as a ray tracer is straightforward:

1. Use these functions in your shaders. In any shader that uses the functions, you should:

```
#include "rayserver.h"
```

If you inspect `rayserver.h` (in the appendix of this chapter), you'll see that the functions described above are really macros. When compiling with BMRT's compiler, the functions are unchanged (all three are actually implemented in BMRT). But when compiling with PRMan's compiler, the macros transform their arguments to world space and call a function called `rayserver()`.

2. Compile the shaders with both BMRT and PRMan's shader compilers. When compiling for PRMan, make sure that the DSO `rayserver.so` is in your include path.
3. Set the environment variable `RAYSERVER` to `"rendrib -rayserver ribfiles"`. The full path to *rendrib* must be specified. The *ribfiles* is the list of RIB files that contain the geometry that you want ray traced. It may be the list of RIB files that you specify to *PRMan*, but it doesn't have to be the same. For example, you may want different options set for the BMRT rendering, or you may want to have the ray tracer know about only a subset of the geometry in the scene, etc. For example, if you are rendering a file called `teapots.rib`, you should:

```
setenv RAYSERVER "rendrib -rayserver teapots.rib"
```

4. Go ahead and render the scene with *PRMan*. For example,

```
prman teapots.rib
```

That's it!

If you're really lazy, here's a script that easily handles the special case of using the exact same RIB files for both renderers:

```
#!/bin/sh
export RAYSERVER="rendrib -rayserver $*"
prman $*
```

Results

Here is an example image rendered with PRMan, unassisted (color version is Plate 7):



The teapot on the right makes calls to `trace()` for reflections, but of course this does nothing in *PRMan*.²

On the next page is the *same* RIB file, with the same shaders, taking advantage of the ray server described above. (color version is Plate 8) ***This picture was rendered by PRMan (mostly), without using shadow or environment maps.***

Pros and Cons

The big advantage here is that you can render most of your scene with *PRMan*, using BMRT for tracing individual rays on selected objects or calculating shadows for selected lights. This is much faster than rendering in BMRT, particularly if you only tell the ray tracer about a subset of the scene that

2. For your viewing pleasure, color versions of these images can be found in the color plates, and in the *gritz* directory on the SIGGRAPH CD.



you want in the shadows or reflections. The following effects are utterly trivial to produce with this scheme:

- Ray cast shadows, including shadows that correctly respond to color and opacity of occluding objects. Moving objects can cast correct motion-blurred shadows.
- Correct reflections, including motion blur.
- Real refraction for glass, water, etc.
- No setup time or multi-pass rendering for these effects.

The big disadvantage is that it requires *two* renderers to both have the scene loaded at the same time. This can be alleviated somewhat by reducing the scene that the ray tracer sees, or by telling the ray tracer to use a significantly reduced tessellation rate, etc. But still, it's a significant memory hit compared to running PRMan alone.

All of the usual considerations about compatibility between the two renderers apply. Be particularly aware of new PRMan primitives and SL features not currently supported by BMRT, texture file format differences, results of noise() functions, etc.

Tradeoffs

Pros:

- **PRMan renders most of the scene (fast!)**
- **BMRT can reflect, refract, shadow with ray casting**
- **Maybe ways to extend the protocol for radiosity, area lights, volumes**

Cons:

- **Two renderers at once, each has its own copy of the scene database!**
- **Possible precision, biasing problems due to different scene representations.**
- **Incompatibilities (texture & shader formats, feature drift)**



Appendix A: rayserver.h

```
/* rayserver.h - SL (PRMan side) include file for ray server.
 *
 * These macros translate calls to trace(), visibility(), and
 * rayhittest() into calls to rayserver(). Each generates different
 * numbers of args, allowing rayserver() to be polymorphic.
 *
 * It is assumed that the rayserver() function itself is implemented
 * as a DSO.
 *
 * Note that the ray server expects its data in world space.
 */

#ifndef BMRT

/* PRMan side only -- BMRT already knows these functions */

#define worldp(p) transform("world",p)
#define worldv(v) vtransform("world",v)

#define trace(p,d) rayserver(worldp(p), worldv(d))

#define visibility(from,to) rayserver(worldp(from), worldp(to))

#define rayhittest(p,d,phit,nhit) \
    rayserver(worldp(p), worldv(d), phit, nhit)

#endif
```

Appendix B: rayserver.c

```

/* rayserver.c - implements the rayserver protocol -- client side.
 * When one of the rayserver functions is called, open a pipe to a ray server
 * program, and transmit queries using a rayserver protocol.
 *
 * Compile me like this (on SGI):
 *   cc -c rayserver.c
 *   ld -shared -o rayserver.so rayserver.o
 */
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <alloca.h>

#include "shadeop.h"

SHADEOP_TABLE(rayserver) = {
    { "color rayserver_trace (point, vector)",
      "rayserver_init", "rayserver_cleanup" },
    { "color rayserver_visibility (point, point)",
      "rayserver_init", "rayserver_cleanup" },
    { "float rayserver_rayhittest (point, vector, point, vector)",
      "rayserver_init", "rayserver_cleanup" },
    { "" }
};

typedef struct {
    int initialized;
    FILE *in;
    int pipe[2];
} RS_DATA;

static RS_DATA rs_data = { 0 } ;

SHADEOP (rayserver_trace)
{
    float *result = (float *)argv[0];
    float *pos = (float *)argv[1];
    float *dir = (float *)argv[2];
    char buf[256];
    float fbuf[16];

```

```

    if (! rs_data.initialized) {
        result[0] = 0; result[1] = 0; result[2] = 0;
        return 0;
    }
    memset (fbuf, 0, 12*sizeof(float));
    buf[0] = 'b'; buf[1] = 't';
    write (rs_data.pipe[1], buf, 2);
    memcpy (fbuf, pos, 3*sizeof(float));
    memcpy (fbuf+3, dir, 3*sizeof(float));
    fbuf[6] = 0.0f; /* time */
    fbuf[7] = 0.5f; /* detail */
    /* 8-11 are reserved */
    write (rs_data.pipe[1], fbuf, 12*sizeof(float));
    read (rs_data.pipe[0], result, 3*sizeof(float));
    read (rs_data.pipe[0], fbuf, sizeof(float));
    return 0;
}

SHADEOP (rayserver_visibility)
{
    float *result = (float *)argv[0];
    float *pos0 = (float *)argv[1];
    float *pos1 = (float *)argv[2];
    char buf[256];
    float fbuf[16];

    if (! rs_data.initialized) {
        result[0] = 0; result[1] = 0; result[2] = 0;
        return 0;
    }
    memset (fbuf, 0, 12*sizeof(float));
    buf[0] = 'b'; buf[1] = 'v';
    write (rs_data.pipe[1], buf, 2);
    memcpy (fbuf, pos0, 3*sizeof(float));
    memcpy (fbuf+3, pos1, 3*sizeof(float));
    fbuf[6] = 0.0f; /* time */
    fbuf[7] = 0.5f; /* detail */
    /* 8-11 are reserved */
    write (rs_data.pipe[1], fbuf, 12*sizeof(float));
    read (rs_data.pipe[0], result, 3*sizeof(float));
    return 0;
}

SHADEOP (rayserver_rayhittest)
{
    float *result = (float *)argv[0];
    float *pos = (float *)argv[1];
    float *dir = (float *)argv[2];
    float *Phit = (float *)argv[3];

