

Using RenderMan® in Animation Production



Siggraph 1995 Course 4

Monday, August 7, 1995

Tony Apodaca

Course Chair

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. This course will teach people how to use RenderMan. We will discuss generating data to pump into RenderMan renderers, programming the RenderMan Shading Language to generate special effects, and accessing the special features of the RenderMan-compatible renderers that are available today. We will examine in-depth the production of several 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 © 1992, 1995 Pixar. All rights reserved.

RenderMan® is a registered trademark of Pixar.

Course Objectives

By the end of this course, the student should understand why using RenderMan is both more difficult and more powerful than using other renderers available to him. He will know the basic techniques for taking advantage of that power to make great animations, see that it is not so difficult after all, and will be enthusiastic in his desire to get out there and do it.

Level/Desired Background

This course is rightfully an “Intermediate” level course. The course does require a basic background in both 3-D computer graphics and programming, but you don’t have to be a rocket scientist. This course will not be nearly as technical as the 1992 course “Writing RenderMan Shaders”. However, some familiarity with the RenderMan and its Shading Language is valuable (e.g., have read the “RenderMan Companion”, or have written at least one shader).

What’s most important is a fascination with and love of computer graphics animation production. We encourage graphics programmers, advanced graphics users, CGI production personnel, or anyone who wants to see how other people cajole RenderMan into doing all that neat stuff to attend.

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 is a must read. Answers to all of the typical, and many of the extremely advanced, questions about RenderMan are found within its pages. Its only failing is that it does not cover RIB, only the C interface.

“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. Unfortunately, it is not as approachable as the Companion.

“The RenderMan Interface and Shading Language,” Siggraph 1991 Course #21 Course Notes (or alternatively, Siggraph 1990 Course #18 Course Notes)

These are the notes to the first Siggraph course on RenderMan. This course follows the presentation of the Companion. If you can find one of these, you may find them very useful.

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

These are the notes to the 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

Tony Apodaca is the Director of RenderMan R&D at Pixar. Tony is co-developer and reigning Chief Architect of the RenderMan Interface Specification. He is also one of the Unknown Implementers of all of Pixar's image synthesis products, both famous and infamous. Tony received his Master's degree in Computer and Systems Engineering at Rensselaer Polytechnic Institute in 1986. His screen credits include *Tin Toy* and *knickknack*, but unfortunately not *Jurassic Park*.

Larry Gritz

Larry Gritz is a Graphics Software Engineer at Pixar. Prior to joining Pixar he developed the Blue Moon Rendering Tools (BMRT), a shareware, RenderMan-compliant renderer which supports ray tracing and radiosity. He expects to complete his PhD at the George Washington University later this year.

Tom Porter

Tom Porter is Director of Effects Animation at Pixar. Tom has been working in 3-D graphics since 1976. He wrote the software for the first commercial paint program (the Ampex AVA), developed fundamental algorithms for digital compositing (which led to the Pixar Image Computer) and for motion-blurred rendering. Tom has taken time out over the years to create still frame images (*1984*, *Textbook Strike*) to advance the state of the art in computer generated photorealism, his main research interest.

Oren Jacob

Oren Jacob is a Senior Technical Director at Pixar. He has been with the company for five years and has worked on numerous award winning commercials like *Listerine Arrows*, *Mission* and *Jungle*, *LifeSavers Conga*, *TetraPak Island*, and short films for Sesame Street like *Luxo Jr. in In and Out*. He has also worked on several commercials that did not win any awards, but his mother still loves him. Oren holds a Master's Degree from the University of California, Berkeley in Mechanical Engineering yet still claims to understand computer graphics.

MJ Turner

M.J. Turner is the Lead Technical Director for the CGI department at Walt Disney Feature Animation. She joined Disney eight years ago and has worked on films such as *Oliver & Company*, *Oilspot and Lipstick*, *The Little Mermaid*, *Rescuers Down Under*, *Prince & the Pauper*, *Beauty & the Beast*, *Aladdin*, *The Lion King* and *Pocahontas*. She is involved with the design and development of new ideas and techniques to be used for upcoming releases. M.J. graduated from the University of Southern California with a B.S. and M.S. in Computer Science.

Joe Letteri

Joe Letteri is a Computer Graphics Supervisor in the Computer Graphics Department at In-

dustrial Light & Magic where he specializes in lighting design and rendering of computer graphic images and special effects for integration into feature films. The award-winning work he did for the photo realistic look of the dinosaurs in *Jurassic Park* has made him one of the pioneers in computer graphics. His screen credits include *Casper*, *Flintstones*, *Jurassic Park* and *The Abyss*. Prior to ILM, Joe worked at MetroLight Studios. He is a member of ACM Siggraph.

Ellen Poon

Ellen Poon is a Computer Graphics Supervisor at ILM. In addition to overseeing a computer graphics crew, she also creates character animations and performs lighting tasks for feature films as well as commercials. Ellen received her B.Sc. in Computer Science from the University of Essex in England. She continued her Ph.D. studies at the University of London where she carried out Computer Graphics research work. She then joined the Moving Picture Company in 1986 and subsequently Rushes in London to produce computer animation pieces commercially. Her screen credits include Perrier *Toy Soldiers*, *Hook*, *Jurassic Park*, *The Mask* and *Disclosure*. Ellen's continuing interests are to extend the realm of creative vocabularies and to explore alternative art form are well beneath underground.

Habib Zargarpour

Habib Zargarpour is a Senior Technical Director at ILM. He has been an illustrator since 1981. He received his B.A.S.C in Mechanical Engineering from the University of British Columbia in VancouverCanada and went on to graduate with distinction in Industrial Design from Art Center College of Design in Pasadena, CA. He stumbled into 3-D graphics in film in 1990 while designing for a film. His recent projects include *The Mask* and *Star Trek: Generations*. He considers computer graphics to be the perfect mix of the technical and artistic worlds. He still likes to paint with real brushes once in a while to maintain his sanity.

Schedule

Welcome and Introduction Tony Apodaca Page 1	8:45 AM
RenderMan: Using the Standard and its Implementations Larry Gritz Page 3	9:00
<i>Break</i>	10:00
Programming RenderMan Shaders Tony Apodaca Page 45	10:15
Writing Surface Shaders Tom Porter Page 98	11:15
<i>Lunch</i>	12:00 PM
Television Commercial Production at Pixar Oren Jacob Page 119	1:30
Computer Graphics at Walt Disney Animation MJ Turner Page 125	2:15
<i>Break</i>	3:00
CG Special Effects Production at ILM Joe Letteri, Ellen Poon, Habib Zargarpour Page 130	3:15
Q & A and Wrap-Up	4:30

Course Notes Table of Contents

Introduction.....	Page 1
RenderMan: Using the Standard and its Implementations	Page 3
Programming RenderMan Shaders	Page 45
Writing Surface Shaders	Page 98
Television Commercial Production at Pixar	Page 119
Computer Graphics at Walt Disney Animation	Page 125
CG Lighting Design for Feature Films	Page 130
Creating Computer Images for Film Using RenderMan.....	Page 134
Energy Ribbon Effect for <i>Star Trek: Generations</i>	Page 137
The RenderMan Interviews.....	Page 145

Using RenderMan in Animation Production

Tony Apodaca
Pixar

Welcome to *Using RenderMan in Animation Production*. In this course, we will explore how RenderMan has been and can be used in the production of high-quality computer animation. The course is significantly less technical than the previous two courses devoted to RenderMan, and is more about the trials and tribulations of high-quality animation production than about the trials and tribulations of RenderMan per se. But there's some of the later, so don't be disappointed.

"RenderMan" is a word which has a lot of overlapping connotations. Most people know that RenderMan is a very high-end 3D image synthesis program which is sold by Pixar. Fewer know better that RenderMan is a family of compatible 3D image synthesis programs sold by Pixar, one of which is particularly popular, that being the very high-end one. Fewer still know that, in fact, RenderMan is a 3D geometry file format which can be output by several programs, but is only read by that very high-end 3D image synthesis program (and perhaps that clone I heard about on the net). The very elite, however, know that RenderMan is actually a scene description definition which allows modelling programs to transmit to image synthesis programs a detailed enough description of a scene to make a photorealistic rendering of the scene.

In fact, RenderMan is all these things. The "RenderMan Interface" is the scene description definition, the "RenderMan Interface Bytestream" is the metafile format which encapsulates the data described in that definition, and "PhotoRealistic RenderMan" is that famous head-of-the-product-family image synthesis program which interprets the description.

But despite the variety of pieces of the RenderMan puzzle, they are all in support of a single fundamental philosophy. RenderMan is a paradigm for image synthesis which states that users should be able to command their renderers to produce the highest quality pictures possible, and that the scene description itself should not be a bottleneck preventing this.

In 1988, Pixar developed and published the RenderMan Interface Specification with the goal that it would contain enough descriptive power to accommodate advances in modelling, animation and rendering technologies for years to come. It has done a remarkably good job, and while it is growing a little long in the tooth after 7 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 renderers which fulfill the spirit of RenderMan will generate images of the highest quality, both in their simulation of the 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*, *Forrest Gump* and *Beauty and the Beast*. The striking results that these and other films have achieved through the use of RenderMan is why we are here today.

However, RenderMan is not a complete solution. It is just the renderer. Animation production includes a lot of other programs, and bits of lore, and endless hard work. In order to work well, Ren-

derMan must fit seamlessly into an entire production pipeline. Alas, this is not always simple. For a variety of technical, political, economic, ecological and pedagogical reasons, the choice to use RenderMan as your renderer leads you down a road fraught with hard, hard work. RenderMan is not well integrated into most user-friendly animation packages, and the customizability of both the geometric and shading descriptions simply begs users to enhance their images beyond what even well-integrated systems can straightforwardly provide. On the up side, this means that people who succeed will have a distinctive look which sets them above and apart from their competition. And *this* is why we are here today. To help you succeed.

So, welcome to *Using RenderMan in Animation Production*. This course covers the theory and practice of using RenderMan to do the highest quality computer graphics animation production. The course will help you use RenderMan as a part of a whole animation production pipeline, showing you where RenderMan is valuable, where it is inappropriate, and how to build bridges between the existing tools and tricks in your production environment. We hope to teach you how enhance your scene descriptions to include information that will help the renderer make stunning pictures. We hope to teach you how to design your object appearances by thinking beyond old-fashioned shading models and texture mapping tricks. You will learn how to invent your own tricks. And we will show you what several studios who have tried and succeeded did, so that you will not be afraid to try, and will not give up until you succeed.

Our lecturers today will cover a wide variety of topics. As we've said, the course is not intended to be overly technical, but we can't get away without talking about some details. Larry Gritz will discuss the RenderMan Interface Specification, and how you get your 3D models (from whatever modelling package) into a one of the available RenderMan renderers, and what you should expect to get out the other end. Tony Apodaca will discuss the basics of programming the RenderMan Shading Language, emphasizing the bag of tricks you will need to do really great things. Tom Porter will talk about what it's like to actually write a shader used for animation production.

After this, things should get really fun. Lecturers from Pixar, Disney and Industrial Light and Magic will take you "behind the scenes" to see how animation is produced at these studios, how RenderMan is used there and how it fits into their production environment as a whole. They'll show films and videos illustrating their results.

The last section in the course notes is a short interview with the technical managers of several studios who are active RenderMan users, which discusses how RenderMan is used and abused in their studios. I hope you find it interesting.

And we may have a few other surprises during the day as well, so enjoy the course!

RenderMan: Using the Standard and its Implementations

Larry Gritz¹

The George Washington University
email: gritz@seas.gwu.edu

Abstract

There's an awful lot to say about RenderMan, and not enough time to say it. I'm going to skip the simple material that you can figure out for yourselves by reading *The RenderMan Companion*, such as the syntax of most of the various RenderMan calls. Instead, I want to try to convey some of the information that *isn't* in the books. This is the information that will transform you from being a RenderMan tinkerer to a real power user. Part of it is "cultural," tricks of the trade that are known to a small community, but not particularly documented. These course notes will try to document some of these tricks and style guidelines.

Overview

- Philosophy moment
- No Bad RIB!
- RenderMan Implementations: what, where, how
- Translators and front ends
- RenderMan resources on the net and elsewhere.
- Miscellaneous hints

Basic RenderMan Philosophy

RenderMan, simply put, is a standard interface for talking to rendering software or devices. It is similar to the way that PostScript is a standard interface for talking to printers. If you are writing a word processor, you don't need to worry about exactly what kind of printer you will be using, as long as it is PostScript compatible. When outputting PostScript, you just specify how things should appear on the page; you don't need to be concerned with the algorithms that the printer uses to draw strokes. Similarly, when using RenderMan, you can specify a 3-D scene and rest assured that a high quality image will result, without needing to know the details of how a renderer produces the picture.

1. Author's current address: Pixar, 1001 W. Cutting Blvd, Richmond CA 94804.

Basic RenderMan Philosophy

- What RenderMan is
- What RenderMan isn't
- What do you get?
- What might you get?

The RenderMan standard consists of three things: a set of procedural calls (to a C library, for example) to communicate to a renderer; an ASCII metafile format for communicating to renderers (RIB); and a special language for specifying the appearances of surfaces, lights, and volumes (Shading Language).

What RenderMan is

- A way to talk to high quality renderers
- Somewhat analogous to PostScript
- A C/C++ binding
- An ASCII/Binary metafile binding
- A Shading Language for describing surfaces, lights, volumes
- A *standard*

There are plenty of things that RenderMan *doesn't* have. It isn't a modeler. It doesn't support modeling primitives or an editable display list. It is almost completely a one-way communications stream — you can't get information back from the renderer, except for the rendered images. There's no interaction, window management, or events. The methods that the renderer employs are opaque to the user. A RenderMan-compliant renderer is only good at one thing: rendering. But you can count on it being very good at that.

When you're using a RenderMan compliant renderer, you're guaranteed to get certain features:

- A hierarchical graphics state of options and attributes (including transformations)
- Orthographic and perspective viewing transformations
- Hidden surface removal
- Pixel filtering and antialiasing
- Gamma correction and dithering before quantization

- The ability to produce images containing any combination of RGB, alpha, and depth information at a user-specified resolution
- Support of all the geometric primitives (polygons, quadrics, bilinear and bicubic patches, NURBS)
- Support of the 14 standard light source, surface, volume, and displacement shaders

As if that wasn't enough, you may also get any of the following optional capabilities (depending on the exact implementation):

- Solid modeling (constructive solid geometry, or CSG)
- Trim curves for NURBS
- Motion Blur
- Area light sources
- Displacements
- Nonlinear deformations
- Depth of field
- Texture and/or bump mapping
- Volume shading
- Ray tracing and/or radiosity
- Programmable shading in Shading Language
- Shadow depth mapping
- Environment mapping
- Multiple levels of detail
- Spectral colors
- Special camera projections

What do you get?

- Hierarchical state of options & attributes
- Viewing, hidden surface removal
- Filtering, antialiasing, gamma correction, dithering, quantization
- RGB, alpha, depth at any resolution
- Lots of geometric primitive types
- 14 standard surface, volume, displacement, and light shaders

What might you get

CSG	Shadow/environment mapping
Trim curves	Texture, bump mapping
Motion blur	Volume shading
Depth of field	Levels of detail
Displacements, deformations	Spectral colors
Area lights, ray tracing, radiosity	Special camera projections
Shading Language	

No bad RIB

RenderMan is a powerful tool. Like all powerful tools, it can be hard to use at times, and it is only useful to the extent that you know how to take advantage of its features (and learn to deal with its... non-features).

When using RenderMan, an important principle is GIGO (“Garbage In, Garbage Out”). RenderMan is only as good as the input you feed it. So we’ve called this the part of the talk “No Bad RIB,” a reference to our feeling that a lot more of the advantages of RenderMan would be available if people would just stop sending it suboptimal input.

In particular, we’ve identified several areas where people could stand a lot of improvement in their RenderMan usage. Life would be better if everybody would:

No Bad RIB!

- Use the client library
- Use curved surfaces
- Use real hierarchies
- Use shaders effectively & efficiently
- Use motion blur
- Use shadows and reflections well

Use the client library

The Procedural API

The RenderMan Interface specification defines two ways to talk to a RenderMan-compliant renderer. First, there is the RenderMan Procedural API. This is a library of subroutines which may be called from any C or C++ program. The functions all start with the prefix **Ri**, for example:

```
#include "ri.h"
RiBegin (RI_NULL);
/* All RenderMan calls go in here */
RiEnd ();
```

This actually illustrates the method of initializing the RenderMan interface and eventually shutting it down. All other RenderMan calls should be made after **RiBegin** and before **RiEnd**. The function prototypes for all the RenderMan API functions, as well as the defined constants and tokens (the all-caps tokens starting with **RI_**) can be found in the header file “**ri.h**”.

There are basically three types of RenderMan calls:

- Setting the *graphics state*, which includes two components: *Options*, which control the look of the entire scene (e.g., output resolution and sampling rate); and *Attributes*, which apply to individual geometric primitives (e.g., color, geometric transformations).
- Declaring geometric primitives. When a primitive is declared, it inherits the current attribute list. Subsequent changes to attributes do not effect previously declared geometry.
- Control (begin/end pairs and other miscellany)

All of the functions of the RenderMan API are well documented in both the RI standard document and the book *The RenderMan Companion* by Steve Upstill. Please refer to these sources for information on the structure, syntax, and semantics of programs which make calls to the RenderMan procedural API.

RIB - RenderMan Interface Bytestream

In addition to the procedural API, there’s an ASCII metafile format called RIB (RenderMan Interface Bytestream). There’s nearly a one-to-one correspondence between procedural calls and RIB. For example, the following program fragment:

```
RiAttributeBegin ();
  RiTranslate (3, 4, 2);
  RiSphere (1, -1, 1, 360, RI_NULL);
RiAttributeEnd();
```

will generate the following RIB:

```
AttributeBegin
Translate [3 4 2]
Sphere 1 -1 1 360
AttributeEnd
```

There are several important points to note about RIB syntax:

- RIB doesn’t have an equivalent to **RiBegin()** and **RiEnd()**, since these are denoted by the beginning and ending of the RIB file itself.
- Procedural calls which take a variable number of arguments (such as **RiSphere**, above, whose argument list is terminated by the **RI_NULL** token) have RIB equivalents which terminate when the next RIB directive token is reached (in this case, **AttributeEnd**).
- Arrays are delimited by square brackets [].

The two main RenderMan implementations available are stand-alone programs which expect RIB as input. Though it’s possible to have the procedural calls invoke a linkable renderer directly, it’s usually easier to generate RIB and render it later (perhaps distributed over a network).

Ways to talk to RenderMan

- Procedural API
 - a library of C language subroutines
- RIB - RenderMan Interface Bytestream
 - ASCII/Binary Metafile
 - One-to-one correspondence to API calls
 - Used as input to standalone renderer
- The procedural API may just produce RIB (“Client Library”)
- Please use the Client Library!

Types of API Calls

- Changes to graphics state
 - Options: apply to entire scene
 - Attributes: bind to geometry
 - includes surface appearance & transformations
 - Can be pushed and popped
- Declare geometry
 - Binds to current attributes
- Miscellaneous control

Using the procedural API

```
#include "ri.h"
RiBegin (RI_NULL);
/* Options for all frames */
RiFrameBegin (fnum);
/* Options for this frame */
RiWorldBegin ();
/* attribs, geometry */
RiWorldEnd ();
RiFrameEnd ();
RiEnd ();
```

Using the Client Library

In “real life,” most of us don’t type in RIB by hand, any more than we type PostScript by hand. Every once in a while, it’s useful to type in a short RIB file to test one thing, but in general, it will be your software that creates RIB.

Some people have their software write RIB directly with `printf`, but this really isn’t a good idea for several reasons: (1) you can easily write invalid or non-conformant RIB; (2) no error checking is done at compile or run time you won’t catch it until you feed the bad RIB to a renderer. The better solution is to make the procedural calls (as if you were talking directly to the renderer), but actually link to a library which outputs correct RIB for each procedural call. This is called the RIB *client library*. Pixar distributes such a library, *librib.a*, with their software. In fact, you can even get the client library source for free, for the purposes of porting it to new platforms. BMRT also comes with a library, *libribout.a*, which serves the same purpose.

An important point we want to emphasize today is that you should *use the client library*! Don’t try to output RIB by yourself. To emphasize this, all the other examples I give today will use the procedural API rather than RIB. You should consult with the RenderMan 3.1 spec if you really need to know the RIB equivalents.

Here’s an example of how to compile and link with the client library (assuming a fairly standard flavor of UNIX):

```
cc myprog.c -o myprog -lrib -lm
```

That’s pretty simple, isn’t it?

Use the Client Library

- Make the API calls in your C program
- Link with *librib.a* (PRMan) or *libribout.a* (BMRT).
- Feed the RIB to a RenderMan-compliant renderer
- Keeps you from generating incorrect RIB and suffering later

Use curved surfaces*Philosophy of RenderMan primitives*

RenderMan, being a specification for describing the rendering of scenes, supports *rendering* primitives (polygons, NURBS), not *modeling* primitives (sweeps, extrusions, surfaces of revolution). The RenderMan interface is supposed to be the means by which a modeling system talks to a rendering system, it isn’t meant to actually be a good description of models. But you, as the modeling/animation system author, need to be aware of the types of rendering primitives that RenderMan supports, and the trade-offs involved in using them.

Curved vs. Flat

Some people like to represent all of their surfaces with polygons. This is not the RenderMan philosophy. Polygons should be used to represent *flat* objects. Curved objects should be represented by curved surface primitives.

Examples of things that should be represented by polygons:

- floors
- walls, usually
- the top of a rectangular table

Examples of things that should use other RenderMan primitives:

- pretty much everything else

Yes, this is almost a religious philosophy. But it offers several advantages: efficiency of space to store the model, rendering efficiency (depending on the implementation), high precision description of surfaces in a scale- and resolution-independent manner, and aesthetics.

Use Curved Surfaces

- Use appropriate geometry
- Lots of fun surfaces to choose from

Things to use polygons for

- Floors
- Walls (usually)
- The top of a flat table

Things not to use polygons for

- Just about everything else

Why not use polygons?

- Curved surface primitives describe surfaces more efficiently than polygons
- Curved surface primitives may render much faster than polygons
- Adaptive/unlimited geometric detail

Geometric primitives

- Polygons
 - simple convex, polyhedra, concave with holes, polyhedra with holes
- Quadrics (and almost-quadrics)
 - sphere, cone, disk, cylinder, hyperboloid, paraboloid, torus
- Bilinear & bicubic patches and patch meshes
- NURBS (possibly trimmed)

Review of Primitives

Polygons. Polygons are useful for representing surfaces that are *supposed* to be flat, so RenderMan supports several kinds:

- Simple, convex polygons: **RiPolygon(int nverts,...)**
- Simple polyhedra: **RiPointsPolygons (int npolys, int nverts[], int verts[], ...)**
- Concave polygons with holes: **RiGeneralPolygons (int nloops, int nverts[], ...)**
- Concave polyhedra: **RiGeneralPointsPolygons (int npolys, int nloops[], int nverts[], int verts[], ...)**

You can look in *The RenderMan Companion* for exact information on how to use these, but here's an example of the simplest kind of polygon definition:

```
RtPoint pts[3] = { {0,0,0}, {3,4,0}, {3,0,0} };
RiPolygon (3, "P", pts, RI_NULL);
```

After giving the number of vertices, you can specify the information about the polygon, as token/value pairs, terminated by **RI_NULL**. The only required parameter is "P" (or the defined token **RI_P**), which specifies that the next argument is a pointer to the vertex positions. You can also specify "N" (or **RI_N**), followed by normals (to get a Phong shaded polygon), "Cs" for surface color, or any other varying parameter which you want to specify at the vertices.