```

```

float *Nhit = (float *)argv[4];
char buf[256];
float fbuf[12];

if (! rs_data.initialized) {
    result[0] = 0; result[1] = 0; result[2] = 0;
    return 0;
}
memset (fbuf, 0, 12*sizeof(float));
buf[0] = 'b'; buf[1] = 'h';
write (rs_data.pipe[1], buf, 2);
memcpy (fbuf, pos, 3*sizeof(float));
memcpy (fbuf+3, dir, 3*sizeof(float));
fbuf[6] = 0.0f; /* time */
fbuf[7] = 0.5f; /* detail */
/* 8-11 are reserved */
write (rs_data.pipe[1], fbuf, 12*sizeof(float));
read (rs_data.pipe[0], result, 1*sizeof(float));
read (rs_data.pipe[0], Phit, 3*sizeof(float));
read (rs_data.pipe[0], Nhit, 3*sizeof(float));
return 0;
}

SHADEOP_INIT (rayserver_init)
{
    int popen2(char *, int[2]);
    char *servername;
    int len, i;
    if (rs_data.initialized)
        return NULL;

    servername = getenv ("RAYSERVER");
    if (! servername)
        return NULL; /* Could not find name of server */

    len = strlen (servername) + 2;
    if (popen2 (servername, rs_data.pipe)) {
        fprintf (stderr, "Error opening pipe to \"%s\"\n", servername);
        return NULL; /* Could not open pipe */
    }

    rs_data.in = fdopen (rs_data.pipe[0], "r");
    setlinebuf (rs_data.in);
    rs_data.initialized = 1;
    return NULL;
}

```

```

SHADEOP_CLEANUP (rayserver_cleanup)
{
    if (rs_data.initialized) {
        close (rs_data.pipe[0]);
        close (rs_data.pipe[1]);
        rs_data.initialized = 0;
    }
}

static void sig_pipe (int signo)
{
    fprintf (stderr, "SIGPIPE caught\n");
    exit(1);
}

static int popen2 (char *cmd, int fd[2])
{
    int fd1[2], fd2[2], pid;

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR) {
        fprintf (stderr, "signal error\n");
        return -1;
    }
    if (pipe(fd1) < 0 || pipe(fd2) < 0) {
        fprintf (stderr, "pipe error\n");
        return -1;
    }
    if ((pid = fork()) < 0) {
        fprintf (stderr, "fork error\n");
        return -1;
    } else if (pid > 0) {
        /* Parent */
        close (fd1[0]);
        close (fd2[1]);
        fd[0] = fd2[0];
        fd[1] = fd1[1];
        sleep (1);
        return 0;
    } else {
        /* Child */
        char *argv[100];
        int i, a;
        char *command = alloca(strlen(cmd)+2);
        strcpy (command, cmd);
        a = 0;
        cmd = command;
        argv[a++] = cmd;
    }
}

```

```

while (*cmd && *cmd != ' ') ++cmd;
if (*cmd)
    *cmd++ = 0;
while (*cmd) {
    while (*cmd == ' ') ++cmd;
    argv[a++] = cmd;
    while (*cmd && *cmd != ' ') ++cmd;
    if (*cmd)
        *cmd++ = 0;
}
argv[a] = NULL;
close (fd1[1]);
close (fd2[0]);
if (fd1[0] != STDIN_FILENO) {
    if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO) {
        fprintf (stderr, "dup2 error to stdin\n");
        close (fd1[0]);
    }
}
if (fd2[1] != STDOUT_FILENO) {
    if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO) {
        fprintf (stderr, "dup2 error to stdout\n");
        close (fd2[1]);
    }
}
if (execvp (command, argv) < 0) {
    fprintf (stderr, "execl error\n");
    return -1;
}
}
return 0;
}

```

Lighting Controls for Computer Cinematography

Ronen Barzel
Pixar Animation Studios

Reprinted from *journal of graphics tools*, v. 2 no. 1; 1997; A K Peters, Ltd.
<http://www.acm.org/jgt/papers/Barzel97>

Abstract. Lighting is an essential component of visually rich cinematographic images. However, the common computer graphics light source models, such as a cone-shaped spotlight, are not versatile enough for cinematographic-quality lighting. This paper describes the controls and features of a light source model for lighting computer graphics films. The model is based on generalized light cones, emphasizing independent control over the shape and texture of lights and shadows. While inspired by techniques of real-world cinematography, it is tailored to the needs and capabilities of computer graphics image generation. The model has been used successfully in production over the past few years to light many short works and the movie *Toy Story*.

Introduction

Photorealism is an important and much-studied goal in computer graphics imagery and illumination. However, photographers and cinematographers know that when it comes to lighting, realism is not the only goal.

The purposes of lighting for cinematography include contributing to the storytelling, the mood, and the image composition, and directing the viewer's eye. The "practical" light sources on a real-world movie set, such as desk or ceiling lamps, are rarely major contributors to the illumination. Instead, various types of lamps and spotlights are placed off camera, in order to create the desired illumination effect. A lighting designer will use whatever techniques, tricks, and cheats are necessary, such as: suspending a cloth in front of a light to soften shadows; positioning opaque cards or graded filters to shape a light; focusing a narrow "tickler" light to get an extra highlight; or hiding a light under a desk to fill in dark areas under a character's chin.

This paper presents a lighting model that has developed over several years in response to the needs of computer graphics film production, in particular for making *Toy Story*. The model gives the CG lighting designer control over the shape, placement, and texture of lights, so that the designer's real-world cinematographic talent can be applied to computer images.

The emphasis of the model is not on realism nor on physically simulating the tools of real-world cinematography. Rather, we take advantage of various sorts of unreality available to us in the CG world in order to get the desired cinematographic effects.¹ Thus, while real-world effects provide motivation, our lighting model is ultimately based on effects that are useful in computer graphics.

1. In fact, real-world cinematographers would doubtless use CG tricks if they could, e.g., have light emanate out of nowhere, cut off a light after a certain distance, or change the direction of shadows.

Note that this paper discusses the technology of cinematography, not the artistry; the art of cinematography is beyond the scope of this paper (and this author). We refer interested readers to cinematography texts such as [Lowell 92] or [Malkiewicz 86]. Of particular relevance, [Calahan 96] discusses the art of cinematography as applied to computer graphics.

Related Work

[Warn 83] developed the cone spotlight model that has since become standard in computer graphics (along with a simple barn door light). It was “based in part on observation...of the studio of a professional photographer,” in order to be able to create better-lit images than were available with then-standard tools. We continue in this vein, extending the model to meet increasingly ambitious cinematic lighting needs. Doubtless others have met similar needs, but we are not aware of other published work in this area.

End-user graphics systems typically have fairly powerful lighting tools. For example, [Alias 95] supports assorted light types, such as spot, point, and volume-restricted. Our model is more flexible in that any parameter or effect can be used with any shape, and we place greater emphasis on light textures and manipulation of shadows.

Our model does not include finite-area light sources (see, e.g., [Verbeck, Greenberg 84] or [Cook, Porter, Carpenter 84]), and it would likely benefit from them; however our light-shaping methods can to a large extent fake their soft lighting effects.

Global illumination methods can yield subtly and realistically lit images, but they typically support limited lighting controls and emphasize light from practical sources. However, the cinematographic emphasis on off-camera lighting and other trickery is not incompatible with global illumination methods. For example, [Dorsey, Sillion, Greenberg 91] models opera lighting using off-camera lights and projected images. [Gershbein, Schröder, Hanrahan 94], and [Arvo 95] also address textured light sources. Since our model is implemented via RenderMan shaders [Upstill 90], it can in principle work with a RenderMan-compliant global illumination renderer such as BMRT [Gritz, Hahn96], although we have not pursued this aspect. Pragmatically, however, global illumination is still too computationally expensive for regular use in computer graphics film production.

The task of a lighting designer can be aided by tools that speed up the process of choosing and varying lighting parameters. For example, [Bass 81] and [Dorsey, Arvo, Greenberg 95] describe techniques to quickly recompute and redisplay images as lighting parameters change; [Schoeneman et al. 93] and [Kawai, Painter, Cohen 93] determine lighting parameters given user-specified objectives. Tools such as these could be used in conjunction with our lighting model.

Overview

“The Lighting Model” describes the controls and features of the lighting model, and the section on “Implementation” sketches its implementation. “Results” presents and discusses several sample *Toy Story* images.

We will not discuss the user interface for interactively placing and manipulating the light sources; it is straightforward but beyond the scope of this paper.

The Lighting Model

The lighting model provides control over several aspects of each light source: *selection*, *shape*, *shadowing*, *texture*, *dropoff*, *direction*, and *properties*. We describe and illustrate these capabilities below.

For clarity, each feature is illustrated in isolation, although the model allows them to be used in any combination. Also, we illustrate with static images, but all parameters can of course be animated.²

Figure 1 shows a sample scene lit only by fill lights; Figure 2 shows the same scene with a simple conical key light. (We use the cinematography terms *key light* and *fill light* for major and minor sources of illumination, respectively.) Fog is introduced to illustrate the effect of the light throughout space.³

Selection

Computer graphics lights can be enabled or disabled on a per-object basis (Figure 3). The ability to selectively illuminate objects in a scene is a powerful feature, which has no analog in real-world lighting. In our experience, per-object selection is used frequently, in particular to adjust illumination separately for the characters and the set.

Shape

The basic task of lighting is the placement and shape of light in a scene. Real-world cinematography commonly uses spotlights and barn door (rectangular) lights to achieve desired shapes; our model provides a generalization of these light effects.

Generalized cone/pyramid. The light affects a region whose cross-section is a superellipse, continuously variable from purely round, through rounded-rectangle, to pure rectangle (Figs. 4–6). The slope of the pyramid can be varied until at the limit the sides are parallel. The pyramid may be truncated, as if it were originating from a flat lamp face, and may be sheared, for window and doorway effects (Figure 6).

Soft edges. To soften the edge of the light, we define a boundary zone in which the light intensity drops off gradually. The width and height of the boundary zone can be adjusted separately. Thus we have two nested pyramids: the light is at full intensity inside the inner pyramid, and has no effect—i.e., 0 intensity—outside the outer pyramid, with a smooth falloff between them (see Figs. 7, 8, and Appendix A on page 109).

Cuton and cutoff. The light shape can be modified further by specifying near and far truncation, again with adjustable-width smooth dropoff zones (Figs. 9 and 10). These have no real-world physical analog, but are very useful to soften a lighting setup, and to keep the light from spilling onto undesired parts of the scene.

Being able to adjust the shape and edge dropoff of lights easily allows for soft lighting of a scene, faking area-light penumbra effects.⁴ In our experience, the majority of lights are chosen to be mostly rectangular, with large edge zones to provide smooth gradation of lighting; conical lights are used mostly if the scene includes a practical source, such as a flashlight.

2. For your viewing pleasure, color versions of all the images in this paper can be found in the color plates, and in the *barzel* directory on the SIGGRAPH CD.

3. The fog is calculated by computing the incident light on hypothetical fog particles along each viewing ray, and accumulating the total light scattered back to the viewer. The fog has no effect on the light incident on the objects.

4. The lack of actual area lights in the model can manifest itself, however, in the shapes of shadows and surface highlights.

The light shape can of course be rigidly rotated, scaled, and placed anywhere in space. This is a strength of CG lighting over real-world lights: we are not encumbered by physical structures and mechanisms that must be hidden or placed off camera; CG lights can emanate spontaneously anywhere in space, giving the lighting designer great freedom.

Finally, another choice for the shape is to have no shape at all: the light affects all of space.

Shadowing

Shadows and shadow placement are an important part of cinematography. Computer graphics has great freedom to control shadows for artistic effect. In the following discussion, it is convenient to think of shadow projection as defining a “volume of darkness” inside of which illumination is inhibited; this volume can be manipulated as an independent entity.

Shadow selection. A light doesn’t necessarily have to cast shadows.⁵ In Figure 11, the key light casts shadows, but the background fill lights do not. Shadows may also be disabled on a per-object basis, as in Figure 12. Thus a difficult problem in real-world cinematography, suppressing unwanted shadows, is trivial in computer graphics.

Shadow direction. The direction that shadows are cast doesn’t necessarily have to follow the light—each “volume of darkness” can be aimed as needed. For example, the light in Figure 13 is the same as in Figure 11, but the shadows have been shifted so that the torus casts a shadow on the cylinder. It is perhaps surprising just how far shadow directions can be shifted from true without incurring visual dissonance. “Cheating” the shadow directions can be a powerful tool for controlling image composition; in our experience, background shadows are often “cheated.”

Shadow sharing. A seemingly bizarre capability is for a light to share its shadows with other lights (Figure 14). That is, a “volume of darkness” defined by a given light and object can inhibit illumination from other lights as well. This allows the lighting designer to strengthen shadows that might otherwise be washed out by nearby lights. In our experience shadows are often shared.

Fake shadows. It is often useful to create extra shadows, to imply nonexistent offscreen objects or simply to darken a scene where needed. In real-world lighting, opaque cards can be placed in front of a light. In our model, *blockers* can similarly be defined; each is specified by a 2D superellipse that can be placed anywhere in space (Figure 15). As with ordinary shadows, the direction that the blocker casts its shadows can be adjusted, and a blocker can be shared among several lights. In our experience, blockers are heavily used, sometimes several per light.

Shape trimming. A blocker can be made large and placed so as to trim the shape of a light, as in Figure 16. Animating a large blocker can be an easy way to fake a door-opening-offscreen effect.

Shadow softening. To keep shadows from being too harsh, any shadow or blocker can be made translucent, to only partially inhibit illumination. Shadow edges can be softened as well: for blockers, this is done via an edge-zone dropoff in the same manner as the light shape; for object shadows, the boundary of the “volume of darkness” is blurred. Finally, rather than going to black, a shadow can be assigned a color; subtle use of colored shadows can add richness to an image.

Texture

Just as images are used in computer graphics to create texture on surfaces, they can be used to create texture in lights via projection.

5. For most rendering algorithms it is of course cheaper and easier *not* to cast shadows.

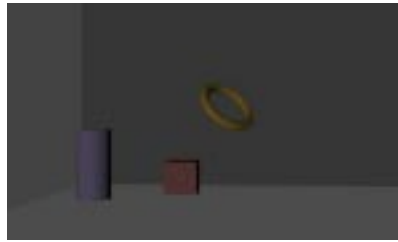


Figure 1. Cylinder and cube on the floor, torus in midair.

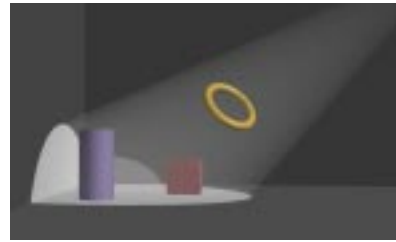


Figure 2. Same as Fig. 1, with a conical key light.

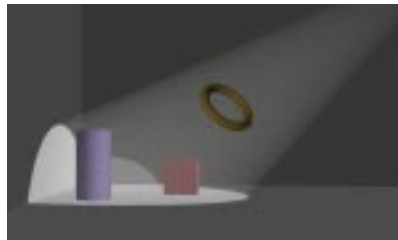


Figure 3. Selection. Same as Fig. 2, but the torus is unaffected by the key light.

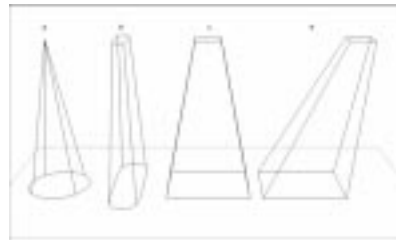


Figure 4. Shape. A superellipse profile is swept into a pyramid, which may be truncated or sheared.

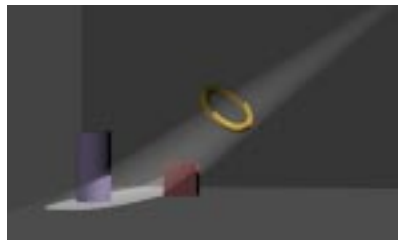


Figure 5. Rounded rectangle shape, as in Fig. 4b.



Figure 6. A sheared barn door light, as in Fig. 4d.