One of the big problems with polygons is that there is no obvious parameterization across the surface, which is important if you intend to use the parametric coordinates to control surface appearance in a shader. The RenderMan interface specifies that the (u,v) surface coordinates on a polygon are simply the (x,y) coordinates in the object space of the polygon. This can be a problem, particularly if the polygon is not parallel to the x-y plane. You can solve this partially by explicitly giving "st" coordinates for each vertex of the polygon:

```
RtFloat stcoords[3][2] = { {0,0}, {1,0}, {0,.5} };
RiPolygon (3, RI_P, pts, RI_ST, stcoords, RI_NULL);
```

Even this is not the best of solutions, since it's not obvious how to map interpolated coordinates to interior points of the polygon. This is especially problematic for polygons with more than four vertices. Yet another reason to avoid polygons!

Remember that polygons follow the left hand rule in RenderMan. Be careful to keep the points coplanar, or there's no telling what artifacts you may get from a renderer.

Quadrics. Several types of quadrics are offered:

- Sphere: **RiSphere (radius, zmin, zmax, thetamax, ...)**
- Cone: **RiCone (height, radius, thetamax, ...)**
- Disk: **RiDisk (height, radius, thetamax, ...)**
- Cylinder: **RiCylinder (radius, zmin, zmax, thetamax, ...)**
- Hyperboloid: **RiHyperboloid (RtPoint point1, RtPoint point2, thetamax, ...)**
- Paraboloid: **RiParaboloid (radmax, zmin, zmax, thetamax, ...)**
- Torus: **RiTorus (majorrad, minorrad, phimin, phimax, thetamax, ...)**

Patches & NURBS. Bilinear patches are quads which are not required to be planar. They can be a substitute for polygons (in addition to being more flexible for other things), and have the added bonus of a straightforward, explicit parameterization. Here's the basic syntax:

```
RiPatch (RI_BILINEAR, RI_P, points, RI_NULL);
```

A pointer to four points is expected, but don't forget that the vertex ordering is different than for polygons! Polygons use left hand rule (vertices are numbered clockwise), but bilinear patches are given in the order: (u=0,v=0), (u=1,v=0), (u=0,v=1), (u=1,v=1).

Bicubic patches are specified similarly:

```
RiBasis (RiBezierBasis, RI_BEZIERSTEP, RiBezierBasis,
        RI_BEZIERSTEP);
RiPatch (RI_BICUBIC, RI_P, points, RI_NULL);
```

Instead of a 2x2 array of points (for bilinear patches), a 4x4 array of control points is needed for bicubics. The **RiBasis** sets the basis matrix and stepping rate (for meshes) for the u and v directions. RenderMan supplies the basis matrices for Hermite, Catmull-Rom, Bezier, and B-Spline bicubics. But you can set your own basis matrix by referencing a 4x4 array of floats, and you don't have to use the same basis in the u and v directions. The current basis is part of the attribute list, so it is pushed and popped with the rest of the attributes.

Both bilinear and bicubic patch *meshes* can be used, which really just means a grid of patches which are jointed seamlessly. Here's the basic syntax:

```
RiPatchMesh (type, nu, unwrap, nv, vwrap, ...)
```

The *nu* and *nv* are the size of the control point array in the u and v directions, *unwrap* and *vwrap* are either **RI_PERIODIC** or **RI_NONPERIODIC** (depending on whether you want the mesh to wrap in each direction), and the optional parameters must include the control points.

Tip: though the usual method of giving control points is to pass a "P" token/value pair, you can also use **RI_PW** (or "Pw") to give 4-vectors in homogeneous coordinates (I haven't found a use for this yet), or give just the z values if you pass the **RI_Z** (or "z") token/value pair. The latter is an easy way to specify height fields.

Finally, there are NURBs, Non-Uniform Rational B-splines. NURBs are really hard to understand, and well beyond the scope of this talk. But they sure are useful and flexible surfaces, so here's an example:

```
RiNuPatch (int nu, int uorder, float *uknot, float umin, float umax,
          int nv, int vorder, float *vknot, float vmin, float vmax, ...)
```

The control points get passed with the optional parameters, most likely with the **RI_PW** token, which means each control point is a 4-vector where the 4th component is the "weight." You could pass 3-vectors with **RI_P** also, but that would give you a nonrational patch, which eliminates part of the flexibility of NURBs. On some RenderMan implementations (currently only *PRMan*), you may be able to specify trim curves, which let you cut out parts of the NURBs surface:

```
RiTrimCurve (int nloops, int ncurves[], int order[], float knot[],
            float min[], float max[], int n[], float u[], float v[], float w[])
```

Use real hierarchies

Use Real Hierarchies

- Attribute and transformation stacks

```
RiAttributeBegin()
```

```
RiAttributeEnd()
```

```
RiTransformBegin()
```

```
RiTransformEnd()
```

- Make usable hierarchies
- Name your objects

The attribute and transformation stacks

As each geometric primitive is declared, it is *bound* to the current graphics state, which includes information on surface color and properties, geometric transformations, etc. You can think of the graphics state as a stack which can be pushed and popped. The current graphics state is the top of the stack. You can save the graphics state by using

```
RiAttributeBegin ();
```

This pushes all of the surface information *and* the geometric modeling transformation. Further changes to the graphics state will effect subsequent geometry, up until you reach:

```
RiAttributeEnd ();
```

which restores the graphics state to its previous condition when the corresponding **RiAttributeBegin()** was executed.

Similarly, there are **RiTransformBegin()** and **RiTransformEnd()**, which push and pop the modeling transformation (but not the rest of the graphics state).

Attribute and Transformation blocks may be nested arbitrarily deeply, but may not overlap. For example, the following is illegal:

```
RiAttributeBegin ();  
RiTransformBegin ();  
RiAttributeEnd ();  
RiTransformEnd ();
```

Making usable hierarchies

Conceptually, an articulated figure is often arranged as a hierarchy of links. This should be reflected in your rendering output. Rather than make a flat list of objects, use the graphics state stack to your advantage. Have the structure of the Attribute nesting reflect the tree-like structure of your articulated figure. This means allowing children to inherit the graphics state of their parent links and specifying only the differences in transformations between children and their parents (which is more natural anyway).

Doing this makes the RIB files smaller and more efficient, may be more efficient in terms of memory usage by the renderer, and makes the files easier to inspect (either by human or computer).

Named objects

There exists a recommended facility for naming objects. Look at the following code fragment:

```
char *name = "forearm";
char *group = "bodyparts";
RiAttributeBegin (RI_IDENTIFIER, "name", &name, RI_NULL);
RiAttributeBegin (RI_IDENTIFIER, "shadinggroup", &group, RI_NULL);
```

While this won't make any difference in the rendered image, you do get two benefits from putting such statements inside your Attribute blocks. First, if the renderer has an error while rendering, this may allow it to tell you the name of the primitive which produced the error. Second, a sufficiently smart modeler (or cleverly written script) may be able to use these hints to make modifications in a RIB file to specific named objects, or to collections of objects in the same shading group (for example, changing the surface shader used by all of those objects).

Naming Objects

```
RiAttribute ("identifier", "name",
            &name, RI_NULL);
RiAttribute ("identifier",
            "shadinggroup", &name, RI_NULL);
```

Why?

- Easy inspection of RIB files
- Hints for modelers / model massagers
- Meaningful errors from the renderer

Using shaders effectively and efficiently

Effective & Efficient Shading

- Overview of shading
- Don't forget RiDeclare()
- Shader space is your friend

Overview of shading, types of shaders

[slide - shading process]

Each geometric primitive may have five shaders associated with it:

- Displacement - can alter the surface position and/or normal in order to simulate bumpy or rough surfaces.
- Surface - determines the color and opacity of each surface point, as viewed from the camera.
- Atmosphere - describes how the color and opacity are attenuated by the “material” between the eye point and the object.
- Interior - determine how light is attenuated by the inside volume of a solid object when it is reflected/refracted.
- Exterior - determine how light is attenuated by the outside volume of a solid object when it is reflected/refracted.

Primarily, you’ll be worrying about displacement and surface shaders.¹

In addition, there are Light shaders, which describe how light sources distribute their energy into the environment. And finally, Imager shaders apply to filtered pixel values just before quantization. (Okay, there are also Transformation shaders, but so far no RenderMan compliant implementations support them.)

Types of shaders

- *Surface* - calculate color, opacity
- *Displacement* - change P, N
- *Atmosphere* - volumetric effects
- *Interior, Exterior* - apply to solids, implicitly called by trace()
- *Light* - determine energy from lights
- *Imager* - operates on filtered pixels
- *Transformation* - nonlinear transforms, unimplemented

Most rendering systems have a fixed set of surfaces and lights that you can choose from. You just vary the parameters to them, or use texture maps and such to try to add complexity. RenderMan allows you to extend the functionality of the rendering system by writing your own shaders in a C-like language called Shading Language (SL for short). This is covered in the next lecture, so I won’t go into it much, but there are some considerations you need to make for shaders when declaring your geometry.

First, the basics: The shaders are specified using the following routines:

```
RiSurface (name, ...);  
RiDisplacement (name, ...);
```

1. Note: interior and exterior shaders only apply to solid (CSG) objects, and only apply to calls to the SL *trace()* function. Therefore they are not applicable to renderers which do not support ray tracing, such as *PRMan*.

And so on, I'm sure you get the idea. The "..." is where you send any parameters to the shader, as *token/value pairs*, terminated by a NULL token. For example, here's a sample surface shader declaration:

```
float Kd = 0.5, Ks = 0.5;
char *texname = "mytexture.tex";
RiSurface ("paintedplastic", "Kd", &Kd, "Ks", &Ks,
          "texturename", &texname, RI_NULL);
```

Notice that you pass pointers to the data. Strings are especially tricky: I didn't pass *texname* as the value, I passed its address. Not doing this properly is a frequent mistake (which I make all the time). Remember your C: *texname* is a character pointer which points to the name of your texture, but *&texname* is the address of that character pointer. So you can't do this:

```
RiSurface ("paintedplastic", "texturename", "mytexture.tex",
RI_NULL); /* wrong! */
```

Why? Because "mytexture.tex" is really the address of an array, in other words a (char *). But RiSurface wants a (char **) as the value for a string argument. So you need to have a (char *) point to the texture name, then pass the address of that pointer to RiSurface. Got it?

RiDeclare

It's important to know that just because you wrote a shader, the renderer doesn't necessarily understand the semantics of your variable names and types. Whenever you use a named shader parameter which isn't one of the parameters to the 14 standard shaders, you need to let the renderer know its type by using RiDeclare:

```
RiDeclare (name, type);
```

For example, if you have the following shader,

```
surface mine (float lumpy = 1)
{
    /* do stuff */
}
```

Then when you specify that a geometric primitive uses this surface, the full sequence of events should be:

```
RiDeclare ("lumpy", "float");
RiSurface ("mine", "lumpy", &lumpval, RI_NULL);
```

Using RiDeclare()

- API/RIB has no knowledge about nonstandard shader parameters
- RiDeclare is the mechanism to give this information:

```
RiDeclare (name, type);  
RiDeclare ("lumpy", "float");
```

How does shader space work, using shader space to scale & adjust shading of objects.

It's probably not good to have your shaders operate in world or camera space. While this may be adequate for static images, once you animate them, you'll quickly see the problem.

[Example video showing the difference between world and shader/object space]

Using shader space

- "shader" space is the current transform when a shader is requested
- Shaders should use "shader" space rather than "world" or "camera"
- This allows for scaling & transformation of shader independent of object
- Uses: avoiding clones, scaling, deregistration

"Shader space" is the coordinate system in effect when the **RiSurface**, **RiDisplacement**, or other shader statement was called. You typically want to move this coordinate system around with your objects. That means you want code that looks something like this:

```
/* Don't put your shader declarations here */  
RiTranslate (object position);  
RiRotate (object orientation);  
RiSurface ("mysurf", RI_NULL); /* Put it after the transformation */  
RiSphere (...); /* Declare your geometry now */
```

The neat thing is, you don't even have to use the exact same transformation for your shader as your surface. Look at the following code fragment:

```
RiTranslate (object position);  
RiRotate (object orientation);
```



```

RiTransformBegin ();
    RiScale (2, 2, 2);
    RiSurface ("mysurf", RI_NULL);
RiTransformEnd ();
RiTransformBegin ();
    RiTranslate (10, 0, 0);
    RiDisplacement ("mydisp", RI_NULL);
RiTransformEnd ();
RiSphere (...); /* Declare your geometry now */

```

Now we've given the surface and displacement shaders each their own transformations (which are actually children of the transformation for the object). In this case, we've scaled the space of the texture *independently* of the texture of the object. Also, we've added an offset to the displacement shader space. Why would we want to do this? Suppose you have two objects made of the same material, say "greenmarble". If you put the objects side by side, they will look *identical*, since each point on both objects has the same shader space coordinates. But if you translate the shader space differently for each object, they will look like two different objects made from the same material, rather than two carbon copies of the very same object.

In addition, you can animate the transformations of the shader space, to achieve a variety of effects.

Liberal use of shader transforms

```

RiAttributeBegin ();
    transform object
RiTransformBegin ();
    transform surface shader
    RiSurface ("mysurf", RI_NULL);
RiTransformEnd ();
RiTransformBegin ();
    transform displacement shader
    RiSurface ("mydisp", RI_NULL);
RiTransformEnd ();
    declare geometry
RiAttributeEnd ();

```

Motion blur

Why is motion blur important?

Film projects at 24 frames per second. Video is a little shy of 30 frames (or 60 fields) per second. That rate is sufficient to fool you into thinking that you are viewing a continuous image, as long as the images are sampled properly. If you are shown 24 or 30 frames per second, and each frame is a perfectly sharp static image, you will see a terribly objectionable artifact known as *strobing*. This is a form of temporal aliasing, and is really ugly.

A real camera has a shutter which opens for a certain length of time while exposing the film, then closes. If objects that the camera is viewing are moving during this time, they will appear blurred in the resulting image. This is an important visual cue, and prevents the ugly strobing artifacts.

Some people will try to tell you that you can simulate motion blur as a kind of post-process on the rendered images. These people are wrong. For a variety of reasons, any attempts to solve the temporal aliasing problem *after* performing hidden surface removal is doomed to failure (though they may be barely adequate for certain types of very restricted motion). You have to take motion into account *while* solving for visible surfaces.

Other people will tell you that you can solve the problem by rendering several sharp images and somehow averaging them to come up with a final frame. They're wrong, too (although as the number of images approaches infinity, the average looks pretty good).

So the only way to do temporal antialiasing properly is to do it as part of the rendering process. Which means that the renderer needs to know how objects are moving or changing over time.

[VIDEO: motion blur vs. strobing example]

Motion Blur

- You're probably rendering for 24 or 30 frames per second
- Strobing is an objectionable artifact (show video)
- To fix, simulate finite shutter speed
- CANNOT be done as post-processing

Motion blur according to the standard

In RenderMan, you can specify changing conditions using motion blocks:

```
RiMotionBegin (int ntimes, float time1, ... float timeN);  
RiStatement (at time 1);  
...  
RiStatement (at time N);  
RiMotionEnd();
```

Pretty simple, right? You just specify a list of N key times, and then make N identical RenderMan calls, with the parameters to the call varying for each key time.

The type of calls made inside the motion block can be:

- Transformations: `RiTransform()`, `RiConcatTransform()`, `RiPerspective()`, `RiDisplacement()`, `RiDeformation()`, `RiTranslate()`, `RiRotate()`, `RiScale()`, `RiSkew()`
- Geometry: `RiSphere()`, `RiCone()`, `RiCylinder()`, `RiHyperboloid()`, `RiParaboloid()`, `RiDisk()`, `RiTorus()`, `RiPolygon()`, `RiGeneralPolygon()`, `RiPointsPolygons()`, `RiPointsGeneralPolygons()`, `RiPatch()`, `RiPatchMesh()`, `RiNuPatch()`, `RiBound()`, `RiDetail()`
- Shading: `RiColor()`, `RiOpacity()`, `RiLightSource()`, `RiAreaLightSource()`, `RiSurface()`, `RiInterior()`, `RiExterior()`, `RiAtmosphere()`

These attributes which bind to geometry as it's declared can be pushed and popped with the rest of the graphics state.

Finally, you need to specify an option that gives the opening and closing time of the camera shutter:

```
RiShutter (float opentime, float closetime);
```

Your articulated figures will be nicely antialiased if you use hierarchies as I described before, and blur all the transformations down the hierarchy.

Motion Blur with RenderMan

```
RiMotionBegin (ntimes, time1, ... timeN);
  RiStatement (at time 1);
  ...
  RiStatement (at time N);
RiMotionEnd ();
```

- You can transform transformations, geometry, shading.
- Set shutter times as an option:

```
RiShutter (opentime, closetime);
```

Motion blur in the implementations

Unfortunately, it's really hard to motion blur certain types of things effectively. So the existing implementations all have certain restrictions.

Pixar's Quick RenderMan (*qrman*) and Vector RenderMan (*vrman*), and BMRT's *rendribv* and *rgl* do not support motion blur. You can specify motion blur, but they will render unblurred images using only the first statement in each motion block.

Pixar's PhotoRealistic RenderMan (*PRMan*) and BMRT's *rendrib* renderer both support only motion blocks with exactly *two* motion keys specified. *PRMan* can blur transformations and geometry, but not shading information. *Rendrib* can only blur transformations, not geometry or shading information. Because *PRMan* uses shadow depth maps, objects will not cast blurred shadows, even if the occluding objects are moving. But *rendrib* will cast blurred shadows for moving objects.

Motion Blur -- the catch

- *qrman*, *vrman*, *rendribv*, *rgl*
all do not support motion blur
- *prman* - can blur geometry, transforms
 - can't blur shading info
 - can't blur shadows or reflections properly
 - only linear, 2 key time blur
- *rendrib* - can blur transformations
 - can't blur geometry, shading info
 - but *can* do shadows, reflections
 - only linear, 2 key time blur

Here's an example of blurring a sphere. Both its transformation and its geometry (the *thetamax* parameter) are blurred:

```
RiShutter (0.0, 0.0166667);  
...  
RiMotionBegin (2, 0.0, 0.0166667);  
  RiTranslate (2, 0, 0);  
  RiTranslate (2.4, 0, 0);  
RiMotionEnd();  
RiMotionBegin (2, 0.0, 0.0166667);  
  RiSphere (1, -1, 1, 90.0, RI_NULL);  
  RiSphere (1, -1, 1, 110.0, RI_NULL);  
RiMotionEnd();
```

Remember that when specifying the motion of the vertices of polygons or patches, ***the number of loops and vertices must match!***

There are differing philosophies about how much to blur. If you have frames every 1/24 second, how long should you blur? A real camera has the shutter open for about half the frame spacing time. Some people try to duplicate this when rendering. This is probably a good idea if you are combining CGI with live action. But if your film is completely CG, there's no reason why you shouldn't leave the shutter open longer, in effect achieving better temporal antialiasing than a real camera.

Shadows and reflections

Shadows and reflections are funny things. They are important visual cues when used in moderation, but can be extremely expensive to calculate correctly. Unlike many rendering systems, shadows and reflections don't happen automatically with RenderMan. You need to get your hands a bit dirty to add these cues.

Shadows and reflections

- Important visual cues
- Expensive to calculate
- People tend to overdo it
- Harder in RenderMan than other systems
- Very implementation dependent

Some general tips for using shadows and reflections:

- Don't shadow any more lights than you have to. Sometimes if you just cast shadows from the brightest light source, you can get all the shadow cues you need.
- Some objects will not cast shadows (for example, floors). Try to take this into consideration (e.g., by not rendering them for shadow maps).

- It's easy to go overboard with reflections. In the real world, very few objects actually have mirror-like surfaces. Use them sparingly.
- Don't get too uptight about accuracy. Using a shadow map or environment map is usually a good enough approximation, and is much faster than ray tracing.

Shadow/Reflection Tips

- Don't shadow more lights than necessary.
- Use knowledge of objects which don't occlude.
- Use reflections sparingly, fake it when possible.
- Don't get uptight about accuracy for a frame that is visible for 1/24 second.

Exactly how to add shadows and reflections to a scene is very much implementation dependent. Which leads us directly to the next topic...

RenderMan Implementations

RenderMan Implementations

- PhotoRealistic RenderMan (PRMan)
- Blue Moon Rendering Tools (BMRT)

PhotoRealistic RenderMan

The first, and most famous, RenderMan compliant renderer was Pixar's *PhotoRealistic RenderMan* (often called *PRMan* for short). *PRMan* is very popular for high end production work, and it has been used for rendering in feature films such as *Jurassic Park*, *Terminator 2*, and *The Abyss*.

Features

PRMan supports all of the standard features, plus the following optional capabilities: Solid modeling (CSG), Trim curves for NURBS, Motion Blur (but only linear as described earlier), true displace-

ment shading (the surfaces actually get bumpy), depth of field, texture mapping, programming in Shading Language, shadow depth mapping, and environment mapping.

PRMan does not support, or only partially supports, the following features

- Area light sources (just plops down a point light at the current origin).
- Volume shaders are only partially implemented.
- Nonlinear deformations (deformation shaders not supported).
- Imager shaders are not implemented (except for one built in imager shader).
- Ray tracing and radiosity are not supported at all.
- Multiple levels of detail and spectral colors are not supported at all.

PRMan's features & nonfeatures

PRMan does:

- CSG
- Trim Curves
- Motion blur (linear)
- Depth of field
- Displacements
- Shading Language
- Shadow maps
- Environment maps

PRMan doesn't do:

- Area lights
- Ray tracing
- Radiosity
- Imager shaders
- Volumes (sort of)

Algorithm info, timing

Basically speaking, *PRMan* dices all the geometric primitives into grids of *micropolygons*, which are approximately pixel sized quads. The shaders are invoked at the vertices of the micropolygons, then a kind of extended, jittered depth buffer is used to resolve hidden surfaces at screen space sample points. These samples are then filtered and quantized, and output as an image. This process is known as the REYES algorithm, and is described in a 1987 SIGGRAPH proceedings paper by Cook, et al.¹

The REYES algorithm is a “scanline” method. It does *not* involve ray tracing at all. Nevertheless, it's extremely high quality, and many of the effects you can get from a ray tracer can be simulated using shadow and environment maps. Actually, most people using real ray tracers tend to go overboard on things like reflections, so maybe it's a good thing that they're a little tricky to do using *PRMan*.

1. Cook, Robert L., Loren Carpenter, and Edwin Catmull. The Reyes Image Rendering Architecture. *Computer Graphics*, 21(4):95-102, 1987.

PRMan's algorithm - REYES

- Dice primitives into grids of pixel-sized micropolygons
- Shade at micropolygon vertices
- Perform hidden surface removal
- Filter, output image
- Remember, PRMan is not a ray tracer!