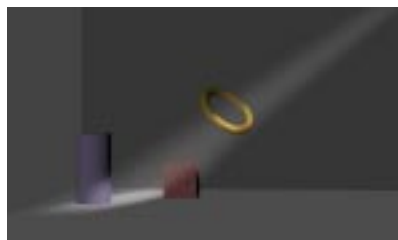


Figure 7. Same as Fig. 5, but with soft edges (Fig. 8).



Figure 8. Nested pyramids define the soft edges in Fig. 7.

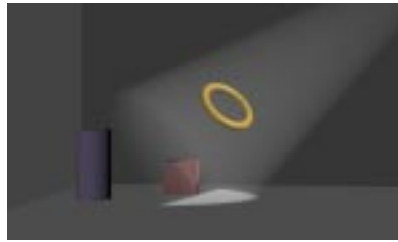


Figure 9. Same as Fig. 2, but with a sharp cutoff.

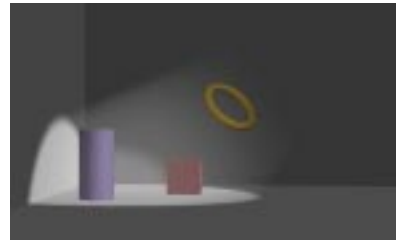


Figure 10. Same as Fig. 2, but with a gradual cutoff.



Figure 11. Basic shadows. Same as Fig. 2, but with shadows from the key light.

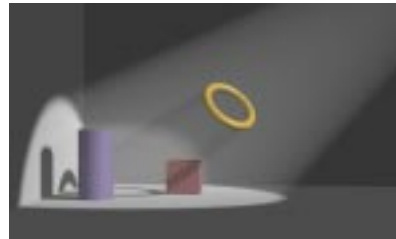


Figure 12. Shadow selection. Same as Fig. 11, but the cube casts no shadow.

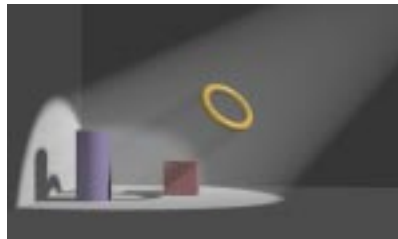


Figure 13. Cheated shadows. All light parameters are the same as in Fig. 11, but the shadow directions have been cheated so that the torus slightly shadows the cylinder.

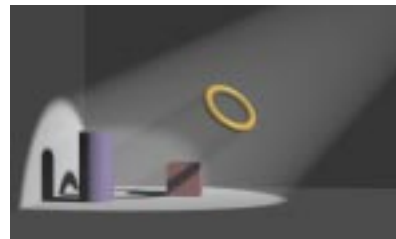


Figure 14. Shared shadows. Same as Fig. 11, but the key light shares its shadows with the fill lights.

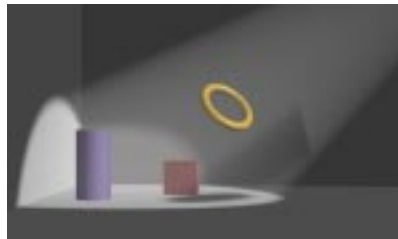


Figure 15. Faked shadow. Same as Fig. 2, but with a blocker that casts a shadow.

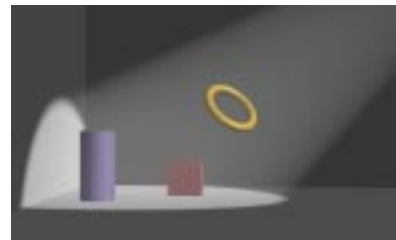


Figure 16. Shape trimming. Same as Fig. 2, but a large blocker has been placed just in front of the rear wall, to eliminate unwanted illumination of the wall.

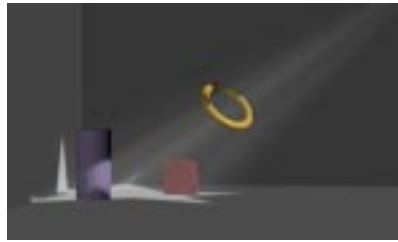


Figure 17. Projecting a matte image as a “cookie cutter” to get alternate light shapes (Fig. 19).

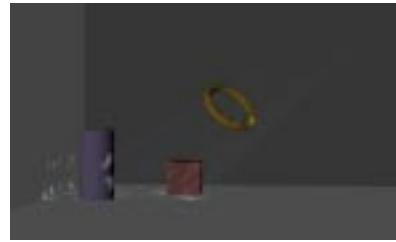


Figure 18. Projecting a matte image to get simulated shadows, here for a dappled-leaf effect (Fig. 19).



Figure 19. The matte images used in Fig. 17 and Fig. 18.

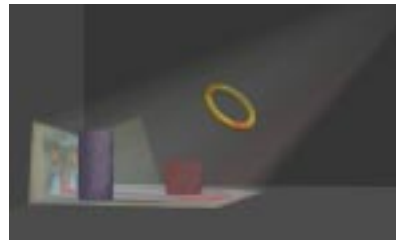


Figure 20. Projecting a color image (the well-known mandrill) to get a slide effect.



Figure 21. Intensity distribution across beam.



Figure 22. Intensity falloff with distance.

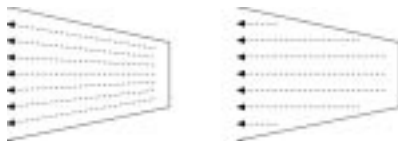


Figure 23. Cross section of light shape, with radial and parallel light rays.

Cookie. A single-channel matte image can be used as a “cookie cutter,”⁶ to get cross-sectional shapes other than the built-in superellipses (Figure 17), or, more subtly, to fake complex shadows from offscreen objects (Figure 18). Figure 19 shows the matte images used in Figures 17 and 18.

Slide. A full-color image yields a slide-projector effect (Figure 20). An unfocused projection (such as from a television set) can be achieved by applying a blur filter whose width increases as the projection distance increases.

Noise. In addition to stored image files, the light can be projected through a 2D noise function that modifies the intensity or color, yielding “dirty” lights.

As with shadows and blockers, it is possible to “cheat” the origin and direction of an image projection, to blur it, and to adjust its intensity.

Dropoff

The intensity of the light can vary, in the familiar manner of computer graphics lighting:

Beam distribution. Figure 21 illustrates dropoff of intensity across the beam using the usual exponentiated cosine function; however, the angle is normalized so that the intensity attenuates to 0 at the edge of the beam.

Distance falloff. Figure 22 illustrates dropoff of intensity with distance from the light source location, using the usual inverse-power attenuation; to keep the intensity well-defined at the light origin we provide a smooth clamp to a maximum intensity (see Appendix B).

In our experience, choosing exponential dropoff parameters is not visually intuitive; it is often easier to have no *a priori* dropoff, and to gradate lighting by using soft shape, cutoff, and blocker edges.

Direction

The light ray direction has two options common in computer graphics (Figure 23):

Radial. The rays emanate from a point at the apex of the pyramid.

Parallel. The rays are parallel to the centerline of the light pyramid, as per a very distant light.

Figure 23 illustrates parallel and radial rays in a light pyramid. The combination of parallel rays with widening shape yields a non-physical, but still useful effect: light rays are created along the edges of the shape. One can also imagine circumstances in which it would be useful to “cheat” the ray direction in other ways, e.g. to have rays parallel to a direction other than the pyramid centerline, or to define an arbitrary curvilinear vector field for the ray direction, but we have not needed such cheats in our standard model.

Note that by choosing a single well-defined light direction, we are implicitly assuming a point source or infinitely-distant source. To support finite-area lights, a mechanism such as ray distribution ([Cook, Porter, Carpenter, 84], [Veach, Guibas 95]) would be needed to be introduced to the model.

Properties

The previous sections have discussed where and in what direction the light reaches the surfaces in the scene. Finally, we have the properties of the “photons” that are incident on the surfaces:

6. “Cookie” is colloquial for “cucaloris,” the technical term for an opaque card with cut-outs, used to block a light.