In any case, the algorithm has some interesting implications, largely stemming from the fact that surfaces are shaded *prior* to hidden surface removal.

- Surfaces which are hidden from view (e.g. behind other objects) are shaded “unnecessarily.” You should try not to specify surfaces that you know won’t be seen, and try to give a large ShadingRate to objects that are occluded, so the renderer won’t waste time shading surfaces that will not appear in the final image.
- One object may be resolved before another is declared, or objects may be culled if not visible. When an object is shaded, the system may not have much information on other objects. This means that you can’t easily compute shadows or reflections.
- Atmosphere shaders in *PRMan* have $I = P - E$, in other words, they shade assuming that *all* surfaces are visible from the camera. It has no way of knowing at shading time if something else occludes, so it can’t properly set I to the space *between* surfaces along the viewing path.

PRMan's artifacts

- All surfaces are shaded, even if hidden
- No info on other surfaces -> no shadows/reflections
- Incomplete volume shading
 - no illuminance inside volume shaders
 - assumes $I = P - E$

Sometimes people complain that *PRMan* is slow. Well sure, if you’re comparing it to a simple Z-buffer based polygon renderer. But actually, for its quality and features, *PRMan* is pretty speedy. It sure beats ray tracing. Or does it? I did some time trials for the next section.

Shadows and Reflections

PRMan doesn't ray trace. So you need to do some trickery to make shadows and reflections.

Shadows with PRMan

- PRMan is not a ray tracer, so shadows are not automatic.
- Basic idea:
 - Render depth image from POV of light
 - Use the shadow map when rendering final scene

For shadows, the basic strategy is to create a kind of texture map, which is a *depth* image from the point of view of the light source. Then when rendering the scene, you can use a light source shader which checks the depth of the surface being shaded to the corresponding depth value in your shadow map. If the surface is farther away than the corresponding lookup to the depth map, it's in shadow, otherwise it's lit. Using this facility changes your basic rendering into the following (schematically):

for each shadowed light source:

Render the scene as a depth image ("zfile") from the point of view of the light
convert the depth image to a shadow map (RiMakeShadow or txmake)

Render the scene from the point of view of the camera

When rendering the depth image from the point of view of the light, remember that you should use an orthographic projection for a distant light (since the rays don't diverge), a perspective projection for a spot light (with fov = the spotlight's cone angle), and a perspective projection with fov = 90 in each of the six directions for a point light (if you want it to cast shadows in all directions).

As an example, here's the actual code from my animation system which sets up the shadow map for a spot light (ignore the peculiar C++ syntax of my animation system classes -- you'll get the gist):

```
void SpotLight::make_shadowmap (void)
{
    float fov = degrees (coneangle*2);
    int jitter = 0;
    char filename[256], filename2[256];
    strcpy (filename, shadmap_filename);
    strcat (filename, ".z");
    strcpy (filename2, shadmap_filename);
    strcat (filename2, ".tex");
    RiFormat (shadmap_res, shadmap_res, -1);
    RiHider (RI_HIDDEN, "jitter", &jitter, RI_NULL);
    RiPixelSamples (1, 1);
    RiShadingRate (1.0);
}
```



```

RiPixelFilter (RiBoxFilter, 1.0, 1.0);
RiDisplay (filename, "zfile", RI_Z, RI_NULL);
RiIdentity ();
RiProjection (RI_PERSPECTIVE, RI_FOV, &fov, RI_NULL);
Point up, right;
PlaceCamera (pos /* from */, pos-Point(0,0,1) /* interest */,
    Point(0,1,0) /*up vector*/);
RiWorldBegin ( );
    /* ask all the objects to render themselves for the shadow maps */
    for (AFSObject *var = object_list; var; var = var->nextptr())
        var->render (RENDER_SHADOWMAP);
RiWorldEnd ( );
RiMakeShadow (filename, filename2, RI_NULL);
}

```

Then, I just use the “shadowspot” light shader, making the call in the following manner:

```

RiDeclare ("samples", "float");
RiDeclare ("width", "float");
RiDeclare ("shadowname", "string");
RiLightSource ("shadowspot", RI_FROM, &from, RI_TO, &to,
    RI_INTENSITY, &intensity, RI_LIGHTCOLOR, &lightcolor,
    RI_CONEANGLE, &coneangle, RI_CONEDELTAANGLE, &conedeltaangle,
    "samples", &shadmap_samples, "width", &shadmap_width,
    "shadowname", &fileptr, RI_NULL);

```

Here are some important tips for making and using shadow maps:

- Render your shadow maps as square images with resolution a power of two, use one sample per pixel, turn off jittering, use a box filter with a width of 1.
- Sometimes shadow map resolutions need to be very high. I rarely am able to make a decent shadow map with resolution any less than 1024x1024.
- Playing with the number of samples can reduce noise, and increasing the width makes more of a “penumbra” effect (makes the shadow edges softer).
- If objects seem to self-shadow, try fiddling with the shadow bias parameter:

```
float bias0 = 0.01, bias1 = 0.02;
RiOption ("shadow", "bias0", &bias0, "bias1", &bias1, RI_NULL);
```

When checking the shadow map, *PRMan* will choose a bias value randomly distributed between the bias0 and bias1 values. Choose too small, and your objects will incorrectly self-shadow. Choose too large, and the shadows will appear detached from the objects. You’ll probably have to choose these values on a scene-by-scene basis. It’s okay to choose bias0=bias1.

- When rendering the depth images, don’t render objects that you know won’t cast any shadows! Examples include walls, ceilings, floors, objects known to not be lit by that particular light source, etc.

PRMan shadow tips

- Render shadow maps:
 - square, power of two resolution
 - 1 sample per pixel, no jittering
 - box filter (width 1)
- Rarely will < 1k x 1k be sufficient
- Vary “samples”, “width” for penumbra

PRMan shadow hints

- Self shadowing?
`RiOption ("shadow", "bias0", &b0,
 "bias1", &b1, RI_NULL);`
- Don't render surfaces which are known not to shadow!

Reflections are done similarly. For flat surfaces which are reflective (like a polished floor), you can render the scene with the camera *below* the floor (and the floor not rendered), then use that *reflection map* as a texture lookup.

Curved objects are a little more difficult. You basically render six different views (in each direction) from the point of view of the reflected object (approximately at its center, and with the object not in the scene). These six images can be combined into a single environment map using `RiMakeCubeFaceEnvironment`. Then a surface shader can access this environment map to “fake” reflections.

Here's a schematic representation of the process:

for each reflective object x :

 Render the scene (all except object x) in all six axial directions, from approximately the position of x .

 Combine these six images into one environment map (`RiMakeCubeFaceEnvironment`)

 Render the scene from the point of view of the camera

Again, to make this crystal clear, here's an example of the routine inside the loop above, from my actual working animation software:

```

void make_reflimage (Point from, Point dir, Point up, int res, float
fov, char *name)
{
    int jitter = 1;
    RiFormat (res, res, 1.0);
    RiHider (RI_HIDDEN, "jitter", &jitter, RI_NULL);
    RiPixelSamples (2, 2);
    RiShadingRate (0.25);
    RiPixelFilter (RiGaussianFilter, 2.0, 2.0);
    RiDisplay (name, RI_FILE, RI_RGB, RI_NULL);
    RiIdentity ();
    RiProjection (RI_PERSPECTIVE, RI_FOV, &fov, RI_NULL);
    PlaceCamera (from, from+dir, up);
    RiWorldBegin ( );
    /* Ask all the lights to declare themselves */
    for (AFSObject *var = object_list; var; var = var->nextptr())
        var->declare_light ();
    /* Ask the objects to render */
    for (var = object_list; var; var = var->nextptr())
        var->render ();
    RiWorldEnd ( );
}

void
AFSMakeEnvironmentMap (char *filename, Point pos, int res, float
fov)
{
    static Point X(1,0,0), Y(0,1,0), Z(0,0,1);
    make_reflimage (pos, X, Y, res, fov, "px.tif");
    make_reflimage (pos, -X, Y, res, fov, "nx.tif");
    make_reflimage (pos, Y, -Z, res, fov, "py.tif");
    make_reflimage (pos, -Y, Z, res, fov, "ny.tif");
    make_reflimage (pos, Z, Y, res, fov, "pz.tif");
    make_reflimage (pos, -Z, Y, res, fov, "nz.tif");
    RiMakeCubeFaceEnvironment ("px.tif", "nx.tif", "py.tif", "ny.tif",
                                "pz.tif", "nz.tif", filename, fov,
                                RiGaussianFilter, 2, 2, RI_NULL);
}

```

Reflections with *PRMan*

- Again, no ray tracing, so we fake with reflection & environment maps
- Flat reflector (e.g. shiny floor):
 - render with camera on the other side of the reflective surface
 - use the reflection image as texture when rendering final scene

There are a few problems with this scheme: (1) the reflections aren't quite accurate, since all they're all computed from one point on the object, rather than seeing the view differently from each point on the surface; (2) objects won't "self-reflect", i.e. the body of a shiny teapot will reflect the world around it, but not its own spout; (3) mutually reflecting objects are very tricky and may require multiple passes.

But most of the time, the environment mapping scheme works okay, it's usually faster than ray tracing, and real scenes rarely have much mirror-like reflection anyway. In a still image, it's easy to see the flaws, but animations go by so quickly that most folks will never notice that the spout isn't reflected in the teapot body.

Reflections with *PRMan*

- Curved reflectors:
 - Render 6 images from POV of object
 - Make env map
(`RiMakeCubeFaceEnvironment`)
 - Use env map in shader for reflective object
- Problems:
 - reflections not quite accurate
 - no self-reflecting
 - mutual reflection difficult

Smart use of `RiShadingRate` and `RiPixelSamples`

One important RenderMan call to keep in mind when using *PRMan* is **`RiShadingRate()`**. This call takes a single floating point number which specifies how often a surface should be shaded. It is expressed in the pixel area corresponding to each shading sample, so 1.0 means to shade approximately once for each pixel, and 0.25 indicates that each shading sample should cover a portion of the surface which projects to about a quarter of a pixel. For test renderings when speed is essential, set this value to a high number (perhaps 4.0). Probably 0.25 is high enough quality for final renderings.

The shading rate is an attribute, so it can be specified on a per-object basis. You can speed rendering time by setting a high value (meaning a low shading frequency) for objects whose surface features don't change quickly, and especially for objects known to be hidden from view or occluded, since any time spent shading objects not visible is wasted time.

You should not set a high frequency shading rate (i.e. a low number) in order to eliminate aliasing in shaders. If written properly, your shaders should antialias themselves at any frequency, and this process will be described in the next talk.

RiShadingRate (float val)

- The pixel area which each shading sample projects to.
- Attribute: can be set per-object
- Large value: low rate (good for test images)
- Small value: high rate (0.25 fine for final image)
- Set to very low rate for objects not visible

The shading rate should not be confused with the pixel samples (set by **RiPixelSamples()**), which specifies the image plane sampling frequency for visible surface determination. Again, a value of **RiPixelSamples(1,1)** is fine for fast test images, but will produce jaggies and other sampling artifacts. A value of (2,2) is probably good enough for antialiasing geometric edges (getting rid of jaggies), but may need to be set even higher if you are using motion blur, depth of field, or other effects. This is something you'll have to experiment with on a per-image (or per-animation) basis.

RiPixelSamples(xsamp,ysamp)

- Raster sampling rate for visible surfaces
- Don't confuse with shading rate
- Use low values for test images (1,1)
- (2,2) good enough for jaggies
- Higher may be necessary for depth of field, motion blur

Blue Moon Rendering Tools

There's another implementation of the RenderMan standard out there: the Blue Moon Rendering Tools (BMRT for short), which is shareware and was written by me when I was a graduate student. We've included BMRT on the course notes CD-ROM for SGI, HP, NEXTSTEP, and Linux.

BMRT really comes with three renderers: *rendrib*, a full implementation of the RenderMan standard that uses ray tracing and radiosity; *rendribv*, an X11-based wireframe previewer of RIB files and animations; and on SGI only, *rgl*, a GL-based polygon previewer of RIB files.

BMRT

- Full implementation of RenderMan
- Shareware -
<ftp.gwu.edu/pub/graphics/BMRT>
- SGI, HP, NEXTSTEP, Linux binaries in the course notes!
- Ray tracing & radiosity

Features and how invoked

The high quality renderer included with BMRT is called *rendrib*, and it's a full implementation of the RenderMan standard which also supports the following optional capabilities:

- Solid modeling (constructive solid geometry, or CSG)
- Depth of field
- Motion Blur (with pretty much the same limitations as *PRMan*, as described earlier).
- Area light sources are correctly supported.
- Texture mapping, environment mapping
- Volume shaders and imager shaders are implemented fully and correctly.
- Ray tracing and radiosity are supported.
- Programmable shading in Shading Language

But *rendrib* does not support, or only partially supports, the following:

- True displacements don't work. You can use displacement shaders, but *rendrib* will convert them to bump maps.
- Trim curves for NURBS aren't implemented.
- Shadow depth maps are not supported, but of course you can easily use the automatic ray cast shadows for this.
- Multiple levels of detail, Spectral colors, Special camera projections, and Nonlinear deformations are not supported.

BMRT - feature diffs from *PRMan*

- Ray tracing
- Radiosity
- Correct area lights
- Correct volume & imager shaders
- NO true displacements (bump maps instead)
- NO trim curves for NURBS
- NO shadow maps (ray tracing instead)

How it's the same as PRMan

BMRT can read the same RIB and .sl files that *PRMan* does. Instead of invoking *prman* or the “render” script which comes with *PRMan*, you invoke the *rendrib* program. *Rendrib* takes a bunch of command line arguments, which can give additional control over output resolution and sampling, and other runtime options. These are all explained in the docs.

How it's different from PRMan

Rendrib and *PRMan* support different sets of features of the RenderMan specification, mostly due to their different underlying algorithms. *Rendrib* actually does do ray tracing (and radiosity, too). That means it's much slower than *PRMan*, but you can do a lot of cool things with it that *PRMan* just can't do. For example, *rendrib* supports the following features not found in *PRMan*:

- True ray tracing using the SL *trace()* function.
- Automatic shadows using ray casting.
- Area light sources.
- Imager and volume shaders are fully implemented and work properly.
- Radiosity, which is a method for calculating diffusely interreflected light in a scene.

There are some incompatibilities, too, just because they were developed in different places with different goals. All the things that are part of the spec, such as RIB and SL, are pretty much the same in the two implementations. But some things which are implementation-specific to *PRMan* are not done in the same way. Here are some things to look out for:

- BMRT will not read compiled shader files (*.slo) which were compiled using Pixar's compiler. Instead, BMRT expects shaders to be compiled using its own shader compiler, *slc*, which results in *.so files. The original .sl source is, of course, the same.
- *PRMan* has a proprietary format for texture files. If you want to texture map a TIFF file using *PRMan*, you need to make a call to **RiMakeTexture()** or run the *txmake* program. But BMRT takes TIFF files directly as texture maps (which I think is much more convenient), and cannot read Pixar's proprietary texture format.
- BMRT doesn't use shadow maps, and can't read the depth images created by *PRMan*. But usually, that isn't a big loss, since you can do shadows automatically using ray casting.

One thing that BMRT just can't do is true displacement shading. You can write displacement shaders just like you would for *PRMan*, but BMRT will automatically turn it into a bump map, which means it will just alter *N*, but not actually move *P*. Much of the time, this is okay. Sometimes, if you scrutinize the images, you'll notice that the bumps don't self-shadow like they do with *PRMan*, and silhouettes are too smooth. And some displacement shaders just won't look right at all.

BMRT doesn't (yet) support Trim Curves for NURBS. And shading rate is effectively ignored.

BMRT/*PRMan* incompatibility

- BMRT won't read *.slo files (use "slc" to compile shaders)
- BMRT won't read *PRMan*'s texture files (but will read TIFF directly)
- BMRT won't read *PRMan*'s shadow maps (but you probably don't need to)

Shadows and Reflections

Probably the most important difference in the way that BMRT and *PRMan* are used is in how to specify shadows and reflections. Compared to *PRMan*, shadows and reflections are childishly simple to do with *rendrib*. For shadows, you just need to declare a particular attribute before the light source declaration:

```
char *shadowstate = "on";
RiDeclare ("shadows", "string");
RiAttribute ("light", "shadows", &shadowstate, RI_NULL);
RiLightSource (...);
```

The default is for shadows to be off. This state can be pushed and popped on the attribute stack just like anything else. This can be done for any light source, including area lights. There's no need to use special shaders, such as "shadowspot.sl".

With *rendrib*, partially transparent objects can cast partial shadows, and colored transparent objects can cast colored shadows. This is something that just can't be done with depth map shadows.

You can specify different attributes for geometric objects which control how the geometry casts shadows, using the following attribute:

```
char *how;
RiAttribute ("render", "casts_shadows", &how, RI_NULL);
```

the pointer *how* points to a string which contains one of the following:

- "none" The object does not cast shadows.
- "opaque" The object blocks light completely if it is hit by a shadow ray.
- "Os" The **Opacity** value should be used for the transparency of the object.
- "shade" Invoke the object's surface shader at each shadow hit point to determine how opaque it is at that spot.

You should set this value to the first one on the list that you can get away with, i.e. the most efficient. Objects that don't cast shadows should be marked as such, just like you wouldn't render those objects for shadow maps with *PRMan*. Objects with complex surface shaders that vary their opacity across the surface need "shade", which is the most expensive option. This can be controlled on an object-by-object basis.

Shadows with BMRT

- No brainer:

```
char *shad = "on";
RiAttribute ("light", "shadows",
&shad, RI_NULL);
RiLightSource (...);
```
- Individual objects may cast no shadows, full shadows, or shade for shadow values.

As for reflections, all you need to do is use a shader which uses the SL *trace()* function. BMRT comes with a couple, such as shiny.sl and gmarbtile_polish.sl. Here's the shiny.sl shader as an example:

```
surface
shiny (float Ka = 1, Kd = .5, Ks = 1, Kr = .5, roughness = 0.05;
      color specularcolor = 1;)
{
    point R;
    point Nf = faceforward (normalize(N), I);
    point IN = normalize (I);
    color ev = 0;
    if (Kr > 0) {
        R = normalize (reflect (IN, Nf));
        ev = trace (P, R);
    }
    Oi = Os;
    Ci = Os * (Cs * (Ka*ambient() + Kd*diffuse(Nf)) +
              specularcolor * (ev + Ks * specular(Nf,-IN,roughness)));
}
```

You can still use environment maps with BMRT, though. This is especially useful when compositing CG elements into live action.

Reflections with BMRT

- Just use a shader that uses *trace()*
- Example:


```
R = reflect (I,N);
Ci = Ka*ambient() + Kd*diffuse(N)
    + Ks*specular(N,I,rough)
    + Kr*trace(P,R);
```
- Can still use environment maps (esp. for combining CG & live action)

Algorithm and timing info

BMRT's *rendrib* renderer uses distribution ray tracing, and also progressive refinement radiosity (if you ask for it). The BMRT user manual explains in detail how to invoke these features.

Conventional wisdom is that scanline rendering methods (such as the algorithm used by *PRMan*) are significantly (orders of magnitude) faster than ray tracing. But is this really the case? I decided to perform an experiment.

I tried an example scene, a kitchen model. First, I rendered using no reflections or shadows, just giving the identical input to both *prman* and *rendrib*. At 640x480 resolution, 4 samples per pixel, and shading rate of 0.25, *prman* rendered the image in 182 seconds, compared to 706 s. for *rendrib*. That means that *prman* is almost 4 times faster than *rendrib* (this is roughly in line with times other people have reported to me). The images were nearly identical in quality.

But is this really a good test? The model is probably fairly typical of the complexity in an animated scene, but generally we're going to want some shadows and maybe a reflection or two. So I generated two RIB files, one meant for *prman*, which calculates a shadow map and three environment maps, and another for *rendrib*, which uses ray cast shadows and reflections. *PRMan* took 740 seconds, compared to 2008 for *rendrib*, which closes the speed gap to under a factor of 3. But viewing the images, I felt that the *prman*-generated image was not of the same quality, so I tried it again using higher resolution shadow and environment maps. At the point where I felt that the *prman*-generated image was of the same quality as the *rendrib*-generated image, it took 1341 seconds for *prman* to render the entire image, including generation of the shadow and reflection maps. This means that it is rendering at about 1.5 times the speed of *rendrib*.

Finally, I went all-out with *rendrib*, and had it use area light sources and preprocess the scene with 100 steps of progressive refinement radiosity, using a minimum of four samples per form factor calculation for non-emitters (and a minimum of 16 samples for emitters). *Rendrib* took 2109 seconds, which is only about two minutes longer than without radiosity, and the results were noticeably better. There's no equivalent for *prman*, so I can't make a direct comparison on the times.

It's hard to tell if this is really typical, and I don't like to extrapolate from a sample size of one. On the other hand, I did not select this scene because I thought it would be particularly hard or easy for either of the renderers, so at least this is an unbiased sample. In any case, it appears that the speed of

an efficient ray tracing solution is very possibly within a factor of two of *prman*'s speed. So it's not at all impractical to use a RenderMan-compliant ray tracer, at least partly, for production work.

Timing Experiment - Kitchen

- Same file, no shadows or reflections
 - *prman*: 182 s. *rendrib*: 706 s.
- Shadow & env maps for *prman*, ray tracing for *rendrib*:
 - *prman*: 740 *rendrib*: 2008
 - *prman*: 1341 (to match *rendrib* quality)
- Using area lights and radiosity:
 - *prman*: N/A *rendrib*: 2109

Combining *PRMan* and *BMRT*

A lot of people use both implementations, depending on the job they have to do. When speed is critical, or you just have to have true displacements or trim curves, you must use *PRMan*. If you need ray tracing or radiosity, area lights, or certain volumetric effects, and you can't fake them with textures, you have no choice but to use *BMRT*.

BMRT vs. *prman*

Usage heuristics:

- speed, true displacements, trim curves
 - > *PRMan*
- Ray tracing, radiosity, area lights, volumetric effects
 - > *BMRT*

But there's nothing that keeps you from using both, particularly if you are compositing separate elements into your final images. For example, suppose you have one object that just has to have, say, dispersive refraction, and it's just too hard to do using *PRMan* (but simple with *rendrib*). You can render the rest of the scene quickly using *PRMan*. Then render the scene using *rendrib*, but only render that object (don't specify the other objects, or make them matte objects), and composite the two images together for your final frame.

BMRT and PRMan

- Can't decide? Use both.
- No reason not to render time-critical with *PRMan*, individual objects with BMRT.
- Composite them together.

Front Ends and Translators

I've tried to compile a list of modelers/animation packages/converters which support the RenderMan interface in some way. Much of this information was lifted from the comp.graphics.rendering.renderman FAQ. Some of it came directly from the producers of the software mentioned, but much is second-hand information, so it's worth contacting the companies directly if you need more information.

SGI machines

Alias

Alias PowerAnimator (also for SGI), versions 5 & 6, output RIB, including NURBS with trim curves.