Intensity. The nominal intensity of the light is specified at its origin or at a target point within the shape. This value is attenuated by the shape boundary, shadows, cookies, noise, and dropout.

Color. The color of the light is expressed as a standard 3-channel RGB value, which can be filtered by a slide, noise, or colored shadow.

Effect. The three standard CG illumination effects are available: ambient flat lighting, diffuse shape lighting, and specular highlighting. They may be used in combination, with a scale factor for each. (For ambient lighting, the ray direction is of course irrelevant.) In practice, diffuse-only lights are usually used for soft fills (Figure 1), while key lights use both diffuse and specular effects. A small ambient component to a light is useful to keep regions from going completely black.

Other information. Depending on what can be supported by the surface-shading model, additional information can be carried to surfaces. For example: a fourth channel of color can contain “ultraviolet” to which a “fluorescent” surface will react by self-illuminating; or we may have an identifier for each light source so that surfaces can react differently to specific lights.

Implementation

The lighting model is implemented as a RenderMan light source shader [Upstill 90]. The programmability of shaders has been invaluable in developing and extending the model.

Following the RenderMan paradigm, a light source is considered to be a functional unit: given a point on a surface anywhere in space, a light source computes the direction and color of the “photons” incident on that point due to that source. Figure 24 gives an overview of the computation. The actual RenderMan code is somewhat messier, however, in particular because RenderMan does not support arrays; there are a maximum number of shadowmaps, blockers, and so forth, and each **foreach** in Figure 24 is actually a series of tests against each parameter.

For efficiency, if any features aren’t used, we skip the corresponding steps of the computation. Also, if at any step the attenuation factor becomes 0, the remainder of the steps can be skipped. Finally, we use a mechanism that generates special-purpose shaders based on the chosen parameters, analogous to but simpler than the method of [Gunter, Knoblock, Ruf 95].

Figure 24 uses shadow maps for shadow generation ([Williams 78], [Reeves, Salesin, Cook 87]), as this is the mechanism supported by our renderer. With shadow maps, the shadow trickery of Section is straightforward: objects can be selectively included in shadow maps; shadow directions can be cheated by adjusting the shadow camera before computing the map; and shadows can be shared by referencing the same shadow map file in several lights. For other shadowing algorithms, these tricks may need to be coded into the renderer.

Results

Plates 1 through 6 on page 107 and 108 show several *Toy Story* frames. We discuss some illustrative features of the lighting in each.

Plate 1 is a simple scene that illustrates the significance of light shape and placement for image composition: the character is framed by a rectangular profile of a barn door light. The light nominally shines through a window offscreen to the right, but the placement was chosen for visual effect, not necessarily consistent with the relative geometries of the window, the sun, and the bed. Notice also the soft (partial-opacity) shadows of the bed on the wall.

GIVEN: POINT P ON SURFACE

COMPUTE: COLOR AND DIRECTION OF INCIDENT RAY

Each light is defined in local coordinates such that the pyramid is along the z axis with its apex at the origin, simplifying clipping and falloff calculations.

```

P1 = transform P to light's coords
atten = 1.0
// Clip to near/far planes
atten *= step(znear-nearedge, znear, zcomp(P1))
atten *= step(zfar, zfar+fareedge, zcomp(P1))
// Clip to shape boundary
atten *= clipSuperellipse(P1) // see Appendix A
// Apply blockers
foreach blocker
    Pb = project P into plane of blocker
    atten *= clipSuperellipse(Pb)
end
// Apply cookies
foreach cookie
    Pm = project P into plane of cookie
    atten *= texture(cookie.filename, Pm)
end
// Apply slide filters
foreach slide
    Ps = project P into plane of slide
    color *= texture(slide.filename, Pm)
end
// Apply noise
foreach noise texture
    Ps = project P into noise space
    atten or color *= noise(Ps)
end
// Apply shadows
foreach referenced shadowmap
    Ps = project P into plane of shadow
    atten or color *= test if in shadow
end
// Intensity dropoff
atten *= Falloff(zcomp(P1))
atten *= BeamDistribution(P1)
// Final output
output(color) = intensity*color*atten
output(rayDirection) = P (radial) or z (parallel)

```

Figure 24. Overview of light computation.

Plate 2 has stripes of light on the characters, that are not shadows of the venetian blind slats, but are generated using a cookie that is adjusted for dramatic effect; The slats are lit from below by a separate light, in order to obtain the desired highlighting.

Plate 3 also uses a cookie (the same one used in Figs. 18 and 19) to generate a dappled leaf effect. A separate cone-shaped light, with its own cookie, spotlights the area around the soldier crouching in the rear. The characters in the foreground are lit with an additional, blue light acting only on them for the blue highlights. This scene also includes a practical light, the red LED, which is used for effect but not as a key source of illumination.

Plate 4 features a practical light source, the flashlight, also using a cookie for the lens ring effect. The flashlight actually includes two light sources, one that illuminates the character and one used only for a fog calculation; the latter has a cutoff so that the fog doesn't obscure the character. This scene also features an "ultraviolet" light (as described in Section), responsible for the blue glow of the white shirt.

Plate 5 illustrates the importance of shadow placement—the shadows of the milk crate were carefully adjusted so that an "X" would fall on the character's face without obscuring the eyes or mouth, since the character is speaking. Blockers were also used to darken the background, both to darken the mood and to accentuate the character's face in the foreground.

Plate 6 contains a variety of techniques: the key light has soft edges; the character's shadow direction is cheated for compositional effect; a light at a grazing angle from the left illuminates only the desk-top, to accentuate its texture; separate lights illuminate only the character to provide extra highlighting; and a cookie is animated from one frame to the next, to provide a falling-rain effect.

Conclusion

The lighting model we describe provides a convenient and powerful encapsulation of control for cinematography. It has been developed in response to needs and requests of lighting designers doing computer graphics film production: the features that we have described are those that have proven useful in practice, and almost all lighting effects used in *Toy Story* were achieved with the available features.

Our lighting model is a straightforward collection and extension of common computer graphics methods. It (or a variation) is perhaps a candidate for inclusion in standard graphics libraries. We would be particularly interested in seeing support for controllable light shape and texture in rendering hardware, in order to be able to interactively create cinematographically lit images.

An intriguing question is to what extent lighting models such as ours—practical but non-physical—can be incorporated into physically-based global-illumination rendering schemes. The combination of controlled lighting and sophisticated rendering has potential for creating images of a quality as yet unseen.⁷

Acknowledgements

This lighting model is an extension of earlier work by Yael Milo and others. The blockers are based on work of Larry Aupperle and Oren Jacob. The fog effect is based on work of Mitch Prater. This model would not have been developed without feedback and use by the *Toy Story* lighting team. Kurt Fleischer, Sharon Calahan, and Uriel Barzel provided valuable suggestions for this paper.

7. Please refer to the Appendix for some example shader code.



Plate 1. (Image © 1995 Walt Disney Corporation)



Plate 2. (Image © 1995 Walt Disney Corporation)

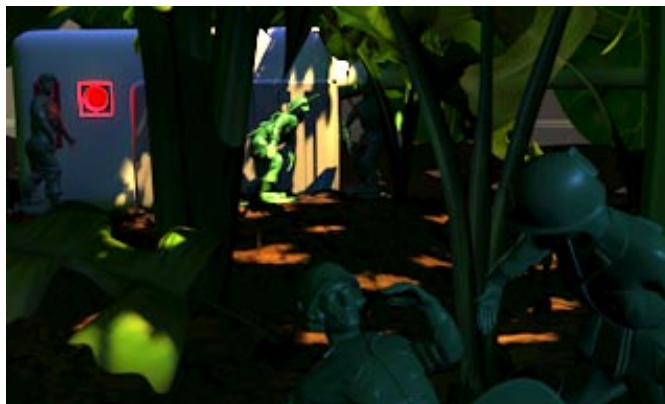


Plate 3. (Image © 1995 Walt Disney Corporation)



Plate 4. (Image © 1995 Walt Disney Corporation)



Plate 5. (Image © 1995 Walt Disney Corporation)



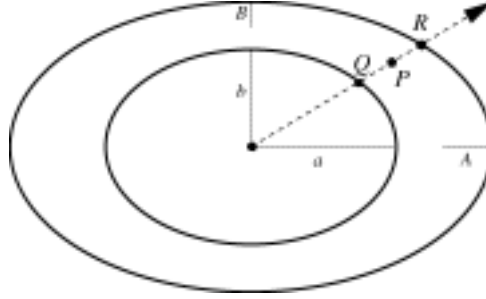
Plate 6. (Image © 1995 Walt Disney Corporation)

Appendix A: Superellipses

A superellipse is a figure that varies between an ellipse and (in the limit) a rectangle, given by:

$$\left(\frac{x}{a}\right)^{\frac{2}{d}} + \left(\frac{y}{b}\right)^{\frac{2}{d}} = 1$$

where a and b are the x and y radii, and d is a “roundness” parameter varying the shape from pure ellipse when $d = 1$ to a pure rectangle as $d \rightarrow 0$ (Figure 4).