SoftImage

Animal Logic (Australia) sells a product called Softman, which reads Softimage scene files and converts for RenderMan-compatible output. It has a real nice shader interface with all params animatable. Automatic shadow and reflection generation, and texture creation from different file formats, individual object shading control, motion blur on object motion, vertex, and camera, all in a nice Motif interface. Contact Animal Logic 61 2 906 1232 or support@dl.oz.au.

Lost In Space makes a product called Siren which converts SoftImage scenes into RIB files. Info is available from siren@lostinspace.com. It runs on SGI's with IRIX 4.0.5 and later, for US \$2000, with bulk and educational discounts available. You need a copy of the SoftImage Developer's Kit, and also Pixar's RenderMan Toolkit in order to use Siren. Siren makes an attempt to convert models, cameras, surface descriptions, etc., into a format which can be fed to a RenderMan-compliant renderer.

Wavefront

The new version of Wavefront's software can input RIB models, but cannot yet output RIB. This capability may be coming soon.

WaveMan, from Mind's Eye Graphics, Inc. in Richmond, VA, converts Advanced Visualizer models to RenderMan. Their phone number is (804) 643-3713; email mindseye@inf.net.

Vertigo

Vertigo's Animation Machine software (modeler & animation system) has very nice support for RenderMan output, but it's only a polygon based modeler. You can generate RIB files or render directly. Vertigo comes with the *PRMan* runtime library linked in for integrated rendering, or you can buy Vertigo bundled with *PRMan*. Their phone number is 604-684-2113.

PRISMS

Side Effects PRISMS animation system for SGI's running UNIX, has a pretty complete implementation, 416/504-9876, get blurb from Janet Frasier. PRISMS 5 is polygonal, but can also do patch meshes.

Front ends and translators - SGI

- Alias - version 5&6 outputs RIB, including NURBS.
- SoftImage
 - Animal Logic: *Softman*
 - Lost in Space: *Siren*
- Wavefront - inputs RIB, does not output
- Vertigo
- Side Effects *PRISMS*

NEXTSTEP

3DReality from Stone Design Corp. (505) 345-4800 info@stone.com Runs on NEXTSTEP. A little long in the tooth and has some quirks, but it's built of dynamically loadable bundles and offers a very accessible API for adding your own shapes, tools, etc. Reads and writes RIBS just fine. Really, really good academic discounts and very friendly tech support.

Intuitiv'3d from Intuitive Systems, Inc. tel: 415-852-0245 fax:415-852-1271, info@intuisys.com. Runs on NEXTSTEP. Sports a terrific interface but is rather slow. Great real-time previews of lighting and shape, so-so modeling tools, great shader manipulation, including "MetaShaders" which store surface, color, displacement and lighting shader info in a single entity. Reads RIBS but saves to its own proprietary format (.i3dw)

solidThinking from Gestel Italia, Ph.:++39 444 964-974, Fax: ++39 444 964-984 Email: info@solid.gestel.it. For NEXTSTEP, this industrial-strength modeler from Italy approaches Alias in its power and refinement. Great modeling tools including NURBS, control of every RenderMan parameter, fast and smooth manipulation of objects and lights. Reads and writes RIBS, reads Wavefront, TDDD and DXF files too. Support for things like particle animation, 3D mice and BMRT is planned.

WavesWorld, a set of UI, modeling and animation objects available only atop NEXTSTEP, available via <http://wave.www.media.mit.edu/people/wave/> An object oriented framework consisting of two "kits" of objects and lots of examples, WavesWorld is based directly atop the RenderMan interface.

Front Ends - NEXTSTEP

- Stone Design - *3DReality*
- Intuitive Systems - *Intuitiv'3d*
- Gestel Italia - *solidThinking*
- WavesWorld

Mac & PC

Showplace & Typestry

Pixar's Showplace on the Mac can both read arbitrary RIB files in and spit out RIB files. This can be extended with their very nice plug-in interface. For example, the venetian blinds plug-in written by Dan McCoy of Pixar puts out trimmed NURBS, so even high end capabilities are supported. Somewhere in the \$200 range.

Pixar's Typestry 2 (Mac and Windows, around \$200) is specialized -- it's mainly for generating 3-D text. It puts out excellent RIB; trimmed NURBS for faces and patch meshes for the extrusions. It's more capable than you might think for a "3d type" program; you can import Adobe Illustrator files and extrude them, and then save the RIB out.

MacroModel

MacroModel for Windows and Macintosh output RIB files, and comes bundled with *PRMan*. It is a modeling program that outputs RIB files as well as providing a rather basic front end for the RenderMan Interface (shader parameter editing, light editing, and such). Note: Version 1.5 of the Windows version outputs RIB files that do not conform to the standard and require some minor editing to work with BMRT and possibly others (though PRMan seems to accept it). All lowercase interface calls must be properly capitalized, but other than that, the output seems to be ok.

Others

FastCad 3D by Evolution Computing; 437 South 48th Street, Suite 106; Tempe, AZ 85281; Phone: (602) 967-8633. FastCad 3D is a 3-D modeling system capable of producing a RIB file for rendering with a program like Pixar's RenderMan for Windows. This DOS program can produce 3D models quickly due to the fact that it is written in Assembly language.

DesignCad 3D for the Mac (\$300) and DOS (\$500) is a polygonal surface modeler which outputs RIB. Supports CSG. No direct shader support.

The Valis Group's Pixel Putty Solo for the Mac (\$349 retail, \$299 direct?) offers an extremely versatile and fluid spline-based modeler featuring nine different NURBS, lattice deformations and Boolean operations on patches, direct rendering to RenderMan using .slo shaders, and event-based key frame animation with inverse kinematics. More information can be had from Valis at: VALIS-GROUP@aol.com or 1-88-VALIS-04.

MicroStation is a general purpose CAD program which can be used as a modeler. It supports NURBS, CSG, fillets, and blends, and can output RIB. It's available for UNIX, DOS, and Windows. I believe it is actually written by Bentley Systems, but is marketed by Intergraph. Cheap academic versions are available (\$150?). Bentley Systems can be reached at: (610)458-5000 or academic@bentley.com (regarding the academic package.)

Autodesys form*Z is a CAD modeler on the Mac which goes for \$1500. Polygons, CSG, and spline meshes are supported. No direct shader support.

Strata's Studio Pro 1.0 (\$1200) for the Mac is a spline and polygon surface modeler, rendering, and animation package. Strata sells a plug-in called Rend-X which outputs RIB (\$200, runs on 68k but not PowerMac). Strata, 2 W. St. George Blvd., Suite 2100, St. George, UT 84770, phone 801-628-5218.

Syndesis Corporation sells a package called Interchange, which is a file format conversion tool. This package converts among over 30 different formats, including RIB, 3dstudio, DXF, Wavefront, Lightwave, and others. It runs under Windows. Contact Syndesis Corp., 235 S. Main St., Jefferson WI 53549. Voice 414-674-5200, fax 414-674-6363.

Alias Sketch 2.0 for the Mac is a spline and polygon surface modeler which supports NURBS but does not have shader support. Price is around \$600.

VIDI's Presenter Professional for the Mac (\$1500) is a spline/patch-mesh based modeler with excellent RenderMan support; you can manipulate shader parameters and everything.

Front Ends - Mac & PC

- Pixar's *Showplace* and *Typestry*
- MacroMedia's MacroModel
- Evolution Computing: FastCad 3D
- Valis Group: Pixel Putty Solo

RenderMan resources

RenderMan resources

- Books, etc.
- Newsgroups
- FTP sites
- WWW sites

Books, etc.

If you don't already have a copy, the best tutorial-like reference on RenderMan is *The RenderMan Companion* by Steve Upstill, Addison-Wesley, 1989, ISBN 0-201-50868-0. This book is a tutorial on the RenderMan procedural API and Shading Language.

The other important printed resource is the official RenderMan Interface Specification 3.1, which is available directly from Pixar for US\$20 (inside the US), \$25 to ship outside the US. Pixar's phone number is 1-800-888-9856.

Two prior Siggraph courses have been offered on RenderMan. "Introduction to the RenderMan Interface" was course #18 in 1990, and "Writing RenderMan Shaders" was course #21 in 1992. If you have access to these course notes, they contain a lot of valuable information. Siggraph course notes from before the CD-ROM days are notoriously hard to get if you didn't buy them at the conference, but some libraries or university graphics labs buy entire sets every year. Check with somebody you know who was at the conference and they just might have a copy.

The book *Texturing and Modeling: A Procedural Approach*, by David Ebert, Ken Musgrave, Darwyn Peachey, Ken Perlin, and Steve Worley has some of the best information on writing procedural textures. The chapters by Musgrave and Peachey have all of the examples given in RenderMan Shading Language.

RenderMan-related books

- The spec - “The RenderMan Interface 3.1”, available from Pixar for \$20
- *The RenderMan Companion*, by Steve Upstill
- Course notes from Siggraph 90, 92
- *Texturing and Modeling: A Procedural Approach*, by Ebert et al.

Newsgroups

As I’m sure you know, the USENET group *comp.graphics.rendering.renderman* has been up and running since December 1994. It’s reasonably active, often with several articles per day. But it can certainly use more traffic, particularly interesting technical discussion and “tricks” like the stuff covered in this course.

comp.graphics.rendering.renderman

- Started in December 1994
- Could use more traffic!

FTP sites

Some FTP sites we know of with RenderMan-related materials are listed below, roughly in chronological order of their first appearance.

ftp://ftp.gwu.edu/pub/graphics/BMRT

The BMRT FTP site contains the binary distributions of BMRT for the various platforms supported: SGI, NEXTSTEP, HP/9000 7xx/8xx, RS/6000, DEC Alpha, Linux.

ftp://archive.cs.umbc.edu/pub/texture

Contains the source code that goes with the book *Texturing and Modeling: a Procedural Approach*, by Ebert, et al. Shader source can be found in the *Peachey* directory, and in the file *musgrave_renderman.tar.Z*.

<ftp://ftp.pixar.com/>

The Pixar FTP site contains binaries for the client library, among other utilities.

FTP sites

- <ftp://ftp.gwu.edu/pub/graphics/BMRT>
BMRT distribution
- <ftp://archive.cs.umbc.edu/pub/texture/>
Ebert book code & examples
- <ftp://ftp.pixar.com/>
Pixar FTP site
 - Contains client library

WWW sites

We know of several WWW sites that are related to the RenderMan standard and its implementations. Sites are listed roughly chronologically from the time of their first appearance.

BMRT Home page: <http://www.seas.gwu.edu/student/gritz/bmrt.html>

The BMRT home page contains general information on BMRT, links to the FTP site to download the distribution, an “image gallery” of images created using BMRT, and pointers to related sites.

GWU Procedural texturing course:

<http://www.seas.gwu.edu/graphics/ProcTexClass/>

This is the page for a class taught by Ken Musgrave on writing procedural textures (based on the *Texturing and Modeling* book mentioned above). All the student projects can be found here, generally including shaders and RIB.

RenderMan Repository: <http://pete.cs.caltech.edu/RMR/>

Tal Lancaster’s RenderMan Repository. Contains links to other RenderMan stuff, shader source code, sample RIB files, and more. Hopefully, people will continue to contribute materials to this site.

Pixar home page: <http://www.pixar.com/>

As I write this in April, it isn’t available yet, but hopefully by the time you read this in August, it will be up and running.

Programming RenderMan Shaders

Tony Apodaca¹
Pixar

One of the great advantages of using a RenderMan renderer is the fact that you can describe the appearance characteristics of objects with as much detail and subtlety as you typically describe the shapes and positions of those objects. The RenderMan Shading Language is a special-purpose programming language for describing appearance characteristics. Shading Language programs, called *shaders*, can be used to model materials and effects in a physically realistic or in an “unrealistic” artistic style.

To describe appearance, you attach to each object a set of RenderMan shaders, which are loaded when the model is loaded, and executed by the renderer when it comes time to compute the colors of the objects in the image.

The power of the Shading Language comes from the fact that it is a fully functional programming language. You can program just about anything. This means that you are not limited to specific mathematical equations, or the old standard shading formulations published in Siggraph papers years ago.

The difficulty with the Shading Language comes from the fact that it is a fully functional programming language. You can program just about anything. This means you have to specify in great detail exactly what you want done, and making these decisions seems confusing and daunting.

This section of the course covers the groundwork for writing sophisticated RenderMan shaders to simulate natural and artificial objects and effects, such as: bricks, plants, fruit, fire, water and special light sources, in rendering styles ranging from photorealistic to cartoon style. It will help you design your shaders by thinking beyond those old-fashioned shading models and texture mapping tricks. You will learn how to invent your own tricks. No one is going to become an instant expert, but you should learn some useful techniques of how to approach shader writing problems, and should leave with the confidence that you too can write visually complex shaders to simulate interesting surfaces.

Shading Paradigm

Okay, on with the show. The RenderMan Interface provides a very rich shading paradigm for describing the objects in a 3-D scene. The paradigm has four facets which give the user powerful tools to describe appearance.

1. Portions of this lecture were originally written and delivered by Darwyn Peachey and Eliot Smyrl of Pixar.

RenderMan's Shading Paradigm

- **Available geometry**
- **Shading “white boxes”**
- **C-like programming language**
- **Shader global environment**



Available Tools

The first facet is the geometry. RenderMan supports a wide variety of geometric primitive types, including curved surfaces as well as polyhedra. There is a powerful transformation stack, which includes user-named coordinate systems. Both of these can be motion-blurred and subjected to depth-of-field. In addition, the interface supports a user-extensible set of surface attributes. The most obvious among these are color and texture coordinates, which are familiar to users of other renderers. These can actually be described in multiple ways. However, the RenderMan Interface allows the user to invent new parameters and attach them arbitrarily to the surfaces of objects. For example, one might do a physical simulation where the temperature of each polygonal vertex might be valuable information. This temperature would then be visible to the shader, which could make images based on it.

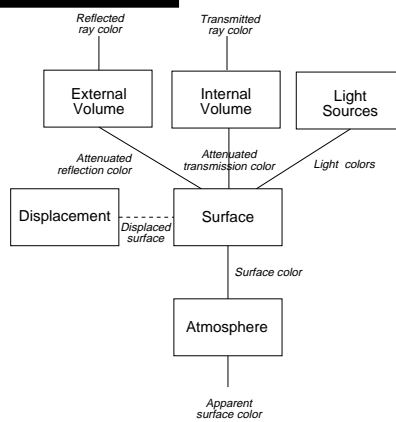
RenderMan Geometry

- **Parametric curved surfaces**
- **Polyhedra**
- **Texture parameters**
- **User vertex variables**



The second facet is the shader *white box* concept. In a black box, you give the box chosen inputs, and the black box gives you results without you knowing how it computed it. A shader white box is just the opposite. You describe the inside of the box, and the system gives you a unified set of inputs without you knowing where they came from. RenderMan describes several places in the rendering pipeline where users may influence the calculations going on within the renderer. This distinct modularity makes it much easier to isolate concepts, for portability and robustness.

Shader Evaluation Pipeline



The third facet is the C-like Shading Language which allows the user to write complex functions to fit into each white box. The Shading Language has certain features of C left out and certain new fea-

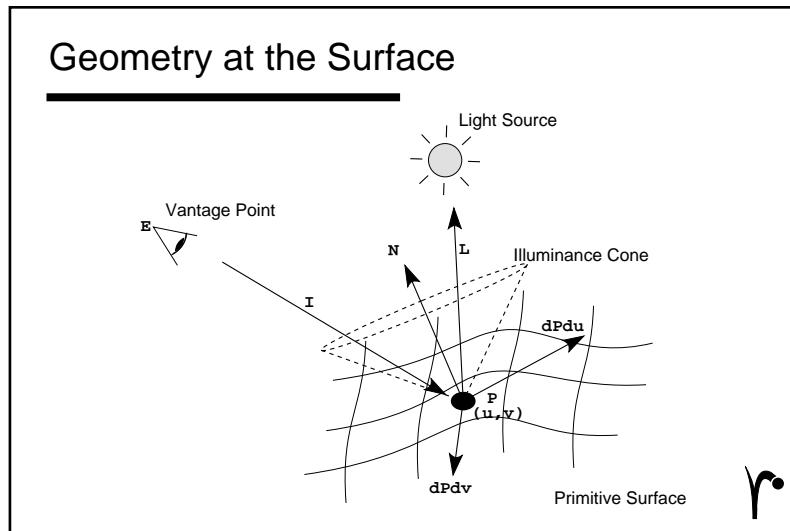
tures added in order to fit better as a special-purpose programming language designed specifically for doing the geometric and color calculations necessary for describing appearances. We are not going to spend a lot of time defining the syntax or enumerating the rich library of predefined functions available in this language. You will catch on as we go along, and the Specification and Companion provide a much more complete description than we could ever provide here. Check them out.

The fourth facet is the rich global environment provided by the renderer to the Shading Language shaders. This environment provides a wealth of information to the shader about the local environment of the point to be shaded. There is a special global state for each type of shader, providing the information relevant to that type. The information available to surface shaders is the most general.

Surface Graphics State Variables

- | | |
|--------------------------------|--------------------------------------|
| • P Surface position | • Os Surface opacity |
| • N Shading normal | • s,t Texture coordinates |
| • Ng Geometric normal | • L,Cl Light vector and color |
| • I Incident vector | • u,v Parametric coordinates |
| • E Vantage (eye) point | • du,dv Change in coordinates |
| • Cs Surface color | • dPdu,dPdv Surface tangents |





So, now we have a language to write shading routines with, a large amount of geometric data to play with in our routines, a place in the rendering pipeline for those routines to be loaded, and hopefully we already have (hidden in the backs of our minds somewhere) a basic understanding of how computer graphics works and what we want to do with it. Now all we need is a basic methodology for writing shaders, so that we can get underway, and a bag of tricks with which we can get fancy. Okay, here goes....

Methodology

The most basic and nearly universal trick to writing shaders, as with any program or subroutine, is *Divide and Conquer*. For shader writing, this means dividing the shading calculation up into four phases.

Divide and Conquer

- **Pattern generation**
- **Layers**
- **Illumination model**
- **Compositing**



The first phase is pattern generation. We will spend lots of time on pattern generation through the rest of this lecture, because for most shaders, this is the most interesting part of the shader. The second phase is layering. Most interesting patterns cannot be described by a single function. Quite often we discover that there are layers of patterns, subtle patterns on top of grosser patterns, or tiny patterns inside larger patterns. It is usually quite helpful to handle each layer separately, and merge them together into the final pattern.

The third phase is the illumination model. For many shaders, we just copy the plastic or metal illumination model, letting the color be derived from the pattern rather than a solid color. For other shaders, the illumination model itself is patterned, or perhaps it is totally non-standard because the particular surface reflects light in a way quite unlike plastic. The fourth phase is compositing. Some shaders have physical layers of totally different characteristics, and the shaders might calculate the patterns and illumination of each physical layer separately, and then composite them together using the standard compositing equations. This divide and conquer methodology lets you create much more complex shaders without losing your mind.

As mentioned above, the largest portion of the effort in most shaders is devoted to generating the patterns on the surface which makes that surface appearance unique. The word which was chosen in the ancient past to describe these patterns was *texture* (perhaps inappropriately, but it's too late to complain about it now). Let's review the standard uses of texture patterns. I use "standard" to describe any use of texturing described in published literature before 1985. The fact that most renderers in the world only let you do these (or usually some subset of these) is only a secondary consideration (wink!).

Standard Uses of Texture Patterns

- **Surface color**
- **Transparency**
- **Bumps and displacements**
- **Reflection and environment maps**
- **Parameter modification**
- **Illumination functions**
- **Projections**



The most familiar use of texture patterns is to set the surface color to an interesting pattern. This was first done by Catmull in 1974. Modulating opacity with texture patterns was introduced by Gardner in 1983. Bump mapping is the perturbing of surface normals based on a texture map. This was introduced by Blinn in 1979. The more obvious but also more difficult version of this technique is displacement mapping, which moves the surface rather than just tweaking the normal vectors. Cook produced the first displacement mapped image in 1984. Reflection maps are simple texture maps which contain the view from a virtual camera located behind a flat mirror. Environment maps are either rendered or drawn images of the entire *world sphere* — a texture which contains the global environment surrounding, but at a large distance from, the objects in the scene. These techniques were also first described by Blinn in 1979, and were used quite extensively by Williams and others at NYIT in the early 1980s.

Parameter modification is the use of a texture map to provide the value for a shading equation variable, such as K_s , rather than using a single constant value. Illumination mapping is the use of a texture map to provide the light intensity coming from various directions rather than using real light sources. This trick was used at NYIT for years to cut down rendering times before it became generally known in the mid-1980s.

Projections are a technique for choosing the texture map coordinates based on the value of some mathematical equation, rather than being limited to the predefined texture coordinates or parametric coordinates at the vertices of the model. Typically the equations used are sweeps of planes, spheres or cylinders through space, which “projects” the image onto the object in a easily calculated fashion. Barr first described it in the literature in 1984, although the technique was used extensively before that time. This technique spawned Perlin’s and Peachey’s solid textures in 1985.

In the mid-1980’s, it was not uncommon for commercial renderers to provide some or all of these features because they were simple extensions to the single shading equation that was compiled into the program. But adding such features was limited to renderer writers, a very small priesthood. The introduction of the Shading Language removed the barriers to users tweaking the shading equations

in whatever way they wanted to, and incidently made it much more difficult to get a Siggraph paper published which described a new neat feature to add to your shading equation.

Tricks

So, now let's consider some obvious tricks that came to my mind as I was writing these course notes. Several of them are hinted at above. Not that I invented any of these techniques. I call them “obvious” not because they were trivial to come up with the very first time, or that you should have thought of them in the last 5 minutes, but because they give you the flavor of the type of things that start to become simple once you remove those barriers.

Obvious Tricks

- **Splitting effects based on a test**
- **Composited texture layers**
- **Texture map containing texture coordinates**
- **Variables controlling texture in unique ways**
- **Multiple textures**
- **Using geometric info for texturing purposes**
- **Nonstandard illumination functions**



Some surfaces have different parts which are made of distinctly different materials. Consider, for example, a gold inlay on a wooden box. With the Shading Language it is quite easy to use a boolean test at every point on the surface to decide whether to compute a wood shader or a gold shader for that point.

Splitting Effects

```

if ( length(P - center) < radius ) {
    /* gold */
} else {
    /* wood */
}

```

r

This concept can be extended to creating layers of partially transparent colors, which are composited on the fly by the shader until it reaches the innermost or opaque layer. For example, the appearance of a planet might be programmed as several cloud layers over water over land.

Composited Layers

```

float trans = (1 - texture("topcloud" [3]));
Ci = texture("topcloud");
Ci += trans * texture("bottomcloud");
trans *= (1 - texture("bottomcloud" [3]));
Ci += trans * texture("water");
trans *= (1 - texture("water" [3]));
Ci += trans * color(.5, .3, .2); /* brown */
Oi = 1.0;

```

r

One interesting effect is to use a texture map to hold 2-D or 3-D coordinates for another texture map access. You could paint some interesting color ramps with odd swirls to create a warped mapping, and then texture any image onto a simple polygon using the warp.

Mapping of a Mapping

```
float new_s, new_t;
new_s = texture("warp"[0], s, t);
new_t = texture("warp"[1], s, t);
Ci = Cs * texture(texturename, new_s, new_t);
```

r

One feature which many renderers are forced to handle is pseudocoloring of some sort. Typically, this involves the user computing some function of his data and applying these color values to polygonal vertices, and the renderer has little to do but copy these colors onto the screen. In RenderMan, the user could put his *data* on the vertices of his geometry, and write a shader which uses that data either directly or indirectly to influence the color of the object.

Special Surface Variables

```
surface hotstuff (varying float
    temperature = 273;) {
    color red = color(1,0,0),
    blue = color(0,1,0),
    white = color(1,1,1);
    Ci = color spline(Cs, Cs, red, blue,
        white, white, (temperature - 273)/500);
}
```

r

Another really obvious trick is to add a special texture map to catch a particular feature which can not be handled by the equations running the rest of the appearance. For example, on the bowling pin image, there are five separate textures controlling the color of the pin. There is a texture for the un-

derlying surface color relative to the height of the pin (which also handles the red crowned section), three texture maps for the three decals applied to the surface in their particular places on the pin, and a texture which has dirt and grit which overlays the rest. There is a sixth texture which controls the pits and gouges which are part of the displacement map.

Sometimes the special texturing effect we desire can be created entirely by taking advantage of information provided in the geometric description of an object. For example, when we needed a glow around a very bright object, we thought about it for a while and realized that we needed something that was bright and opaque near the center, fading to black and transparent near the edges of a circle. Fading out in a roughly cosine or cosine squared way. Can you see this coming? The dot product of N and I on a sphere is exactly perfect!

Glow

```
color yellow = color(1,1,0);
Ci = yellow *
    pow(normalize(I).normalize(N), 2.0);
```

r.

It's quite fun to play with non-standard illumination functions. Let's consider a LP vinyl record (anyone remember them?). The surface of an LP is not really flat with lots of surface microfacets. It is much more nearly a sawtooth wave, perhaps with some plateaus between the teeth. When light hits this, it reflects very specularly in three preferred directions, rather than one, giving three distinct specular highlights and a very unique surface appearance.

Nonisotropic Specular

```
point Nf = normalize(faceforward(N, I));
point delta = normalize(dPdu);
color spec = specular(Nf, I, roughness) +
             specular(Nf + delta, I, roughness) +
             specular(Nf - delta, I, roughness);
Ci = Cs * (Ka * ambient() + Kd * diffuse(Nf)) +
        Ks * specularcolor * spec;
```

Math

The hard part of writing shaders is engineering complex effects from these primitive ones. Let's review a little of the relevant mathematics and basic graphics which you can leverage as building blocks to make more interesting things.

Everyone remembers trigonometry. Sine equals opposite over hypotenuse. Cosine equals adjacent over hypotenuse. Tangent equals opposite over adjacent. There will always be places where a few mathematical identities like the Law of Cosines or the Double Angle Equations will help you solve for a certain value. However, what is much more likely to come up in graphical programming is simple vector algebra.

Trigonometry and Analytical Geometry

- **dot product of two vectors**

scalar

$$A \cdot B = |A| |B| \cos \theta$$

- **cross product of two vectors**

perpendicular to both vectors

$$A \wedge B = |A| |B| \sin \theta$$

- **normalized vectors**



We can add and subtract vectors. The Shading Language correctly handles simple operations on the vector types `point` and `color`, so we don't need to write boring loops for each operation. In addition, the Language provides dot and cross product of `points`. Recall that the dot product of two vectors is a scalar which gives you the angle between the vectors. Vectors which point in roughly the same direction have a positive dot product, vectors which point in roughly opposite directions have a negative dot product, and we take advantage of this relation a lot. The cross product of two vectors is another vector which is perpendicular to both vectors. The magnitude of this vector is also related to the angle between.

One very important thing to remember is that in both of these relations, the length of the vectors themselves get into the equations. If we could guarantee that the vectors always had length exactly 1.0, we wouldn't have to worry too much about that. Unfortunately, the Shading Language does not automatically normalize any vectors for you, so you *must* remember to do that. Many is the time I've spent several hours trying to debug a shader which "can't possibly give me that picture!", only to find that I forgot to `normalize` some vector.

An extremely important skill required for any serious shader writer is the ability to visualize patterns in terms of shapes of mathematical functions. If you can easily recognize the pattern of veins on a banana leaf as a set of tangent functions raised to increasing powers, you will go far! Some of the functions that you will use hundreds of times per week include `sin` and `cos`, `modulo` (aka `floor`) and `absolute value`.

Simple Mathematical Patterns

- **sin, cos and friends**
- **modulo (floor) for sawtooth waves**
- **step functions and square waves**
- **absolute value**
- **algebra**



The first is the most obvious, the sine and cosine functions. They make great smooth patterns, are easy to modulate both in frequency and amplitude, and only require one function call. The second pattern is the sawtooth (aka triangle) wave. It is very easy to make this wave using the floating point modulo function provided by the language. The third pattern is the square wave. This is also easy to manufacture using step functions, and you can also control the duty cycle by choosing the appropriate offsets. And, of course, you will use an infinitude of algebraic manipulations of these to shift their origins, modify scales, and compose and combine them into more complicated functions.

Sometimes you need a curve that rises more quickly or more slowly than linear. This is quite easy to do with exponentiation. Intuitively we know that if you raise a number to a power greater than one, it gets bigger very fast, and if you raise it to a power less than one, it's like taking a root, so it gets smaller. When writing shaders, we are usually exponentiating values between zero and one, and those values work backwards from the intuition.

Exponentiation

x^y for $y < 1$

gamma

x^y for $y > 1$

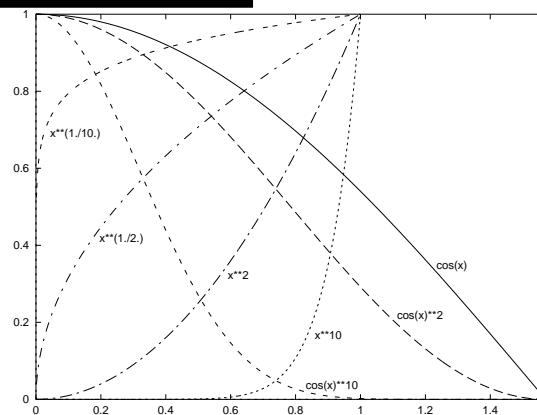
$\cos(2\pi x)^y$ for $y > 1$

specular highlights

r

We can easily see how we can generate very sharp slopes by using exponentiation on our earlier triangle waves. This is great if you want to have certain emphasized features like a sharp color band at a certain value. One of the most common uses for exponentiation is to raise the cosine function to a low power, for example, when using Phong specular highlights. I think few people have ever graphed that function to try to understand just how sharp it really is.

Behind the Power Curves



r

Quite often we find ourselves needing to interpolate some parameter between some values. There are several ways to do this, with increasing complexity and smoothness.

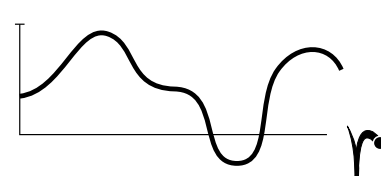
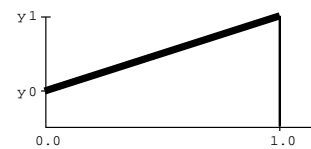
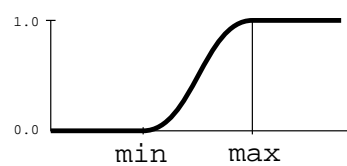
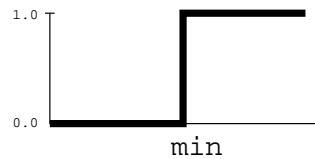
Interpolation

- **step**
- **linear interpolation**
“lerp”
- **smoothstep**
- **spline**

r

The simplest is a step interpolation. When the controlling parameter exceeds some threshold, we switch over to the higher value. In the Shading Language, this can be done with an `if` statement, or with the `step` function, depending on the situation. A slightly better way is to linearly interpolate between two values. This is often called *lerping*. If you do this in two dimensions, it is called a *bil-erp*, and in three dimensions, it is (not surprisingly) called a *trilerp*. In the Shading Language, lerps are done using the standard equation $y = y0 + (y1 - y0) t$. Since this is often done for colors, there is a presupplied shortcut function called `mix`.

Interpolations



If you want some more smoothness, such as derivative continuity on the ends, we suggest the `smoothstep` function. This is the standard cubic interpolation sometimes called *ease-in/ease-out*. We use this quite often to prevent staircasing artifacts, as you will see later in the day. Finally, if you need to interpolate smoothly between many values, there is a general-purpose Catmull-Rom interpolatory spline function `spline`, which can interpolate between floats, points or colors. Color splines are very interesting and useful for many effects effects, such as those associated with one-dimensional textures.

Before we leave the math review, I should mention the problems of transforming points and vectors. It is quite common for people to know that computer graphics uses 4-by-4 matrices to describe transformations. It is somewhat less common for people to know that these matrices are used because they allow us to take advantage of homogeneous coordinates. It is unfortunately very uncommon for people to understand what homogeneous coordinates are, and why we want to use them. Here is why: if you have a 4-by-4 transformation matrix M , you can describe all translations, rotations, scales, shears and perspective transformations within the same matrix. You can then use the same matrix to transform points, direction vectors and (with a little work) normal vectors.

Transformations

- **transform() your points**

$$p = (x, y, z, 1) M$$

- **vtransform() your direction vectors**

$$v = (x, y, z, 0) M$$

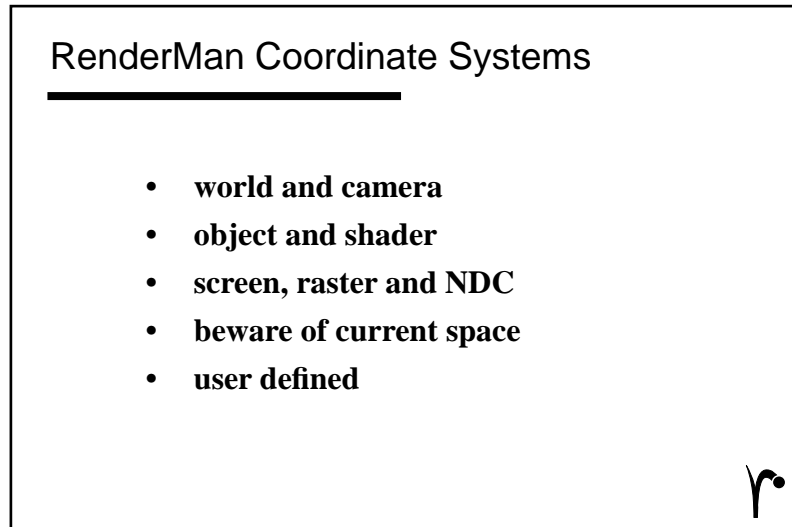
- **ntransform() your normal vectors**

$$n = (x, y, z, 0) M^{-I T}$$

✓

3-D points are transformed by sticking a 1.0 on the end of the point to make a 4-D homogeneous point. Multiply that by the matrix, divide out the w , and you have the transformed point. 3-D direction vectors are transformed by sticking a 0.0 on the end of the vector to make a 4-D homogeneous direction vector. Multiply that by the matrix, drop the 0.0, and you have the new transformed direction vector (note: not quite this simple for perspective matrices). 3-D normal vectors are transformed like direction vectors, but use the inverse transpose of the matrix. Many people try to transform normal vectors through the standard matrix, and sometimes works, but only if the standard matrix equals its own inverse transpose. This happens surprisingly often (translation, rotation and uniform scales), which is why most people think it always works.

RenderMan has three separate Shading Language functions, specifically designed for transforming these three different types of floating-point triples (plus another one for transforming colors). Be sure to use the correct one at all times!



You should spend some time learning understand why and how to use RenderMan coordinate systems. You can transform points and vectors to any space you need to in the Shading Language using the appropriate `transform` function, and it is quite often the case that certain patterns are much easier to generate in one coordinate system than in another. There are four built-in coordinate systems that are very useful. *world* coordinates are obvious, the coordinate system at `RiWorldBegin` from which everything else flows. *camera* coordinates is the space centered around the camera, with the positive *z*-axis directly ahead, *x* left and *y* up.

object coordinates is the space centered around the object. That is, the coordinate system which was active at the time the particular geometric primitive was defined. This is slightly different from *shader* coordinates, which is the coordinate system which was active at the time the shader was defined (i.e., when `RiSurface` was called). Sometimes these two are the same, but quite often the model includes transformations between the `RiSurface` and the `RiSphere`, for example.

screen space is the coordinate system in which `RiScreenWindow` is specified, so it is post-projection plane but is similar in layout to *camera*. *raster* is the pixel coordinates of the final image. *NDC* is a similar space which runs from 0.0 to 1.0 from left to right, and from top to bottom, across the whole image.

The Shading Language also includes the concept of *current* space. This abstract space is defined to be whatever the renderer finds most convenient to do its shading in. Some renderers will do shading in *world* space. Other renderers will do shading in *camera* space. In any case, it is the default space for all points and vectors, and it is not a good idea to assume that it is any particular space.

The user can also define his own *named coordinate systems* by using `RiCoordinateSystem` in the model. The shader can transform to and from any of these spaces, as well, and this can be a pow-

erful tool. For example, you may wish to do some of the calculation in the local coordinate system of one of the lights. Simply name the coordinate system when the light is defined, and then use that name in the shader.

The Final Goal

Making Your Life Easier

- **custom geometry to hold custom effects**
- **well-placed coordinate systems**
- **precalculated texture coordinates**
- **all manner of hacks and cheats**



The best programming trick is to do less programming. We have already seen that special vertex variables, or convenient geometric properties of objects, or specially designed coordinate systems can help us generate interesting effects. Therefore, it should not be at all surprising that the enterprising shader writer attempts to make his programming job easier by manipulating the geometry to present him with opportunities to use these tricks. For example, glints, glows, hairs, and other surfaces effects which occur “just off the edge” of the surface are usually done with partially-transparent geometry placed conveniently adjacent or surrounding the objects. Placing a special coordinate system conveniently around the object at some angle makes it easy to apply projections to objects at some rotation not perfectly aligned with object space.

Placing texture coordinates with a special purpose program is sometimes the only way to get exactly the effect you want on a surface. For example, objects which morph will travel move thorough texture space under most solid textures and texture projections. If instead you write a program which attaches the “unmorphed” texture coordinates onto the coordinates of the morphed object, the texture will “stick” to the object.

But after all this math and paradigm and shading language syntax is all said and done, the first and most important step in writing cool shaders is to master Jim Blinn’s *Ancient Art of Chi’ Ting*. Very roughly, the fundamental tenet of this philosophy is: “If it looks good enough, it *is* good enough.”

Ancient Art of Chi' Ting

- **Learn when simulation is appropriate**
“photo-realism”
- **Learn when hacking is appropriate**
“photo-surrealism”
- **use smoke and mirrors**



Sometimes, it is really necessary to do a full simulation of the surface in order for it to look right. People are very good at noticing when something looks wrong, particularly if it is mostly right. At other times, something that is not correct, but is somehow similar to it, is even better than correct. Such things are often called *art*. But most of the time, something that is close to right is good enough. Maybe the object is small, or moving fast, or the audience doesn't really know what it should look like anyway.

In fact, the real goal is not to make a perfect shader, but rather to make a *successful* shader.

Criteria for a Successful Shader

- **Convincing in context of whole image**
- **Reasonably efficient in terms of CPU usage**
- **Very efficient in terms of programming effort**
- **Usability and reusability**



The shader must be convincing to the audience, but as a part of the image as a whole. Shading Language programmers tend to examine shaders in isolation, on simple objects against a black background, and this usually makes us work too hard to get our shaders to be “just right”.

For pride’s sake, the shader should be reasonably efficient in terms of compute time. Useless calls to normalize unit vectors are embarrassing. But of far more importance is efficiency in programming effort. A shader has to be run a few billion times before the CPU time it uses accumulates to match the hours and days you spend programming it, and the audience has to see the animation many times before they notice the tiny glitch that you see in still frames.

On the flip side, a well written shader can almost certainly be reused, the algorithm tuned slightly to give a wide variety of effects. The better engineered it is the first time, the more likely you’ll understand it when you try to adapt it next time.

Texture Generation

Much of our effort and ingenuity in writing shaders goes into the process we call *texture generation*. Broadly speaking, “texture” is any variation in shading characteristics across a surface. You can vary any shading characteristic as some function of surface parameters. This is a consequence of the arbitrary programmability of shading calculations in the shading language. There are several basic approaches to calculating the textures values, each one applicable to certain situations. Let’s look at several methods for creating texture.

Types of Patterns

- **stored images (texture files)**
- **regular patterns**
- **pure stochastic patterns**
- **perturbed regular patterns**
- **random placement patterns**
- **perturbed access to texture files**

Texture Files

```
Ct = texture("name.tx");  
Ct = texture("name.tx", ss, tt);  
Ct = texture("name.tx", ss, tt,  
            "swidth", 2, "twidth", 4);  
Ct = environment("name.env", D);
```



Of course, simply accessing a stored texture image (texture file) is the easiest and most obvious thing to do. This is a very powerful way to get realistic or natural textures, by taking photographs of the real materials and then scanning them into your computer. Texture file access is very easy in the shading language, using either the standard texture coordinates (s, t) or arbitrarily computed ones (ss, tt). The "swidth" and "twidth" parameters can be used to blur the texture somewhat by widening the texture filter. Environment textures are indexed by a direction instead of (s, t) coordinates, and can be used to simulate reflection and refraction.

Disadvantages of Scanned Textures

- **environment of original texture**
- **limited in size**
seams, unnatural periodicity when repeated
- **limited in resolution**
pixel size of scan may be visible



It would be nice if implementing a texture consisted only of finding a sample of the desired material, photographing it, and digitizing the photograph. But this approach is rarely adequate by itself. If the material is not completely flat and smooth the photograph will capture information about the lighting direction. Each bump in the material will be shaded based on its slope, and in the worst case, the bumps will cast shadows which obscure other features. Even if the material is flat and smooth, the photograph often will record uneven lighting conditions, light source color, reflections of the environment surrounding the material, highlights from the light sources, and so on. This information generates incorrect visual cues when the photograph is texture mapped on to a scene with simulated lighting and environmental characteristics that differ from those in the photograph. A beautiful photograph often looks out-of-place when texture mapped onto a computer graphics model.

Another problem with a scanned texture image is that it has a finite size and resolution. Because its size is limited, it will be necessary to repeat the texture in order to cover a very large surface. Most scanned textures can't be repeated or "tiled" across a surface without ugly visible seams between tiles. The opposite edges of the texture don't match unless the image is carefully retouched with a paint program. Even after seams are eliminated, the texture can exhibit unnatural regularity or repetition. Prominent features in the texture image are replicated over and over and make the tiled nature of the texture obvious and objectionable. On the other hand, the texture is limited in resolution, so zooming in on the texture may reveal pixel artifacts.

So we embark on the task of learning how to generate textures procedurally, without ruling out the use of texture files where appropriate, or where we can combine texture files and procedural texture to gain the advantages of both.

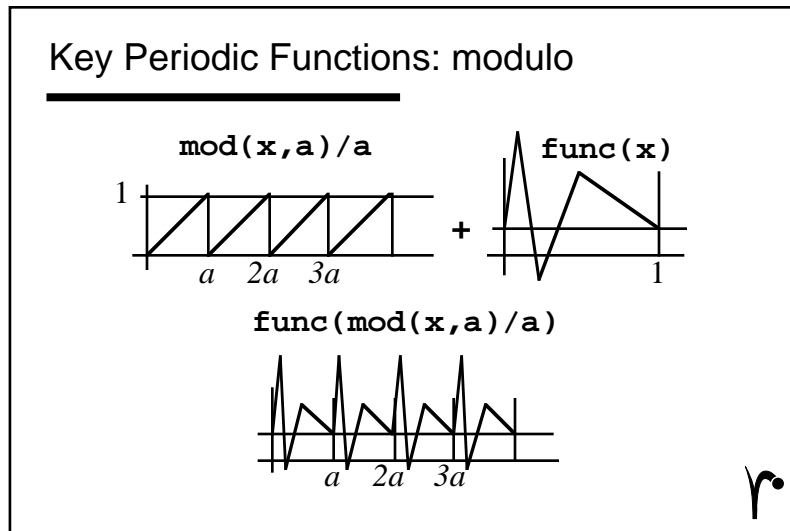
Regular Patterns

Regular patterns are those without stochastic components. They are created by the application of geometric reasoning and programming cleverness. Here are some of the most important and frequently used techniques.

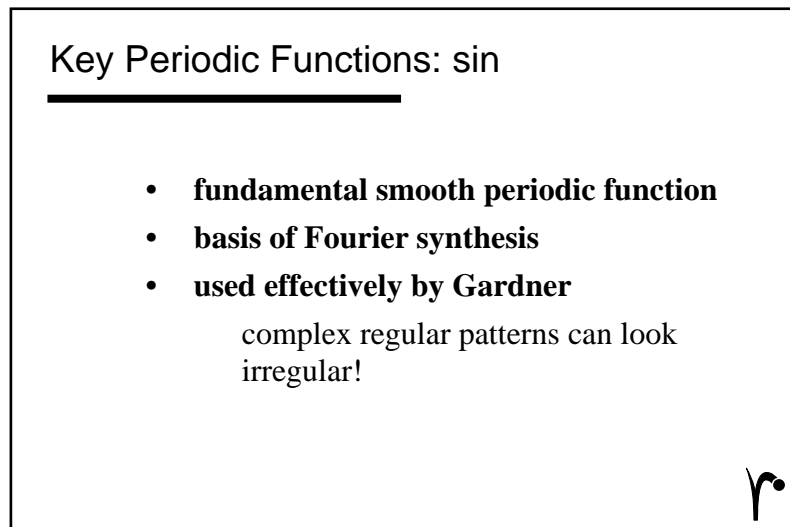
Regular Patterns

- **have no stochastic component**
- **lines, grids, checkerboards, polygons**
- **built up using standard geometric programming tricks**



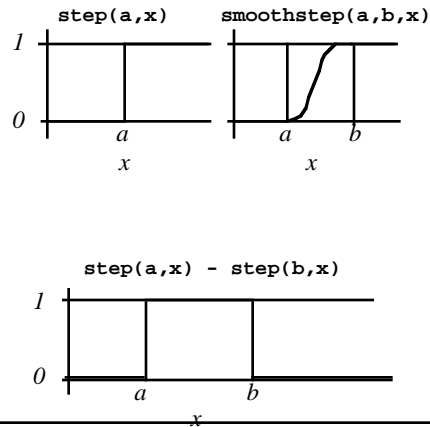


The mod function is a basic building block for periodic functions. Often we take a texture coordinate and use mod to reduce it to a periodic zero-to-one sawtooth function: $\text{mod}(x, a)/a$. Then we define our overall function as a composition of the sawtooth and a basic shape defined on the half-open interval $[0, 1)$. The resulting function is a periodic repetition of the basic shape function.



Similarly, the trigonometric functions are great sources of periodic functions. They are wonderfully smooth and have a well-understood frequency content (which becomes very important when we talk about antialiasing later). Throwing a few sine waves together can create complex, well-behaved regular patterns which nonetheless look irregular, as evidenced by Gardner's work in clouds.