We want to soft-clip a point P to a shape specified by two nested superellipses, having radii a, b and A, B . That is, given P , compute a clip factor of 1 if it is within the inner superellipse and 0 if it is outside the outer superellipse, varying smoothly value in between. We assume that the 3-space point has been projected into the first quadrant of the canonical plane of the ellipse.

We express the ray through P as $P(s) = sP$, and intersect it with the inner superellipse at Q , and with the outer at R . To find the points P, Q , and R , we express them as:

$$P = P(p), \quad Q = P(q), \quad R = P(r).$$

Trivially, $p = 1$. To compute q , we derive:

$$\begin{aligned} \left(\frac{qP_x}{a}\right)^{\frac{2}{d}} + \left(\frac{qP_y}{b}\right)^{\frac{2}{d}} &= 1 \\ q^{\frac{2}{d}} \left(\left(\frac{P_x}{a}\right)^{\frac{2}{d}} + \left(\frac{P_y}{b}\right)^{\frac{2}{d}} \right) &= 1 \\ q^{\frac{2}{d}} &= \left(\left(\frac{P_x}{a}\right)^{\frac{2}{d}} + \left(\frac{P_y}{b}\right)^{\frac{2}{d}} \right)^{-1} \\ q &= \left(\left(\frac{P_x}{a}\right)^{\frac{2}{d}} + \left(\frac{P_y}{b}\right)^{\frac{2}{d}} \right)^{-\frac{d}{2}} \\ &= ab \left((bP_x)^{\frac{2}{d}} + (aP_y)^{\frac{2}{d}} \right)^{-\frac{d}{2}} \end{aligned}$$

and similarly $r = AB \left((BP_x)^{\frac{2}{d}} + (AP_y)^{\frac{2}{d}} \right)^{-\frac{d}{2}}$. The final clip factor is given by $1 - \text{smoothstep}(q, r, p)$, where RenderMan's **smoothstep** [Upstill 90] computes:

$$\text{smoothstep}(q, r, p) = \begin{cases} 0, & p < q \\ \text{Hermite interpolation} & q \leq p \leq r \\ 1, & p > r \end{cases}$$

For a pure rectangle, $d = 0$, we simply compose x and y clipping to compute a clip factor:

$$(1 - \text{smoothstep}(a, A, P_x)) \times (1 - \text{smoothstep}(b, B, P_y))$$

which gives a different falloff at the corners than the limit of the round calculation, but suits our purposes.

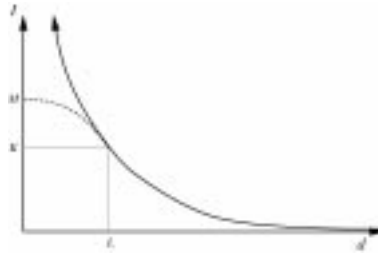
Falloff functions other than the Hermitian **smoothstep** could be useful in some circumstances, but we have not experimented with any. It could also be useful to support asymmetric edge widths; but we have not as yet done so.

Appendix B: Intensity falloff curve

The common inverse power formula for light intensity can be expressed as

$$I(d) = K \left(\frac{L}{d} \right)^\alpha$$

where d is distance from the light source, α is attenuation exponent, and K is the desired intensity at a canonical distance L . This expression grows without bound as d decreases to 0 (solid line):



A common solution is to clamp the intensity to a maximum value; however this yields a curve with discontinuous derivative, potentially causing Mach banding. Instead, we use a Gaussian-like curve (dashed line) when inside the canonical distance:

$$I(d) = \begin{cases} Me^{\left(\frac{d}{L}\right)^\beta}, & d < L \\ K \left(\frac{L}{d} \right)^\alpha, & d > L \end{cases}$$

where $s = \ln\left(\frac{K}{M}\right)$ and $\beta = -\frac{\alpha}{L}$ are chosen so that the two curves have matching value $I(d)=K$ and slope $I'(d) = -\frac{K\alpha}{L}$ at distance $d=L$.

References

[Alias 95] Alias | Wavefront, a division of Silicon Graphics Canada Limited, Toronto, 1995.

- [Arvo 95] James Arvo. "Applications of Irradiance Tensors to the Simulation of Non-Lambertian Phenomena." In *Computer Graphics* proceedings, Annual Conference Series, ACM SIGGRAPH, 1995, pp. 335–342.
- [Bass 81] Daniel H. Bass. "Using the Video Lookup Table for Reflectivity Calculations: Specific Techniques and Graphics Results." *Computer Graphics and Image Processing*, 17:249–261(1981).
- [Calahan 96] Sharon Calahan. "Storytelling Through Lighting: A Computer Graphics Perspective." *SIGGRAPH 96 course notes*. New York: ACM, 1996.
- [Cook, Porter, Carpenter 84] Robert L. Cook, Thomas Porter, and Loren Carpenter. "Distributed Ray Tracing." *Computer Graphics, (Proc. SIGGRAPH 84)*, 18(3):137–145(July 1984).
- [Dorsey, Arvo, Greenberg 95] Julie Dorsey, James Arvo, and Donald Greenberg. "Interactive Design of Complex Time-Dependent Images." *IEEE Computer Graphics and Applications*, 15(2):26–36(March 1995).
- [Dorsey, Sillion, Greenberg 91] Julie O'B. Dorsey, François X. Sillion, and Donald P. Greenberg. "Design and Simulation of Opera Lighting and Projection Effects." *Computer Graphics (Proc. SIGGRAPH 91)*, 25(4):41–50(July 1991).
- [Gershbein, Schröder, Hanrahan 94] Reid Gershbein, Peter Schröder, and Pat Hanrahan. "Textures and Radiosity: Controlling Emission and Reflection with Texture Maps." In *Computer Graphics* Proceedings, Annual Conference Series, ACM SIGGRAPH, 1994, pp. 51–58.
- [Gritz, Hahn 96] Larry Gritz and James K. Hahn. "A Global Illumination Implementation of the RenderMan Standard." *journal of graphics tools*, 1(3):29–48(1996).
- [Guenter, Knoblock, Ruf 95] Brian Guenter, Todd B. Knoblock, and Erik Ruf. "Specializing Shaders." In *Computer Graphics* Proceedings, Annual Conference Series, ACM SIGGRAPH, 1995, pp. 343–350.
- [Kawai, Painter, Cohen 93] John K. Kawai, James S. Painter, and Michael F. Cohen. "Radioptimization—Goal Based Rendering." In *Computer Graphics* Proceedings, Annual Conference Series, ACM SIGGRAPH, 1993, pp. 147–154.
- [Lowell 92] Ross Lowell. *Matters of Light & Depth*. Philadelphia: Broad Street Books, 1992.
- [Malkiewicz 86] Kris Malkiewicz. *Film Lighting*. New York: Prentice Hall Press, 1992.
- [Reeves, Salesin, Cook 87] William T. Reeves, David H. Salesin, and Robert L. Cook. "Rendering Antialiased Shadows with Depth Maps." *Computer Graphics (Proc. SIGGRAPH 87)*, 21(4):283–291(July 1987).
- [Schoeneman et al. 93] Chris Schoeneman, Julie Dorsey, Brian Smits, James Arvo, and Donald Greenberg. "Painting with Light." In *Computer Graphics* Proceedings, Annual Conference Series, ACM SIGGRAPH, 1993, pp. 143–146.
- [Upstill 90] Steve Upstill. *The RenderMan Companion*. Reading, MA: Addison-Wesley, 1990.
- [Veach, Guibas 95] Eric Veach and Leonidas J. Guibas. "Optimally Combining Sampling Techniques for Monte Carlo Rendering." In *Computer Graphics* Proceedings, Annual Conference Series, ACM SIGGRAPH, 1995, pp. 419–428.

- [Verbeck, Greenberg 84] Channing P. Verbeck and Donald P. Greenberg. “A Comprehensive Light-source Description for Computer Graphics.” *IEEE Computer Graphics and Applications* 4(7):66–75(July 1984).
- [Warn 83] David R. Warn. “Lighting Controls for Synthetic Images.” *Computer Graphics (Proc. SIGGRAPH 83)*, 17(3):13–21(July 1983).
- [Williams 78] Lance Williams. “Casting Curved Shadows on Curved Surfaces.” *Computer Graphics (Proc. SIGGRAPH 78)*, 12(3):270–274(August 1978).

Mock Media

Scott F. Johnston
Fleeting Image Animation, Inc.

Introduction

Before photorealism there was non-photorealistic imagery. For thousands of years before the invention of the camera, artists drew, painted, carved, and printed their impressions of the world. Most artists are not compelled to pursue realism and quite often purposely avoid it.

In addition to synthesizing the photographic process, there is a growing trend among digital artists to imitate other media realistically. Work by Meier on painterly rendering, Winkenbach et al. on pen and ink illustration, and Curtis on watercolor show an interest not for “non-photorealism,” but rather for reproducing other media. During development of “The Lion King”, the goal for the wildebeest stampede was described as *cartoon realism*. The intent was to integrate CG with the rest of the film and “really look like a cartoon.”

Tools designed for producing photorealistic effects can be successfully tailored to create a variety of other looks. In addition, information encoded by these tools can provide temporal coherence, allowing animation to be created in techniques which would be too difficult to achieve traditionally.

Most research in the past ten years has been focused on photo-realism. The implicit belief has been that the process of achieving this goal will deliver the tools required to recreate alternate techniques. Interestingly, before hardware was capable of producing photorealistic images, CG pioneers relied on alternatives, like the pen-plotting methods which produced Peter Foldes film “Hunger” in 1973.

The pursuit of other techniques never went away, but has been overshadowed by substantial progress in photorealism. Relatively recently, technology has begun to allow artists to succeed in the quest for realism. That people are turning to alternatives is further proof of this achievement.

Rendering Mock Media

Flat-media

Most existing research has been on “flat media.” These are canvas or paper based techniques, like drawing, painting, pen and ink illustration, and wood block printing. A critical property of flat media pointed out by Meier [3] is that the materials are or appear to be applied on the image plane. Exact perspective and texture mapping can often reveal the photorealistic underpinnings of a renderer.

Space filling curves: Wood block printing & Pen and Ink Illustration

In both wood block and pen and ink illustration, line quality is kept fairly consistent. As an object's apparent size changes, by advancing, receding, or scaling, the line density should be adjusted-- a far or small object should be drawn with fewer lines than a near or large object. Similarly, a surface that pinches together should have fewer lines in the pinched area and more lines in the stretched area.

To work in animation, these changes should evolve over time in a coherent manner-- having a new solution on each frame can cause undesirable strobing.

In both techniques, the image plane is filled by lines that define the surfaces. Lighting is accomplished by one of two methods: For pen and ink illustration, the density of lines varies as a function of image intensity, but the individual line widths remain constant. This can be accomplished by layering lines over one another, often at opposing angles creating cross-hatching. For wood block printing, the line density remains uniform, but the width of the line varies based on the image intensity.

Example: Wood block Printing¹

Wood block and linocut printing involve carving grooves into a block of wood or linoleum, inking the surface and pressing it onto paper to print a mirror image negative of the image carved onto paper. Commonly, more than one block is used for a single output image, with different aspects of the image on each block. These layers can be printed with different colored inks to create multi-tonal imagery. Alternatively, a single black ink can be used and the resulting image can be hand-tinted.

The contours of the grooves in a woodcut are often designed to accentuate features of the surface being described. The width and spacing of the grooves can be used to control tonality, or perceived value. (The width is usually controlled by the depth of the chisel marks.)

Following contours makes synthesis fairly natural, because the iso-parametric curves of surface patches often follow these features. (Or at least a simple transform of the parametric coordinate system.)

The shader solution involves developing a uniform linear space-filling parametric surface texture.

Simple Stripes

A simple, naive method for striping an object is to use *mod()*. For example, stripes running along the *v* direction of a patch can be shaded by:

```
sawtooth = mod(u * frequency, 1);
triangle = abs(2.0 * sawtooth - 1.0);
square = filterstep(duty, triangle);
```

This first defines a sawtooth wave of a specified frequency along the *u* direction. The sawtooth is converted to a triangle wave, and then to a square wave with a given duty-cycle. As the triangle wave's value raises above the duty-cycle value, the step function will trigger.²



This function provides uniform striping when applied to a simple patch.

-
1. Please see the Appendix at the end of the course notes for shader code implementing the ideas in the woodblock discussion.
 2. For your viewing pleasure, bigger and better versions of all the images in this paper can be found in the *johnston* directory on the SIGGRAPH CD.



When applied to an object with non-uniform geometry, the stripes become non-uniform as well. Where the object pinches together, like the top of the sphere, the stripes become narrower, and where the object is wider, around its equator, the stripes are wider.

As an object changes size, so do the stripe details. The smaller sphere has the same features, just smaller.

Uniform line density

To achieve consistent line spacing and width, the stripe frequency is modified based on the size of the object-- increased where the surface is stretched, for more stripes, and decreased where it is pinched, for fewer.



RenderMan provides derivatives across surfaces, and *length(dPdu)* is a good measure of how stretched or pinched a surface is in the direction normal to the stripes. The value in this image is proportional to the length of the derivative in the *u* direction.

Because the length variation is continuous across *v*, directly modulating the frequency of the stripes by this value produces a continuously changing frequency, which is difficult to control, and therefore undesirable.

To compensate for this, rather than modulate the frequency continuously, adjust the frequency in discrete steps following a simple rule: Every time the *length(dPdu)* doubles, use twice as many stripes.

```
logdp = log(length(dp))/log(2); /* Log base 2 of length(dp) */
ilogdp = floor(logdp);          /* Floor to discretize */
stripes = pow(2,ilogdp);        /* Restore multiplier */
sawtooth = mod(u * stripes * frequency, 1);
```

Normalize line width

The discrete stepping introduces some undesirable artifacts (jumps) in the width of the lines. This can be corrected by tapering the duty-cycle across the transition from one frequency to the next.

```
transition = logdp - ilogdp;
```

The transition variable will vary from 0.0 to 1.0 in the interval between discrete frequency jumps. Since the frequency doubles at each transition, this can be used to halve the duty-cycle across the region.

```
square = filterstep(duty*(1 - transition/2), triangle);
```



Tapering by changing duty-cycle

As with the pinching and stretching, when the object changes size, the length of the derivative will scale, and thus the amount of striping will change. Because of the discrete steps, the frequency changes will be coherent over time-- new stripes appear to “draw on” or “erase off” in between other stripes as they spread and contract. The overall density remains uniform.



Detail reduces as size reduces

Coherent frequency adjustment

If the frequency is changed over time, the stripes slide across the surface:



Frequency changes cause stripe sliding on surface

By moving the frequency term inside the log function, frequency adjustments react the same as scale adjustments-- as existing lines thin and separate, new lines are added in-between.