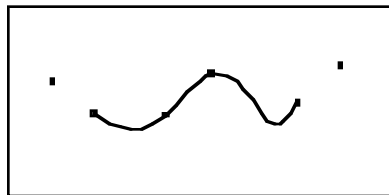
Key Periodic Functions: steps



Not infrequently, texture patterns have sharp transitions between two regions. These are best implemented as `step` functions. `smoothstep` is heavily used to produce gradual transitions rather than sharp ones, often to avoid aliasing artifacts. We'll discuss aliasing in detail later, too. One advantage of using a `step` construct instead of an `if` statement is that there are easy ways to smooth it out (i.e., `smoothstep`) in order to antialias it. Another advantage is that it avoids the pitfalls sometimes associated with the `if` statement and area operators.

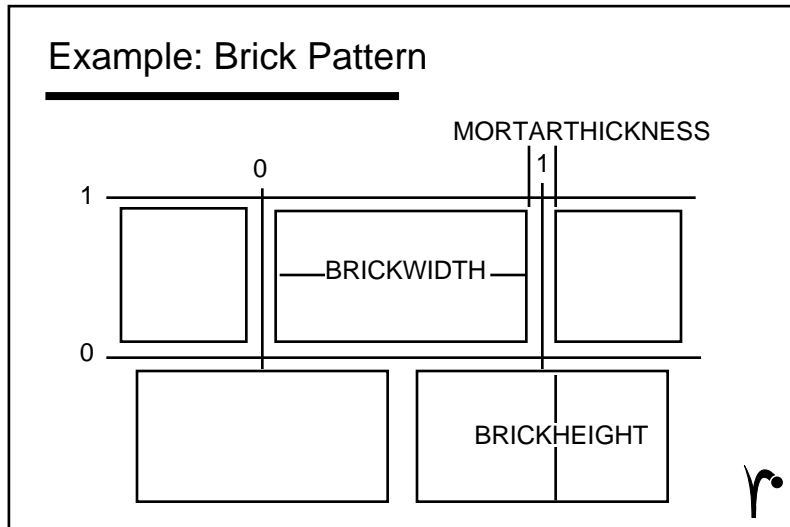
The pulse is another important functional building block. A rounded pulse can easily be built from `smoothstep` in a very similar way.

Favorite Built-In Functions: spline



- **Catmull-Rom spline through float or color values**
- **need an extra point at each end**

The `spline` function provides a good way to map a continuous float parameter into a collection of colors which change more sharply and help distinguish the different values of the parameter. This is the pseudo-color technique well-known to scientific visualizers.



Example: Brick Pattern

```

ss = scoord / BMWIDTH;
tt = tcoord / BMHEIGHT;
if (mod(tt*0.5,1) > 0.5)
    ss += 0.5;      /* shift alternate rows */

tbrick = floor(tt); /* which brick? */
sbrick = floor(ss); /* which brick? */
ss -= sbrick;
tt -= tbrick;
w = step(MWF,ss) - step(1-MWF,ss);
h = step(MHF,tt) - step(1-MHF,tt);
Ct = mix(Cmortar, Cbrick, w*h);
    
```

A small 'r' logo is in the bottom right corner.

Here's our first serious example of a regular pattern, namely, a simple brick pattern. `BMWIDTH` is the width of the brick plus the mortar, and `BMHEIGHT` is the height of the brick plus the mortar. `MWF` is the "mortar width fraction" and `MHF` is the "mortar height fraction." `tbrick` and `sbrick` are unique coordinate values for each brick. They will be useful in the "improved" versions of this shad-

er shown later. The key to the version of the brick shader on this slide is in the two pulses w and h which control the horizontal and vertical color selection between mortar color and brick color.

Here's the full shader listing (too long to put on a slide!):

```
#define BRICKWIDTH      0.25
#define BRICKHEIGHT    0.08
#define MORTARTHICKNESS 0.01

#define BMWIDTH        (BRICKWIDTH+MORTARTHICKNESS)
#define BMHEIGHT       (BRICKHEIGHT+MORTARTHICKNESS)
#define MWF            (MORTARTHICKNESS*0.5/BMWIDTH)
#define MHF            (MORTARTHICKNESS*0.5/BMHEIGHT)

surface
brick(
    uniform float Ka = 1;
    uniform float Kd = 1;
    uniform color Cbrick = color (0.5, 0.15, 0.14);
    uniform color Cmortar = color (0.5, 0.5, 0.5);
)
{
    color Ct;
    point NN;
    float ss, tt, sbrick, tbrick, w, h;
    float scoord = s;
    float tcoord = t;

    NN = normalize(faceforward(N,I));

    ss = scoord / BMWIDTH;
    tt = tcoord / BMHEIGHT;

    if (mod(tt*0.5,1) > 0.5)
        ss += 0.5;          /* shift alternate rows */

    tbrick = floor(tt); /* which brick? */
    sbrick = floor(ss); /* which brick? */
    ss -= sbrick;
    tt -= tbrick;

    w = step(MWF,ss) - step(1-MWF,ss);
    h = step(MHF,tt) - step(1-MHF,tt);

    Ct = mix(Cmortar, Cbrick, w*h);
}
```

```

/* "matte" reflection model */
Ci = Os * Ct * (Ka * ambient() + Kd * diffuse(NN));
}

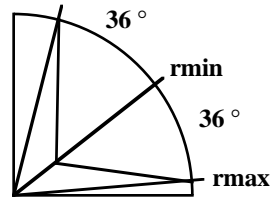
```

Example: Star

```

sa = 2*PI/npoints;
p0 = rmax*(cos(0),sin(0),0);
p1 = rmin*
    (cos(sa/2),sin(sa/2),0);
ss = s - sctr; tt = t - tctr;
angle = atan(ss,tt) + PI;
r = sqrt(ss*ss + tt*tt);
angle /= sa;
angle -= floor(angle);
if (angle < 0.5)
    angle = 1 - angle;
d0 = p1 - p0;
d1 = r*(cos(angle),sin(angle),0) - p0;
Ct = mix(Cs,starcolor,step(0,zcomp(d0^d1)));

```



The star is another regular pattern, one that looks quite hard until you think about it in polar coordinates. The diagram shows that each point of a five-pointed star is 72 degrees wide. Each half-point (36 degrees) is described by a single edge. The shader converts the parameters (s , t) into polar coordinates with respect to the center of the star, and then determines which half-point it might be in from the angle. A little linear algebra tells you which if you are inside or outside the edge for that half-point, and viola.

Stochastic Patterns

The remaining types of patterns are all “irregular,” that is, they have a stochastic component. However, the term *stochastic*, which usually implies “random,” is being used in a special sense here. Our stochastic functions are not random at all. In fact, our stochastic functions take inputs and are repeatable, deterministic functions of those inputs. This is essential to our purpose, because we need functions which remain the same from frame to frame of an animated sequence. On the other hand, we don’t want to see a visible, regular pattern in the stochastic function — it must be pseudo-random.

Stochastic Patterns

- **irregular patterns have stochastic component**
- **special meaning of “stochastic”**
- **not random! repeatable functions of inputs**
- **no apparent patterns; pseudo-random**
- **noise() function is our stochastic building block**



The `noise()` function is our basic stochastic primitive. The `noise` function was introduced by Ken Perlin in his Siggraph '85 paper “An Image Synthesizer.” Perlin and Hoffert give a more precise definition of the function on pages 255 and 256 of the Siggraph '89 paper “Hypertexture.” J.P. Lewis analyzes the deficiencies of this `noise` function in the Siggraph '89 paper “Algorithms for Solid Noise Synthesis.”

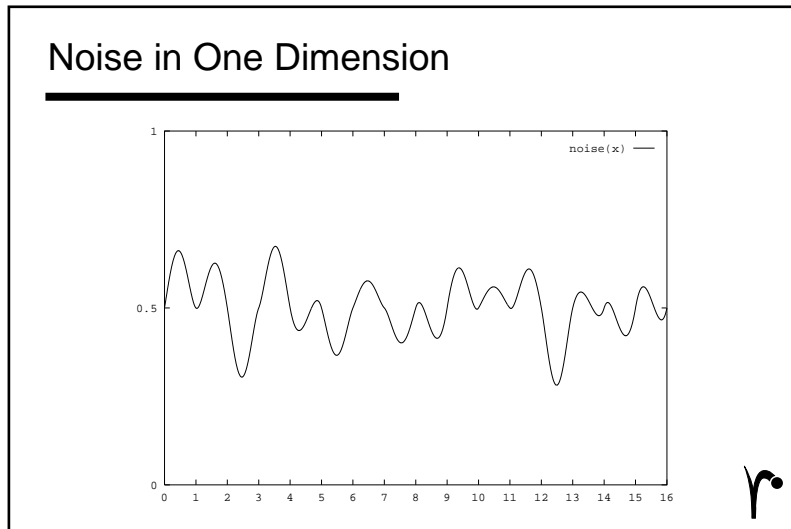
Noise Function


- **repeatable pseudo-random function**
- **1D, 2D, or 3D input**
- **float, color, or point output**
- **value ranges from 0 to 1**
- **value is 0.5 at integer locations**
- **oscillates between integer locations**
- **`snoise = 2 * noise - 1`**



Note that the RenderMan `noise` function ranges from 0 to 1 with a value of 0.5 at integer points, while Ken Perlin's `noise` function ranges from -1 to 1 with a value of 0 at integer points. It is sometimes convenient to define a signed noise function `snoise` which behaves like Perlin's function:

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



- ### Properties of Noise
- **continuous and smooth**
 - **dominant frequency of 0.5 to 1**
 - **approximately band-limited**
little high-frequency content
- 

Although the noise function is not, strictly speaking, band-limited, it does have a dominant frequency range, so it can be used to build up more elaborate stochastic functions by summing noises of different frequencies. Aliasing artifacts can be reduced by simply not including higher frequency noise components when the sampling rate is too low.

Building Stochastic Functions

- **transform noise inputs:**
offset: `noise(x + k)`
frequency: `noise(f*x + k)`
- **sum multiple components**
spectral synthesis
- **thresholding noise**
`step(0.65, noise(x))`



There are many ways of using noise. Different frequencies can be weighted and summed to produce noises with desired power spectra. Offsets and rotations of the noise parameter provide different noise values and avoid directional artifacts which sometimes appear in noise. Raw noise has its own characteristic appearance; other patterns can be created by perturbing regular patterns with a little noise. Threshold functions like smoothstep can be set up to change rapidly if noise exceeds a certain value; this creates more sharply defined edges within the otherwise smooth noise function.

Pure stochastic patterns consist of sums of noise components of various frequencies. They tend to lack any regular structure and can be called “amorphous.”

Pure Stochastic Patterns

- **“amorphous” patterns**
- **noise with specified frequency spectrum**
- **thresholding, color mapping to emphasize pattern**
- **fluid flow phenomena: vapor clouds, stone materials formed by mixing**



Several calls to `noise` can be combined to build up a stochastic function with a particular frequency/power spectrum. Repeated calls of the form

```
value += amplitude * noise(Q * f)
```

with amplitude varying as a function of frequency f will build up a value with any desired spectrum.

The *turbulence* function described by Ken Perlin is essentially a stochastic function with a “fractal” power spectrum, that is, a power spectrum in which amplitude is proportional to $1/f$.

```
value += abs(snoise(f*s))/f;
```

Derivative discontinuities are added to the turbulence function by using the absolute value of the `snoise` function. Taking the absolute value folds the function at each zero crossing, making the function undifferentiable at these points. The number of peaks in the function is doubled since the troughs become peaks, and the overall frequency is therefore doubled as well.

As an example, we’ll look at the world’s simplest marble shader, which uses a “spectral function” synthesized from four noise components with a $1/f$ power spectrum. This “fractal noise” is reminiscent of *turbulence*, but it doesn’t have the absolute value feature.

A simple marble shader

```
float texturescale = 1;
point PP;
float i, f, marble;
#define NNOISE 4
marble = 0;
f = 1;          /* starting frequency */
for (i = 0; i < NNOISE; i+= 1) {
    marble += noise(PP * f) * 1/f;
    f *= 2;
}
marble = clamp(4*marble - 3, 0, 1);
CT = marble_color(marble);
```

CT is set to a color `marble_color` which is a color spline, or other straightforward one-dimensional function of `marble`, the value of a stochastic function with a “fractal” power spectrum.

Perturbed Regular Patterns

Perturbed Regular Patterns

- **begin with a regular pattern**
- **add noise to calculation to perturb pattern**
- **makes pattern irregular, more interesting or “natural”**



The most useful type of pattern is the “perturbed regular pattern,” which combines a basic regular structure with elements of irregularity to keep it interesting. Most natural materials are somewhat irregular and non-uniform. Even man-made materials are irregular due to poor quality control, shipping damage, and weathering.

Example: Improved Brick Shader

```
ss = scoord / BMWIDTH;
tt = tcoord / BMHEIGHT;
if (mod(tt*0.5) > 0.5)
    ss += 0.5;          /* shift alternate rows */
tbrick = floor(tt);    /* which brick? */
ss += 0.2 * (noise(tbrick+0.5) - 0.5);
sbrick = floor(ss);    /* which brick? */
ss -= sbrick;
tt -= tbrick;
w = step(MWF,ss) - step(1-MWF,ss);
h = step(MHF,tt) - step(1-MHF,tt);
Ct = mix(Cmortar, Cbrick, min(w,h));
```



In the brick shader, we can add a line of code which perturbs the horizontal location of each row of bricks a little to suggest that the bricklayer was human and therefore error-prone.

Random Placement Patterns

- **regular features or subpatterns**
- **dropped at random positions, orientations in texture space**
- **sometimes called “bombing”**



“Bombing” or placing subpatterns at random positions and orientations in the texture space is a useful trick. In the following wallpaper example, we break up the texture space into a grid of cells. Each cell has a star in it, and the location of the center of the star varies depending on noise. To make the texture more irregular, we use a noise value to decide whether to place a star in a given cell or leave it empty.

Example: Wallpaper Shader

```
#define CELLSIZE (1/NCELLS)
ss = s * NCELLS; tt = t * NCELLS;
scell = floor(ss); tcell = floor(tt);
sctr = CELLSIZE * (scell + 0.5 +
    0.6 * snoise(scell+0.5, tcell+0.5));
tctr = CELLSIZE * (tcell + 0.5 +
    0.6 * snoise(scell+3.5, tcell+8.5));
ss = ss * CELLSIZE - sctr;
tt = tt * CELLSIZE - tctr;
```

now just use the star example, omitting the first line.



Texture files can be made more interesting by accessing them using perturbed coordinates. This introduces additional irregularity in the texture and can hide some of the artifacts that result when the same texture file is repeated many times across a surface.

Perturbed Access to Texture Files

- **access texture file adding noise to calculation of texture coordinates**
- **humorous distortions**
- **adds variety to natural textures such as wood**
- **hides obvious repetition in repeated texture tile**



Example: Perturbed Access

Access texture file with “noisy” (s,t) coordinates

```
Psh = transform("shader", P);
ss = s + 0.2 * noise(Psh) - 0.1;
tt = t + 0.2 * noise(Psh+(1.5,6.7,3.4)) - 0.1;
Ct = texture("name.tx", ss, tt);
```



Light Sources

Our discussion up to this point has really focused on shading surfaces, and it is true that the majority of the shaders that most individuals write will in fact be surface shaders. However, occasionally it will be necessary (or interesting) to program the light sources as well. This is done with light source

shaders. Light source shaders calculate the amount of light that leaves a light source and arrives at the surface of an object. The graphics state provides the information about which point is of interest, etc. and the shader then calculates the color of the light ray C_l . This state information is very similar to that provided to surface shaders, but subtly different, so be careful.

Light Source Shaders

- **Calculate amount of light leaving light source and arriving at surface**
- **Input: graphics state and L**
- **Output: C_l**

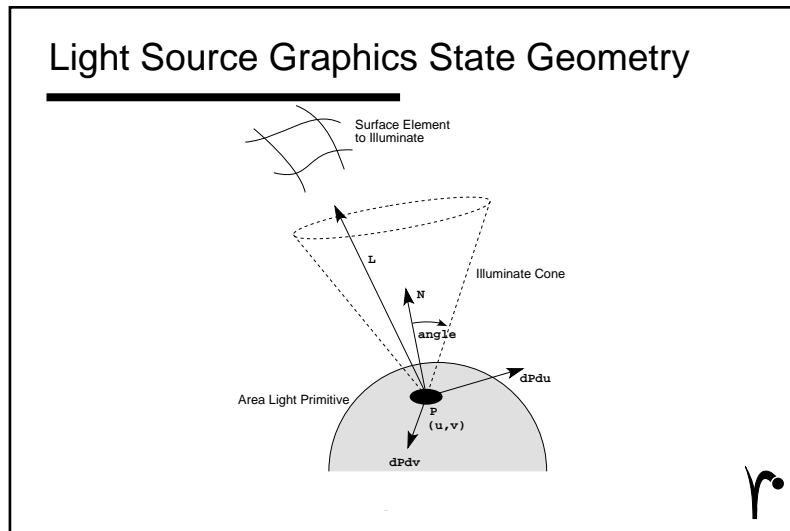


One important thing to note is that the vector L points out of the light toward the surface when programming a light shader. When programming a surface shader, L points out of the surface toward the light.

Light Source Graphics State Variables

- **L Outgoing light ray direction**
- **P Light position**
- **P_s Illuminated surface position**
- **E Position of eye**





Simple lights, ones that cast the same amount of light everywhere, or ones that cast light in a very straightforward pattern, are easy. And boring. Instead, you might want to change the characteristics of the light in a much more complex, theatrical ways. Lights that fall in patterns, lights that have color variation, lights that have texture. There are nearly as many ways to do this as there are to write surface shaders, but here are a couple of examples that we've used.

This is a very simple shader that produces a nice fiery effect. It is programmed to flicker in an interesting way as time passes. Simple, but effective.

firelight.sl

```
light
  firelight(
    float intensity = 1;
    color lightcolor = 1;
    point from = point "shader" (0,0,0);
    float time = 0;
    float freq = 1; )
{
  uniform float flicker = noise(freq*time +
                                0.25);

  illuminate( from )
    Cl = intensity * lightcolor / L.L;
}
```

The noise function has a 0.25 offset because noise with an integer parameter always returns 0.5. You can imagine lots of other ways to generate fiery or ripply effects. The classic stage fire cylinder, for example, could just be done with a rotating texture map, or noise() “streamers” moving up. The next example is more complex.

```
light
mirrorball(
    float intensity = 1;
    color lightcolor = 1;
    point from = point "shader" (0,0,0);
    point up = point "shader" (0,0,1);
    point front = point "shader" (0,-1,0);
    float numspots = 36;
    float spotsize = .25;)
{
    float sp, tp, ns;
    point upvec = normalize(up - from);
    point frontvec = normalize(front - from);
    point pp, stcenter;
    float spotrad = numspots/36*spotsize;
    float dfc, atten, lr;
    float jittamp = 1;

    illuminate( from ) {
        /* get spherical coordinates sp and tp */
        tp = acos(normalize(L) . upvec)*numspots/PI;
        ns = floor(sin(floor(tp)/numspots*PI)*numspots+0.5);
        pp = normalize(L - upvec*(upvec . L));
        lr = (pp ^ frontvec) . upvec;
        if (lr <= 0) lr = -1; else lr = 1;
        sp = lr * ns * acos(pp . frontvec)/PI;

        /* jitter spot centers */
        stcenter = .5+jittamp *
            (point
            noise(floor(sp)+234.5,floor(tp)+501.5)-.5);

        /* attenuate by distance from spot center */
        tp = mod(tp, 1)-xcomp(stcenter);
        sp = mod(sp, 1)-ycomp(stcenter);
        dfc = sqrt(sp * sp + tp * tp);
        atten = 1-smoothstep(0, spotrad, dfc);

        Cl = intensity * lightcolor * atten;
    }
}
```


The first section of code inside the `illuminate` statement calculates spherical coordinates based on the three direction vectors determined by the points passed in to the shader. The next section uses `noise()` to bomb light circles onto a grid established on these spherical coordinates. The light is animated by changing the directional vectors.

Shadows

Some rendering algorithms, such as ray-tracing and radiosity, attempt to calculate global illumination effects, and therefore provide shadows (almost) automatically. However, RenderMan renderers that do not simulate global illumination can still produce shadows, often with less cost and more flexibility than global illumination renderers.

The primary mechanism for rendering shadows without global illumination is the *shadow map*. A shadow map is a texture map which contains depth values from the point of view of the light source. The value stored in each pixel of the map is the depth of the closest object to the light in that pixel. That object shadows any object which is behind it (from the point of view of the light). The shadow map lookup is based on a position in the scene. That point is transformed into the space in which the map was rendered, and the distance from the light is compared to the value stored in the map. Points that are closer to the light than the “shadow casting object” are considered to be in light. Objects that are farther from the light are shadowed.

Shadow Maps

- **Depth value stored in texture pixels**
- **Z-buffer computed from light source**
- **Camera at point in scene**
- **Value in map is closest surface**

Making a Shadow Map

- **Render a depth file from light position**
- **RiMakeShadow() turns depth image into shadow map**
- **Access from light shader with shadow() function**



One thing to consider is that if you plan to use the shadow map in a light source at infinity (such as distantlight), you should use an orthographic projection to render the depth file. If you plan to use the map in a light source such as a spotlight, use a perspective projection.

This is how you would call the shadow map in a distantlight.

shadowdistant.sl

```
solar( to - from, 0.0 ) {
    Cl = intensity * lightcolor;
    if (shadowname != "") {
        Cl *= 1 - shadow(shadowname,Ps);
    }
}
```



In order to transform points into the lookup points into the light's view space, the shadow map actually contains the viewing matrices of the shadow's camera. In this way, it is not necessary for the RIB/shader programmer to build a special user-defined coordinate system around each light source. The matrices automatically handle the field-of-view and screenwindow considerations of the light's

camera. This leads to several interesting tricks. Since the map “knows” how to do the projection, the light vector L doesn’t really need to be aligned with the shadow, so you can use a single shadow map to shadow several lights, or slide the light around to move the specular highlights without affecting the positions of the shadows. In the next section, we will examine casting shadows without using a shadow map, and if you do, you need to do this projection on your own.

Camera Transformation

- **Transformation to image camera position stored in depth image**
- **Used in shadow() function**
- **Flexibility**
 - other lights can use map
 - camera does not have to be at light position
 - can even call shadow() from surface shader



Fake Shadows

A shadow map is accessed at every shaded point inside the illuminate cone of the appropriate light source, and this slows down rendering somewhat even in areas which may not need any shadows. At the cost of some realism, you may be able to speed up shadow rendering in some circumstances using procedural intensity manipulation, or other types of texture maps, instead of a shadow map. In some situations, that fudging of realism is exactly what you want, so even better!

Procedural Shadows

- **Similar to texture generation in surface shaders**
- **Don't need texture or shadow maps**
- **Much faster**



Here are some examples. The first shader just generates a square lighted region, simulating barndoor stage lights.

```
light barndoor ( float left = -1; right = 1;
                 float bottom = -1, top = 1;
                 float edge = 0.1;
                 float intensity = 1; color lightcolor = 1;)
{
    float tau, x, y;
    point P2, L2;
    P2 = transform("shader", P);
    axis = vtransform("shader", "current", point(0,0,1));
    illuminate(P, axis, PI/2) {
        L2 = transform("shader", Ps) - P2;
        tau = (1 - zcomp(P2)) / zcomp(L2);
        x = xcomp(P2) + tau * xcomp(L2);
        y = ycomp(P2) + tau * ycomp(L2);

        C1 = intensity * lightcolor *
            smoothstep(left, left+edge, x) *
            (1 - smoothstep(right-edge, right, x)) *
            smoothstep(bottom, bottom+edge, y) *
            (1 - smoothstep(top-edge, top, y));
    }
}
```

The first lines inside the `illuminate` statement calculate the intersection of the illumination ray with a plane in front of the light source (just like projecting the scene onto a camera's film plane). This trick depends on the light source being aimed like a camera, that is, set into the scene with a

transformation matrix so that the light is at the origin of its local shader coordinate system, beaming light out over the z-axis. This is a very handy trick, and will be used in several of the remaining shaders in this section. I refer to it as the “slide-projector” projection. The long final statement evaluates the light intensity to full on when the point is within the box, full off if it is outside the box, and a soft penumbra when near the edge.

You can get the effect of a light through a paned window by just adding crossbars to the same shader.

window.sl

```

Cl = smoothstep(left, left+edge, x) *
    (1 - smoothstep(right-edge, right, x)) *
    smoothstep(bottom, bottom+edge, y) *
    (1 - smoothstep(top-edge, top, y));
Cl *= intensity * lightcolor *
    (1 -
        smoothstep(-edge, 0, x) *
        (1 - smoothstep(0, edge, x))
    ) * (1 -
        smoothstep(-edge, 0, y) *
        (1 - smoothstep(0, edge, y))
    );

```



Here the first line does the box evaluation, as in the previous shader. The second line superimposes crossbars on the result of the first calculation. If you want venetian blinds instead, just add a bunch of horizontal bars to the barn doors.

Procedural techniques such as these can often be used for special-purpose lights, when you know exactly what the geometry will look like. For example, light shining out of a hurricane lamp, or through a sconce.

The next obvious trick is to cast a shadow which is held in a texture map. This is exactly analogous to a slide projector, which casts the image of a texture (the slide) into the scene. Such a shader is presented in the *RenderMan Companion*. In this case, we can put the shadow of objects into the slide, and cast those shadows. The shadows don’t even need to be of real objects from the scene. They can be fake objects, phantom objects, or objects which are “off screen” and are never seen on-camera.

Shadows of fake objects

- **Might want shadows of objects that don't exist in scene**
- **Can't or don't want to generate a shadow so**
- **Paint or scan it!**



When you have a shadow image, modify the light source shader to access it using the camera image coordinates, similar to the barndoor projection above.

distprojlight.sl

```
illuminate(P, axis, PI/2) {  
    /* calculate x and y as before... */  
  
    shade = 1 - texture(texname, x, y);  
    Cl = shade * intensity * lightcolor;  
}
```



This will project the shadow onto your scene like a slide projector. If you use a separate painted shadow for each frame of an animation, you will get animated shadows.

This technique is very effective in producing various effects, from menacing shadowy monsters to the canopy of a forest.

Those techniques all are derived from manipulating the intensity of the light that is cast from a light source into the scene. Of course, you don't always need to do it that way. Now, imagine that you have a bunch of characters on a simple floor plane. You can splat shadows of the characters directly onto the floor using just an alpha image in the surface shader of the floor. We call these “drop shadows”.

Making Drop Shadows

- **Render image (without floor) from light**
- **Make single-channel texture map**
- **Project “P” into light’s “NDC”**
transform to light position and orientation
scale image range to [0–1] horizontal and vertical
- **Call texture map**



floor.sl

```
surface floor( float Ka=1, Kd=1 )
{
    point Nf;
    float x,y,shade;
    Nf = faceforward(normalize(N),I);
    /*... project as before... */
    shade = 1 - texture("shadow.tex",x,y);
    Oi = Os;
    Ci = Os * Cs * ( Ka*ambient()
        + shade*Kd*diffuse(Nf) ) ;
}
```



Aliasing in Shaders

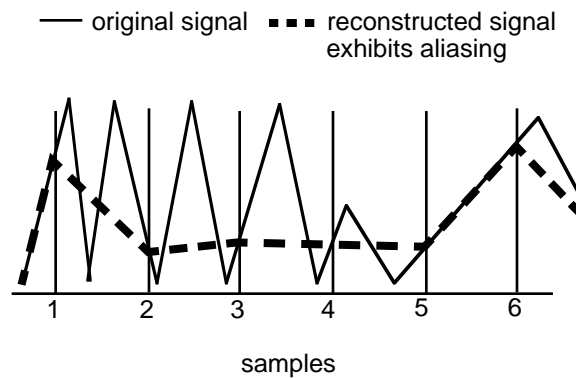
As if writing realistic shaders isn't challenging enough, we have to deal with aliasing artifacts as well. It would be nice if the RenderMan renderer could solve shader aliasing problems as well as it solves geometric aliasing (jaggy edges on geometry). As we will see, however, this isn't the case, and we have to do quite a lot of work to suppress aliasing artifacts in our shaders.

What is aliasing?

- **classic examples**
- **jaggy edge or line**
 - checkerboard perspective
 - signal frequencies too high for sampling rate
- **high freq energy reconstructed as low freq artifact**



Aliasing



Aliasing occurs when a regular sampling process is used to sample and reconstruct a signal whose highest frequencies are greater than one-half of the sampling rate. The high frequency energy shows up as a low-frequency “alias” in the reconstructed signal.

How to Antialias

- **low-pass filtering of signal before sampling**
- **stochastic sampling**
 - high freq show up as noise
 - LOA (less objectionable artifact)



We can prevent aliasing in two ways. The best way is to filter out the high frequencies before sampling. Since the high frequencies can't be reproduced properly, we're better off without them. An alternative is to use stochastic sampling of the signal. The high frequency energy is still present in this method, but it appears as noise in the reconstructed signal, and noise is generally less objectionable to the human eye than are low frequency aliases.

Why Doesn't Renderer Antialias?

- **can't solve shader aliasing after shading**
- **ray tracer could stochastically sample shader, but...**
- **prefiltering is a cleaner solution**
- **automatic prefiltering of texture file is easy**
- **automatic prefiltering of procedural texture is hard**



A RenderMan ray tracer using stochastic sampling is in a better position to apply this method of shader antialiasing. But even if we had such a renderer, the best image quality would still be obtained

by prefiltering the shading functions to eliminate both aliases and the noise generated by stochastic sampling of high frequency signals.

So how can we write shaders that prefilter the shading function to reduce objectionable high frequencies?

Writing Antialiased Shaders

- **limit frequency content of shader**
 - clamping
 - filtering
- **filter width from sampling rate**
- **simplest filter is box**



There are two cases: functions that are synthesized as a sum of components of various frequencies (spectral functions), and other functions. The “clamping” technique applies to the spectral functions. For other functions, we have to apply analysis and ingenuity to figure out how to build filtering into the shader.

Clamping a Spectral Function

- **function consists of sum of components**
- **attenuate all components with frequency near or above one-half of sampling rate**
- **sudden amplitude change will pop or alias**



The term *clamping* comes from the SIGGRAPH '82 paper "Clamping: A Method of Antialiasing Textured Surfaces by Bandwidth Limiting in Object Space," by Alan Norton, Alyn Rockwood, and Philip Skolmoski.

Clamped Turbulence Function

sudden clamping

```
value = 0
cutoff = clamp(0.5 / swidth, 0, MAXFREQ);
for (f=MINFREQ; f < cutoff; f *= 2)
    value += abx(noise(f*s))/f;
```

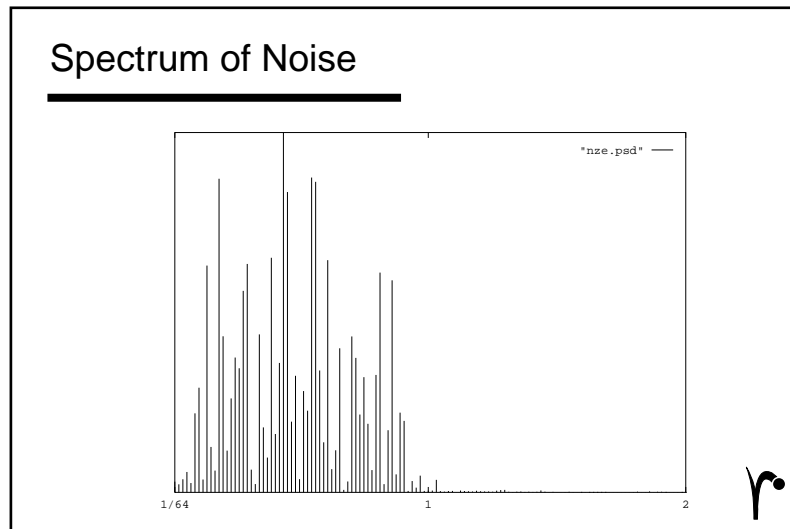
gradual clamping

```
value = 0
cutoff = clamp(0.5 / swidth, 0, MAXFREQ);
for (f=MINFREQ; f < cutoff/2; f *= 2)
    value += abx(noise(f*s))/f;
value += 2*(cutoff-f)/cutoff *
    abs (snoise(f*s))/f;
```



The "sudden clamping" version of the *turbulence* function comes from the SIGGRAPH '85 paper "An Image Synthesizer," by Ken Perlin. Quite often, however, sudden clamping is very visible in animations as a sudden "pop", as an entire octave of high frequency suddenly (dis)appears as the object passes some filterwidth threshold. We can ameliorate (but not eliminate) this problem by fading in the highest frequency so that it doesn't "pop". Usually. This is called "gradual clamping".

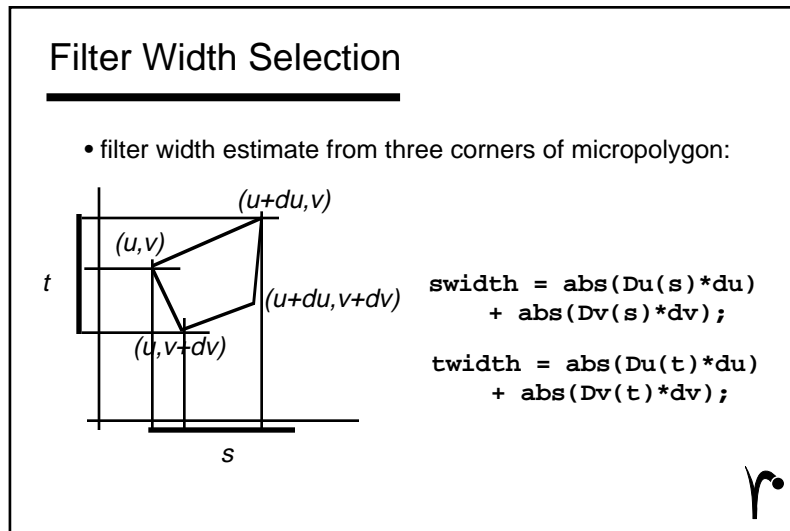
One function that we very often wish to clamp is noise. In order to see if we can do a relatively good job of clamping noise's frequency content, we need to know just how well band-limited an individual call to noise really is. After all, there's no point in trying to clamp to "only the right frequencies" of calls to any function if even a single call gives us frequencies beyond our ability to handle.



We can see here that, although there is some leakage, noise is reasonably well band-limited. Its spectrum is a bit spiky, and it is definitely *not* white within its pass-band. It is clear, though, that it does not really have a predominant frequency.

You'll notice that the clamping equations above refer to a parameter `swidth` which is the sample width in the appropriate space and direction. For example, it might be the size of the shading sample in raster space, or in texture space. In fact, choosing the appropriate sample size is one of the most important aspects of doing filtering correctly. In PhotoRealistic RenderMan, a "sample" is a micro-polygon: a tiny rectangular portion of the object which is roughly one pixel in size (controlled by `RiShadingRate`), but definitely not aligned with the pixel grid. In ray tracers such as BMRT, the sample size is in some sense the projection of the (sub)pixel onto the object, and therefore is pixel aligned but spans a funny (usually) trapezoidal area on the object itself.

In either case, filtering is done as some form of area sampling, and the projection of the sample onto the appropriate space always determines the size of the region for which we wish to find the "average value". Quite often, for simplicity, the actual projected trapezoid is approximated with an axis-aligned rectangle in feature space. This can be done by using the derivatives as follows.



This trick for determining filter width works pretty well most of the time. Note that the filter will fit the micropolygon exactly if $(s, t) = (u, v)$.

An alternative filter width estimate is to use the cross product of the two edges of the micropolygon to compute the area of the micropolygon. This estimate will be wrong if the micropolygon is not a parallelogram, and it gives the same filter width (square root of the area) for the s and t directions, so the two directions must be filtered equally.

Animation Considerations

Finally, I want to discuss a few points about antialiasing shaders used in animations. There are two important considerations that must be taken into account for animation shaders above and beyond those written for still images.

Animated Shader Considerations

- **Temporal Antialiasing**
- **Resolution Independence**
- **White-noise patterns**
- **Choice of coordinate system**



First, there is the issue of temporal antialiasing itself. RenderMan renderers handle motion blur of objects both in position and in shape. Future RenderMan renderers may actually be able to motion blur shading parameters, but none of them do at the current time. PhotoRealistic RenderMan shades all objects at shutter open and then “smears” the colors across the screen. This means that some unfortunate effects can occur, such as specular highlights moving on a spinning top, or objects dragging light into a shadowed region. BMRT casts rays at the appropriate time, so those effects don’t occur, but it still doesn’t interpolate shader parameters. Why is this important? In PRMan, shadow textures, displacements and filter size calculations are all based on shutter-open geometry, and sometimes shaders need to know this in order to operate correctly on fast-moving objects. Similarly, if a shader’s texture changes rapidly as a function of time (a flickering light, for example), the shader may need to antialias itself over the time that the shutter is open to get the desired effect.

Another problem shaders often have in animation is that the object on which they are applied will significantly change scale, either between shots, or often during a single shot. This change in scale will very often severely tax your carefully planned clamped noise synthesis code. Beware of those huge scale changes which cause “pops” that defy even gradual clamping. You may be forced to calculate a more accurate analytical antialiasing function of the frequency space (which is far beyond the level of this talk. See the 1992 course notes for pointers).

If the object must behave correctly whether the object is 5 pixels wide or fills the whole screen, it quite often will be written with wholly different sections of code responsible for antialiasing at various scales. In geometric modelling, this is known as *level of detail*, and the concept is equally valid in shader writing. A shader may be gray at a distance, have a bump texture when a little closer, have a displacement texture when even closer, and have a full-fledged fractal procedural displacement when examined in the microscope.

It is not unheard of for shaders which have very high frequency content (such as dirt, sand, etc.) to suddenly go totally wacko under animation. The obvious culprits of shading rate and pixel filtering will be tried to no avail. It looks great in a still, but “rages” like crazy when it moves. Careful investigation will discover that the shader is in fact aliasing like crazy in the still, too, but since the in-

tended effect is a very high frequency content, the point-sampling of the pixels just clips a few (or a few dozen) octaves of noise and the “appearance” is the same. Until it starts moving, and the point-sampling clips differently in each frame. Ouch!

Finally, and this has been mentioned before, when you are writing shaders for animation, be extra careful to pick the right coordinate system for doing your calculations. Calculations which look great in the default *current* space in a still frame will suddenly go nuts when the camera move starts (PRMan’s current space is actually camera). Or calculations in *object* space will go awry when the object starts to deform, and the surface starts undulating through its 3D texture.

Summary

Oh, there are a lot more tricks you are going to have to learn before you can start making really good use of the Shading Language in your animation production pipeline. Tricks like building libraries of example shaders as a base, and the elegant choice between complex shaders on simple geometry and simple shaders on complex geometry, and how to write shaders that are debuggable without having access to the renderer source code. However, these are tricks that only time will teach. It isn’t impossible, simply hard work. We’ve all been through it, and are still going through it, and we’re getting pretty good results along the way, so you can, too. Good luck! We knew the job was dangerous when we took it. It’s a ugly job, but someone has to do it. And if you find any magic bullets, let me know. I’m still looking.

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

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



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 language 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);
```



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);
```