Frequency inside log function

Screen Space attenuation

This procedural surface texture compensates for stretching and pinching of the object and the overall object size, but is still being computed on a 3D object in world space. The detail would still be maintained as an object recedes in space. Also, towards the edge of the sphere, where the surface bends away from us, the apparent frequency increases.

To mimic a natural technique, the shader needs to be modified to work relative to the canvas: screen space.

An approximate solution is to transform the derivative into screen space and clamp it in z .

```
dp = tmp * (transform("screen", P+dPdu/tmp) - transform("screen", P));
setzcomp(dp, 0.0); /* Normal dropoff compensation */
```



In this image, where the value indicates the length of dp , the edges become darker as the normal falls off.

Note, that for large $dPdu$, the transform can create problems across the perspective divide depending on the scene and RenderMan implementation. The “tmp” term indicated is a compensating hack. For these examples, a “tmp” of 100.0 was used.

Now as an object advances or recedes, or as a surface drops away from us, the screen-space derivative will scale and shrink and thus the frequency will adjust to attempt to maintain uniform line widths.



Line width compensation as normal drops off

Tapered tips

The frequency transition of lines is still abrupt. While this may be appropriate for thin-line pen and ink illustration, for woodcut printing, an adaptation of how the “transition” variable is used gives a better appearance.

If the triangle wave is representative of grooves cutting into wood, adding a new line between two others should be accomplished by slowly chiseling to a desired depth, not by abruptly starting a new groove. The following code increases the frequency across transitions by continuously narrowing the peaks and raising the valleys into a new set of peaks.

```
transtriangle = abs((1 + transition) * triangle - transition);
square = filterstep(duty, transtriangle);
```

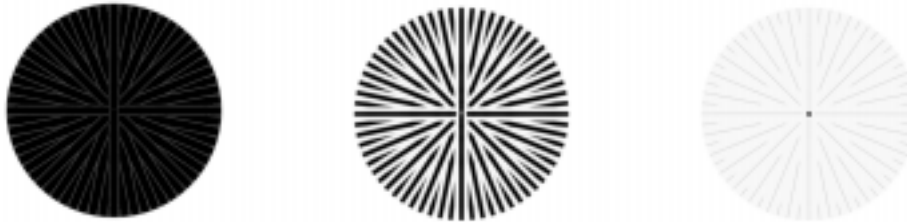
The grooves now have shallower entry and exit points.



Line width compensation with tapered tips

Space filling algorithm

Looking down at the sphere and modifying the duty-cycle shows the space-filling nature of the resulting procedural texture.



Duty cycle adjustment showing space fill

Angle

Our development has been along isoparametric lines and works both in u and v . The parametric coordinate system can also be rotated, as long as the derivatives are also rotated.



Angle adjustment

Lighting

Thus far, we've been keeping the frequency and duty-cycle consistent. Lighting computations can be used to modulate these to create a variety of effects. For pen-and-ink, the frequency would also be modulated by lighting intensity; duty-cycle tapering can be used to maintain constant width. For wood block printing, the duty cycle is the measure of how wide or narrow the lines appear, and can be made directly proportional to desired intensity.

A simple lighting equation is sufficient.

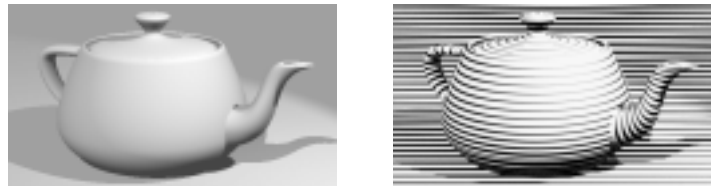
```
diffuzeC = Ka * ambient() + Kd * diffuse(faceforward(normalize(N),I));
diffuze = (comp(diffuzeC,0)+comp(diffuzeC,1)+comp(diffuzeC,2))/3.0;
/* Straight average here. Luminance can also be used. */
```

While the lighting can be adjusted for different layers, it is useful to have a separate linear scaling term so that different features can be pulled from the same lighting condition.

```
diffuze = (diffuze - lo)/(hi - lo);
```

This replaces, or modulates the duty cycle.

```
square = filterstep(duty * diffuze, triangle); /* usually want duty = 1.0
now. */
```



Intensity as duty cycle step threshold

Layers

Post-process compositing can be used to “print” a set of render layers into a final image.



Compositing layers

Some layers, particularly light area cross-hatches, can be used for “anti-lighting,” erasing from the underlying layers rather than being composed with them.

When rendering multiple passes for multiple layers, complex lighting can be computed in a separate pass and stored as a texture map. Subsequent levels will not have to compute the lighting and in fact can have all lights turned off.

To use a pre-existing render, the “diffuze” term is evaluated via a texture look-up in screen space. Note that you may need to pass your shader the aspect ratio of the image to properly scale the indices.

```

if (mapname != "") {
    Pscreen = transform("screen", P);
    screenx = 0.5 * (xcomp(Pscreen) / aspect + 1.0);
    screeny = 1.0 - 0.5 * (ycomp(Pscreen) + 1.0);
    diffuze = float texture(mapname[0], screenx, screeny);
}

```

An added benefit to rendering intensity in a separate pass is that the image resolution of the lighting pass can be significantly smaller than the striping passes. When computed at the same resolution, high frequency lighting information, such as a hard shadow edge, can cause abrupt changes in line width. In wood block printing, we want to soften these transitions while maintaining high-contrast lines. A smaller intensity map with texture filtering accomplishes this.

Stochastic Adjustments

Noise can be used to make the effect more natural.



The line width can be adjusted and lines made to stipple slightly by adding noise to the “diffuze” term as a function of the parameter normal to the striping direction.

To be accurate, the noise frequency should be adjusted so the frequency of the stipple transitions between octaves of noise as the apparent frequency in this direction changes. In practice, this isn't always necessary.



The peaks of our triangle waves can also be adjusted within a stripe to jiggle the stripes across the surface. In this case, their width is maintained, but the placement of the visible portion of the stripe is moved in the direction normal to the stripe.

Animation

The noise functions can be kept fixed, for locked frame-to-frame coherence, but in animation it can be more natural to allow them to vary over time. Offsets to the noise functions can either be random, causing new stippling and jiggling for each frame, or can be evolved over time, producing a smoother, continuous variation.



Combining both techniques and raising the frequency

Recommended Reading

- Cassidy J. Curtis, Sean E. Anderson, Joshua E. Seims, Kurt W. Fleischer, and David H. Salesin. "Computer-Generated Watercolor." *Computer Graphics (Proc. SIGGRAPH 97)*, pages 421-430. 1997.
- Paul Haeberli. "Paint By Numbers: Abstract Image Representations." *Computer Graphics (Proc. SIGGRAPH 90)*, pages 207-214. 1990.
- Barbara J. Meier. Painterly "Rendering for Animation." *Computer Graphics (Proc. SIGGRAPH 96)*, pages 477-484. 1996.
- Victor Ostromoukhov and Roger D. Hersch. "Artistic Screening." *Computer Graphics (Proc. SIGGRAPH 95)*, pages 219-228. 1995.
- Takafumi Saito and Tokiichiro Takahashi. "Comprehensible Rendering of 3-D Shapes." *Computer Graphics (Proc. SIGGRAPH 90)*, pages 197-206. 1990.
- Steve Upstill. *The RenderMan Companion*. Reading, MA: Addison-Wesley Publishing Company, 1990.
- Georges Winkenbach and David H. Salesin. "Computer-Generated Pen-and-Ink Illustration." *Computer Graphics (Proc. SIGGRAPH 94)*, pages 91-100. 1994.
- Georges Winkenbach and David H. Salesin. "Rendering Parametric Surfaces in Pen and Ink." *Computer Graphics (Proc. SIGGRAPH 96)*, pages 469-476. 1996.