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 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);
}

```

Television Commercial Production at Pixar

Oren Jacob
Pixar

Abstract

This talk will focus on the animation process at Pixar. Specifically, the production of the Listerine *Arrows* commercial, which won the Gold Clio in 1994 for Best Computer Animation, will be considered step by step as a case study in the work flow through Pixar's commercial production group. Copious use of historical anecdotes will give the reader the opportunity to see how badly we screw up during the course of one of our commercials. Attention will be given to the various issues surrounding RenderMan as it applies to the production process as well as how RenderMan's use affected the final look and feel of the spot.

Background

Commercials that come to us here at Pixar usually fall into two categories. First, there are the fully developed story boards that we are instructed to execute exactly as drawn. These are the less interesting of the two kinds of jobs and we only take this kind of work if the look and feel of the spot matches closely with that old Pixar Look that you've all grown to know and love. The second, and usually more interesting kind of work, are the general concept boards that come in. We are usually allowed significant latitude to redo the boards and execute them in a fashion that is interesting to us. What is interesting to Pixar? Usually a strong story component is added to the idea with particular attention to the use of CG character animation, as opposed to a live action component. While we do do live-action CG combo work, it does not comprise a majority of our efforts and we usually leave the hard stuff to our esteemed colleagues over at ILM across the bridge.

Pixar has established a strong tradition of quality work in the realm of completely computer generated character animation and the Listerine *Arrows* commercial follows in this vein. The original concept for Listerine as Robin Hood was presented by John Lasseter and Andrew Stanton to J. Walter Thompson, NY in our original bid for the Listerine campaign several years ago. JWT came back to us in the Fall of 1993 and requested that we produce the last two spots in that long running campaign, "Robin Hood" and "Patton", for release in early 1994.

To create a commercial at Pixar, there are three people who come together to lead the effort. There is the *Producer*, in charge of maintaining good client relations, bidding for jobs, wrangling the crew, and, in general, being a caring but authoritative parental figure. The producer spends most of the time on the telephone dealing with clients and managing all of us, which is no small task I might add. There is the *Animation Director*, in charge of the story boards, overall character design, animation, and most aesthetic decisions in the production. The animation director spends most of their time worrying about the final look of the spot and, in general, overseeing the animation as it progresses through the production pipeline. And there is the *Technical Director* who takes care of the modeling, shading, lighting, and rendering. The technical director spends most of their time typing really fast because they are behind schedule. In the case of Listerine *Arrows*, the Producer was Darla Anderson, the Animation Director was Jan Pinkava, and I, Oren Jacob, was the Technical Director.

Listerine *Arrows* Play-By-Play History

At this point in the talk it is appropriate for me to begin to follow, chronologically, the progress of *Arrows* through the production pipeline. In a live talk, videotape would be used at each of the points where we snapshot the production, to better understand what was going on and what the footage looked like. Unfortunately, we can't show videotape in printed notes, so you'll have to use your imagination.

9/27/93

Starting at the beginning of the production process, this is the animatic as it was presented to JWT NY. There are several things to note about this. First, there is the always important issue of music. The Cool Mint Listerine *Jungle* commercial that Karen Robert, Andrew Stanton, Eliot Smyrl and Tom Porter did earlier had that great tune by Baltimora, "Jungle Boy". In our initial effort to go a little more retro, we found the Herbie Hancock song "Rockit" from the early 1980s. Unfortunately, even though the record scratching has a remarkable resemblance to arrows flying through the air, the agency did not like our suggestion and the original soundtrack of Baltimora was brought back and used with a few slight modifications.

But I digress. The video we produced at this stage is referred to as an animatic. It is a series of hand drawn images, executed by Jan, that represent the various shots in the commercial and how they are framed. The drawings are shot to videotape and then edited together to form a working print of the final spot. This, along with our ill-fated soundtrack suggestion, were presented to JWT for approval.

Having received the OK, it was then up to Jan and I to analyze the various shots and determine what models were in view, how they moved, what angles you could see them from, and how close they were to camera. It became obvious from the beginning that there were several significant technical challenges in this commercial. First, we had to construct an articulated, gloved hand and move that along with the model. And, second, we had to build the forest.

10/4/93

Just to spend a few minutes on how Pixar actually does animation, it is worthy to note that do not use commercially available software packages like Alias, Wavefront or SoftImage to give our characters motion. Instead, we have a procedural language, full of clever and interesting functions designed to deform meshes in 3-space, which is basically what animating surfaces is all about. This language, called mdl for obvious reasons, has been developed by Bill Reeves, Eben Ostby, Darwyn Peachey, Rich Sayre, Ronen Barzel and numerous others since our days at LucasFilm. At this point in time, it contains a very rich set of geometric primitives and operations to manipulate those primitives. In particular, the language itself is vectorized so that array operations can, if one is careful, be coded up as simply as scalar operations making things like matrix dot products and component additions essentially trivial. It is also important to realize that since one is working within the construct of a programming language, as opposed to a spiffy looking GUI application, that one has complete freedom to construct very clever and complicated procedural deformation functions. Things like bends and twists have been incorporated into the language as hard-coded functions but deformations like creasing and crunching can be built by hand as well.

In light of all that, the construction of the hand posed several interesting problems. First, there was the complexity of the shape. As anyone who has tried to model the human form (or even parts of it) with a connected series of cubic splines can tell you, it is hard. In particular, areas under the armpits, between the legs, and between the fingers are often chosen as places to stick your dangling points

that you just can't figure out how to match across mesh boundary seams. In our case, since we were constructing a hand which not only had to move in a complicated way but also had to be seen very close in frame, we could not afford to just dump stray points in the finger webbing and hope that nobody noticed.

The way we finally decided to execute this model was to send Jan home one weekend in October with a very large quantity of styrofoam and plaster. Jan was then instructed to mess up his house as much as possible, which he did, and he brought us back this sculpture. I then went to work blow-drying the plaster (which hadn't hardened because it is foggy up in the Berkeley hills) and after getting help from several interns holding shrink-wrap-heat-guns to it for a number of hours, it was dry and hard enough to scribe mesh lines on. Several different patterns were attempted, but it was hard to meet the design requirements of one-to-one CV-to-CV mesh continuity across the boundary seams, as well as maintain a sensible number of individual meshes. Since all curved surfaces at Pixar are cubic, never polygonal, just inputting the points themselves was insufficient. Instead, we had to input the profile curves. This was done with a Polyhemous 3 Space Digitizer and once all the profile curves were input, the meshes were constructed within Alias, which has a very good family of tools for mesh construction and alignment.

There are several important features of the hand's final mesh construction that should be noted. First, the majority of the bending occurs either in areas where there are no seams, or in areas where the curve that crosses the seam is all the same distance from the center of the circular bend. For example, the points that make up the top of the middle finger are nearly coplanar so that when the middle finger bends down, those points will not bend apart and tear the finger open. It is also important to note that the areas in which multiple meshes come in contact with one another are regions of the hand where bending is minimized. For example, the mesh that constitutes all of the fingers meets the palm of the hand below the last finger knuckle so that none of the finger bending occurs across that boundary. This emphasis on not bending the model at mesh seams made maintaining surface continuity much easier than having to evaluate the various meshes on a frame by frame basis to reassemble the hand back into a smooth shape.

10/18/93

Having dealt with that problem to our satisfaction, there was the small issue of the forest. A forest is a rather large and complicated thing. How does one build a forest? Where do you start? Well, after examining the animatic, it was clear that there were several shots in which the Listerine bottle was sitting up on a branch. So, we decided to start from there - build a branch.

Our first attempt involved constructing an entire tree in Alias. It was easy to build tubes there and also easy to build extrusions, but I really wanted to use the "birail surface" command on a closed ring to build a branch, taking advantage of the birail's capability to reduce the profile's radius along the length of the branch. At that time, Alias couldn't do that. So, we first built the top half of the branch and then the bottom half and then attached the surfaces together. We got some pretty good results out of that attempt, but once we tried to stick these branches inside of our animation system, Menv, we realized that we wanted to tweak the profiles just a little and move around a few points. Given the fact that we had to run a converter on the Alias output files to get our system to understand them, it was not easy to just go back and forth between the two worlds of Alias and Menv in a timely enough fashion to complete the design. So, we opted for a different approach. We decided to build a function in mdl that would take a profile curve and an extrusion path and construct a tube of reducing radius along the extrusion by replicating copies of the profile at various radii and orientations. This worked very well and allowed Jan to fiddle with each of the branches just a little bit to get the

look that he wanted. It also allowed us to install animation controls on the branches to change their layout and replicate them. This is a way to get lots of trees in a short amount of time. In fact, there were only 8 trees built for the entire spot, 3 of which are special purpose sets. So we constructed a forest out of just 5 canonical trees.

11/23/93

Having built the first trees, we cribbed the bottle from a previous Listerine commercial and started doing layout. The in-progress version of the commercial from November 23, just before Thanksgiving, showed the result of finally taking Jan's drawing from the animatic and trying to assemble a three dimensional set that corresponded to those drawings. The camera that was going to render the footage was installed and a few simple camera moves were put in to give us a sense of what each of the shots was going to do. There was no real shading going on here. All surfaces are colored flat. The only real reasons to do a take like this are: first, to push everything through the rendering pipeline once to verify that it does in fact work; and second, to see how the 2D drawings map into 3D. This is a very important step in realizing storyboards. Just because a talented artist can draw a picture on a flat piece of paper does not in any way map to a TD's ability to construct that set in 3D. Everything always looks a little bit different when you finally render it, and it is at this stage where aesthetic decisions on the layout of the forest and the characters are made. It is also here where, sometimes, entire shots get cut. We cut the 'Men In Tights' spoof shot because there wasn't enough time to keep that shot and still get everything else that we wanted in the commercial into the 30 seconds that we were given.

12/15/93

By this time in mid-December we are finally computing the commercial with plastic shading on everything and are beginning to add in more detail. One of the important features of the test video at this stage is the leaf canopy. We had spent a lot of time considering what would be the best way to build an entire canopy of leaves over a forest. And, at this point, without any interesting lighting, shading, or shadowing, the approach of just sticking a whole ton of leaves up there yielded some very promising color variation. In fact, a very nice variety of different shades of green resulted simply from the fact that the various leaves surface normal's point in different directions, and the one solar light source in the scene brought out all that range in intensity. This was encouraging because we knew that once we start putting real shaders on the leaves and turn on shadows then we would begin to get some very good looking footage.

12/29/93

At this point we made one of the first attempts at real lighting on a shot, shot 2. Because shot 2 had such a deep view into the forest, we used this as our testbed for the color values and intensities that we were searching for in the rest of the commercial. I think that, for a first attempt at putting shadows on the leaf canopy, it came out pretty good. We then decided that we were going to go with this tree and leaf construction technique and just spend the rest of our time optimizing the lighting conditions and building real shaders for the forest.

It is also in this take that we suffered the first failure of what we called the "caustic effect". This is a blue light that shines through the bottle onto the poster in shot 5, making you think that the bottle is standing just off frame left. We intended for the blue light to shimmer, as it might if you held a clear glass of blue liquid up in the sunlight and looked at its cast shadow on a table or something. This didn't work well the first time. As it turned out, the caustic effect, the leaves, and the bark on

the trees were probably the three thorns in our sides throughout the entire production. It is very hard to replicate natural phenomena like these and Mitch Prater did an admirable job in finally executing these. We now realize the importance of deciding clearly early on how complex shaders like these should look. Sorry about all that trouble Mitch :-).

1/5/94

More tests, more failures. Timothy Leary would've been proud of some of them. It should be noted that forgetting a minus sign is a bad thing. RenderMan doesn't find forgotten minus signs well. An important lesson for those that want to head down this path in the future.

1/11/94

In this take, we at Pixar Shorts learned an important lesson in communication. Mitch, our hero from the previous discussion of leaves and caustic effects, decided that he didn't have enough work to do yet (!) and wanted to make each of the 250,000 leaves in the canopy flap in the wind. Mind you, by this time, I (theoretically acting as the lead technical director on this job which had grown orders of magnitude more complicated than any of us had expected) hadn't had more than 4 hours sleep for the past 6 weeks. And then, Mitch comes skipping into my office and tells me that he wants to make the entire forest move in the wind. I was skeptical to say the least. Mitch tested a version of his shader that just perturbed the normals on the leaves without actually moving them, but he apparently perturbed them a little too much. What we got out was a force 10 gale tearing through our little forest glen.

The frightening thing about this story is that a videotape left Pixar looking like this, without anyone knowing that it was only supposed to be an internal test. Doh! As it turns out, the agency in New York loved Mitch's idea and this really saved the whole production because Darla wheedled out another week for us to final this effect - which actually meant we could finish the commercial. Until we got that one week extension, we were in danger of not meeting our deadline, which is a very bad thing to do in a business where agencies purchase air time before contacting the production houses that are slated to do the work.

1/16/94

We are now nearing the emotional low point of production. The more work we did, the more problems we had. For example, the bottle went nuclear on us in a few of the shots. The bottle's particular bright blue is very NTSC-illegal and we're still not sure why this happened. Various theories on additive light from reflection maps, multiplies instead of additions in the shader, and just incorrectly typed RGB triples abound, but nobody has been able to prove any of them yet. The hand is also broken pretty terribly, and this deserves a little special attention.

The leather shader on the glove was based on a three dimensional call to noise(). The parameter passed into the noise() function is P, the surface location being sampled. This works wonderfully for static geometry, but as I mentioned earlier, our mdl language allows us to deform geometry in 3 space. That is exactly how we are animating the hand, by bending points in object space. So, when we pass in our new, bent P, the call to noise() returned a different value because P has changed. This means that we need some stationary space which is continuous across all the meshes of the hand so that we can call noise() and pass in points in that stationary space. Since it is essentially impossible to invert a nonlinear deformation like a circular bend in a shader, we used a technique affectionately called "make sticky".

I believe that this was first described in the literature by ILMers in Cinefex, but I'm not really sure. In any event, it a process whereby the points of a mesh not only have their geometric location after deformation stored in the rib file, but also their original, unbent location as well is stored as a vertex variable. This is very easy for bspline, bezier, and linear patches, but RenderMan has a limitation that on a nurbs patchmesh, you can only assign vertex variables to the 4 extremal points in the patch-mesh. This is clearly too coarse a sampling, so what has to be done is that every small subpatch in the nurb mesh has be broken out and instanced as its own little nurb mesh, with vertex variables assigned to contain the original, unbent location of the points. Having done that, in the shader we now have access to both P, the real location of the geometry in object space, and Porig which is the point P before it was deformed. We now color point P based on the noise(Porig) and that way the three dimensional noise pattern sticks to the surface as it bends, as opposed to the surface bending through noise space which would be, well, completely wrong.

1/19/94

It actually got worse than the last take. About everything in this version is broken. This was the emotional low point of the production. It didn't get any worse than this. How could it? The spot is due in 7 days and there is no hope.

1/24/94

Things are looking much better. The lighting is coming together nicely. All the models have been fixed and aren't crashing machines anymore. The shaders are almost all done. We've even added in some really nice looking grass that is almost completely out of frame with respect to NTSC. But all you Siggraph cognoscenti will appreciate it.

1/26/94

Final, as of 4:22 pm. We popped the champagne, we drove home, we slept for several days.

Thank you and goodnight.

Computer Graphics at Walt Disney Animation

M.J. Turner
Walt Disney Feature Animation

Introduction

Somehow the public gets the idea that computers do everything. And that somehow it's the computers that make the movies and these computers are taking the place of humans. Nothing could be further from the truth.

It is true that some things the computer is very efficient at but, it doesn't matter how sophisticated these computers become they will only be able to do exactly what someone tells them to do. And that someone has to be creative in order to produce interesting pictures. Computers are *not* creative. That is where Disney's success has been; investing in creative people to produce entertaining pictures that audiences will want to return to time and time again.

The following is a general flow of some of the basic steps involved in the making of an animated film at Disney. In particular, the descriptions will take on a certain slant toward the involvement of the CGI (Computer Graphics Imagery) Department.

Within the Rendering section there is a discussion of some of the specific details about what was done to produce the 3D pictures for a few of Disney's latest films.

Story

Every film starts with a story. A story can be anything from a simple idea to a script or an adaptation of an existing story. Once a story is given a "green light" to become a film it goes into pre-production. And from there the story will be continually fine tuned all the way through the production of the film. When a story enters the pre-production phase the different department leads begin to work out what they can provide for the film.

As pre-production proceeds the directors will work with the CGI lead. Their purpose, in the beginning, would be to come up with the most *bang-for-your-buck* shot, that would provide an important story telling point in the movie that could not be achieved with conventional methods. In other words, how could CGI be used to help make an impact on a dramatic story point.

It is important for CGI to be involved early in the pre-production in order to have enough lead time to develop the special software that is needed to produce the CG images for the film.

Other CGI elements that have been tried and true are used for helping with the overall look of the film. These elements may not be highlighting a dramatic story point but they help with the overall mood of the film. The advantages of using a computer for animating props is that the animator does not have to spend as much time worrying about how the object looks but only about how it moves. Therefore the computer can be used to draw the objects and the animator is free to animate.

Storyboarding

Once the general story is nailed down and/or a script is ready, artwork is drawn on small cards to represent the action from scene to scene. These cards are then posted in order onto bulletin boards, called storyboards. A storyboard is a "comic strip" version of the movie. The story development that

happens at this level is to decide how the different scenes are going to fit together from scene to scene and sequence to sequence through the film. The storyboard is then transferred to film with the proper timing for each scene. This is called the work reel.

The storyboarding process continues all the way through the pre-production and production of the film. Fine tuning it all the way to the end.

Layout and the Work Book

The next step is to take the story sketches and work out the final staging and dramatic angles for each scene. This information makes up the work book. The work book then becomes the guideline for the rest of the staff to know how the scenes are going to hook up with each other.

A CGI workstation can be used for planning 3D camera moves. 3D stand-ins can be used to help rough out the action in order to determine where the camera should be for the scene. It is also decided where it makes sense to do 3D moves as opposed to 2D traditional moves on artwork.

Animation

This is the phase where the “comic strip” comes to life. Animation is where the characters and their environment come to life. The majority of Disney Animation is still done with pencil and paper (*pencil animators*). The CGI animation (*computer animators*) makes up a small portion of the animation that is created. However, CGI involvement is ever growing with each new film.

The CGI department has a variety of animation types that are performed. They are based on the type of character or environment that is being animated.

- **Character animation**
This is articulating all motion for a given 3D character.
- **Behavioral character animation**
This is when there are multitudes of characters that need to be in scenes behaving in specific manners (behaviors). The behaviors are animated in such a way that they can transition between any of a set of like-minded behaviors. Each character will have a set of like-minded behaviors. Then they are combined with in-house software to behave in a semi-random fashion. They can then be choreographed as scenes dictate.
- **Prop animation**
This is animating motion of a rigid 3D model.
- **3D camera moves with 3D environments**
3D models are built for the stage and camera moves are animated through them. This very often includes character animation from 2D and 3D means.
- **3D camera moves with reference models**
This is for giving the pencil animator perspective reference of the character to be animated.

Color Modeling

This is where the Art Director and Directors come to see, for the first time, all the elements of a scene together in color. Color modeling is where they decide what the look for the scene should be. They are not only picking the colors but deciding what compositing “tricks” should be used to get the effect they want for the scene. They will be looking at pencil drawn as well as computer drawn elements.

All the CGI rendered elements are rendered as separate elements (i.e. a paint level, a tone level, a shadow level, a depth level etc.). These are then put into the in-house compositing system called CAPS (Computer Animation Production System) that will allow the Art Director and Directors to control how much of these different levels they choose to see. CAPS has a wide variety of compositing “tricks” at their disposal for creating a look and feel that is right for their film.

For instance, the depth level is used with a dust turbulence **over** the tone; that is used with a percentage value **over** the paint level **over** the shadow; that is used with a percentage value **over** other elements; that is finally **over** the background.

Plotting

CGI needs to interact with the 2D artwork and vice versa. Therefore, a way to give CGI artwork to a *pencil animator* is needed in order to line up the 2D artwork with the CGI artwork. So, one of the steps in the CGI process is to send 3D animation to a plotter for transfer to paper (the 2D artwork is scanned with a digitizing camera in order to line up 3D artwork with 2D).

The 3D animation is rendered with an in-house RenderMan driver that has an understanding for implementing hidden line drawings. A set of special shaders are used to generate pixel-based images with enough information to generate vectors which approximate a hidden line drawing of the 3D model. These vectors are then passed to an in-house plot program which sorts vectors and plots them on a pen plotter.

The driver understands value changes for each of the RGB channels that are passed by the shader assigned to different surfaces. The edge detecting software is based on the Sobel gradient operator. By assigning different levels of blue to different surfaces it will detect an edge and thus draw a line between them. It will use red and green in a similar fashion for lines desired inside of surface boundaries. Control for the lines is achieved through the shaders defining RGB values across the surface.

Rendering

Because of the special needs for image output that CAPS requires, just like the plot driver there is a RenderMan driver for generating CAPS image format.

Early CGI

Rescuers Down Under was the first film to actually have 3D rendered elements. Prior to that CGI plotted to paper right along with the 2D animation and was xerox onto cell. Then to Ink & Paint to be painted by hand. And finally, placed under a camera and shot to film.

Decisions are made early on as to how much the 3D elements need to blend in with the 2D elements. Most of the time it is desired to blend in seamlessly. However, there have been times when it was a design decision to make the CGI elements stand out. One such time was the ballroom sequence for *Beauty and the Beast*. It was a design decision to make this a special moment for Belle and the Beast. So, a fully 3D environment with 3D camera moves around the 2D characters was used. It was a magical moment to be captured separate from the rest of the film.

The Lion King

The following are the rendered elements of the wildebeest.

- **body**
Flat shading was used with black color at the hooves. Color Models determined the range of

colors that the wildebeest could vary between. Then randomly they would get one of these values applied to their bodies.

- **face/stripes**

Also flat shaded but, a texture lookup file was created that defined with red, green and blue where the color separation would be. This way the color assignment could happen in the shader and the texture would not have to change for each different color assignment.

- **character lines**

The most interesting aspects of the rendering of the wildebeest was the character lines. The RenderMan plot driver (described above), provided the flexibility of creating lines that fall-off based on certain parameters and the geometry. For instance, the lines for the legs up into the body. It was not a flawless system but, at least it took it most of the way there and then the rest was able to be cleaned up. An interactive tool would display the scene frame by frame and allow adjustments to be made to extend some of the lines and erase others.

- **tone**

This is a black & white image that defines what part of the wildebeest was in and out of shade. It is based on the normal to where the light is. For the wildebeest it was high noon. This level would define where the shading should take place and then within CAPS it was determined how much of the effect would get applied over the paint level.

- **shadow**

A cheat. Texture map of the wildebeest from above was mapped to a rectangle underneath the wildebeest that followed it around everywhere.

- **hair**

It was decided to animate the hair for the wildebeest by pencil. We could have written some dynamics for generating the hair but it was decided to go for a more hand drawn look. The hair was drawn once for each behavior.

The hair for the back and chin was done by texture mapping the hair animation on four different patches. These patches followed the animation of the wildebeest.

- **chin**

Two of the patches penetrated out of the chin of the wildebeest. They butted against and pointed out from each other. They also had a slight warp to them so they wouldn't be perfectly flat. And finally a slight displacement, so that when they were looked at straight on they would not disappear.

- **mohawk**

This patch penetrated straight out the back of the wildebeest. It also had a slight warp to it so it wouldn't be perfectly flat when observed from the front. This patch was used so that a side view would show hair up off the back of the wildebeest.

- **drape**

This patch laid just above the back of the wildebeest. This helped pull the hair away from the body so that it didn't look like the hair was just pasted to the body of the wildebeest.

- **tail**

A cycle of tail motion was animated with pencil and then scanned in for creating texture maps. Then a special model was created for the tail which then got the texture cycle mapped onto it. It was transparent where the tail drawing did not exist.

Pocahontas

Most of the work in *Pocahontas* is done with a variety of texture mapping techniques. For the ship and canoe straight ahead texture mapping was used. In-house software was used for mapping curves onto a surface. Curves would be constructed onto the surface of a Nurb and then they would get flattened out and plotted to paper where the background department would then paint the detail of the surface on boards using the curves as reference. Then those paintings would be digitally scanned in and then reverse the process and voila - a painted 3D model.

(Note: There are some really good 3D paint systems on the market but they still don't seem to give the background artist all the flexibility desired. Nor do they provide a medium in which background artists are most accustomed to working in (i.e. paint on canvas)).

Because Grandmother Willow's form moves and the motion of the bark was to have a certain look we developed our own method for texture mapping the pieces of bark. The painting of the original bark was done relatively the same way as the canoe and the ship but the technique that was used to apply the texture to the surface was where it differed. It was desired to be able to "float" the larger pieces of bark over the surface of the tree without having them stretch, but not so much so that it looked like these pieces were floating away from Grandmother Willow. We designed a way to define certain regions of the surface. Then with that region, the static position of the surface and the animated surface a texture map would be generated that defined the parametric space of the surface. This was then used with the texture map of the painting of the bark to paint the surface of Grandmother Willow.

Fantasia: Pines of Rome

The whales were rendered using a variety of shading tricks. The most difficult part about rendering these mammals was seaming up the surface boundaries. Again some in-house software was applied to the whales to make the Nurbs continuous from Nurb to Nurb. With that in place it was also necessary to have a reference model of the whale in it's static zero position in order to prevent any grotesque stretching of the surfaces. In addition to applying surface properties based on where the model in the static position is in world space.

The bumps on the head and jaw, the pleats on the belly and the cuts on the flippers were generated by displacement shading.

The water is generated by several different components. The first is the general motion of the water which is a spring-based dynamic motion. On top of that is applied a partial system ripple effect that is defined from where the whale breaks the surface of the water (xz-plane) and propagates out from there. Then there are a set of shaders that sit on top of all that to define the color fall-off for the water and the sparkly hilites on top of the water.

Conclusion

Animation is an enormous collaborative effort. It takes a tremendous amount of work and dedication to get a feature film conceived and distributed to theaters. I hope this has served as a helpful glimpse into part of the work that is done within the walls of Walt Disney Feature Animation.

CG Lighting Design for Feature Films

Joe Letteri
Industrial Light and Magic

In film production, the cinematographer uses lighting to create the mood of a scene and help deliver the emotional impact of the story. That is also the role of lighting in computer graphics for film. Matching real-world lighting can make objects look more realistic or make a scene appear more natural. Lighting can also help bring characters to life.

There are several different examples of lighting for different types of situations, including:

- Exteriors - outdoor light in broad daylight
- Interiors - including direct and/or indirect light from multiple sources
- Nighttime scenes

Exteriors

At first glance, daylight would seem to be an ideal lighting situation for CG. You can model the sun quite convincingly by treating it as a point light source that is infinitely far away and whose rays arrive at the surface in parallel. RenderMan provides a `solar` () construct in the light source shaders for you to do this.

The difficulties arise when one considers the ambient light in the scene. In a typical daylight scene, a small, but not insignificant, contribution comes from two main sources: the sky re-radiating light from the sun, and sunlight that bounces from the ground or other nearby objects. In both cases this light is non-uniform and directional, so it becomes complex and difficult to capture.

The standard CG method of using a constant term for ambient light is insufficient. The result is a surface that is rendered as too flat, with no variation in lighting. There is also a loss of surface detail due to lack of shading.

One solution is to use multiple diffuse light sources. Turning off the specular contribution of a light that has a point source gives it a more diffuse quality, making it appear more like a bounced light. However, an indirect light produces a soft shadow, so care must be taken to hide the hard shadow edges of each point source that is used.

Example: Brachiosaur sequence from *Jurassic Park*

This is an example of lighting a large, slow-moving creature in broad daylight. The main source of light is from the sun, which is high in the sky and slightly away from camera. Note that, as is common with an exterior sequence, the position of the sun has changed during each shot. In order to provide continuity, though, we placed our CG sun to provide a balanced source for the entire sequence.

Since the sun was coming mostly from the far side of the brachiosaur, we had the additional challenge of creating natural-looking ambient light to fill in a large part of the creature.

Note also how cast shadows were blended with the shadows on the ground and in the environment to help integrate the lighting into the scene.

Interiors

Interior lighting can be complex due the number and nature of the light sources that might be used on the set. However, the existence of multiple sources in the image gives the artist more freedom in designing the CG lighting. The preferred method is to emulate the way a set would be lit, using a mix of lights in their traditional roles as key light, fill, rim light, etc.

RenderMan allows one to easily create a library of light sources that can be used for various situations and effects. These can simply be useful variations of a standard spotlight shader: for example, adding a “barn-door” falloff to the edges of the beam to control where the light falls, or using distance attenuation of the light source to help create depth within the scene. A light shader that allows one to project an image is useful for simulating light coming through a window. One can also extend this technique to create a *cucoloris*, which in traditional film lighting is a shadow mask placed in front of the light to create a shadow pattern. This can be very effective to create a sense of space within the scene.

In fact, the use of shadow can be very important in creating the mood of a scene. Shadow is not merely the absence of light, but is a major compositional element in lighting design. A creature or character moving through shadow gives the viewer important clues about the space that it is in. Shadow can be used to hide an action, to make it more mysterious, or, by contrast, to direct the viewer’s attention to an area that the artist wishes to highlight.

Examples: Raptor kitchen sequence from *Jurassic Park*

This sequence uses multiple light sources to match the light coming from various places on the set. Shadows are used to dramatic effect to make the raptors appear more mysterious and dangerous. Shadows and reflections are also used to place the creatures physically within the set.

Night

Lighting night exteriors is in many ways a combination of daylight and interior techniques. The convention in film is for night scenes to be lit by a single source, tinted slightly blue, to represent the moon. In fact, in the standard film trick of shooting *day-for-night*, the “moon” is actually the sun, with a darkened exposure and a blue filter.

In scenes where an artificial light is being used for the moon, a blue gel is still usually placed over the light to give it that nighttime look. Additional lights are placed as needed, much as on an interior scene, all made slightly blue to give the impression of the moon as the main source of light. As long as the “moon” remains the dominant source, we are accustomed to seeing additional lights in the scene, without which not much else would be visible. This gives us the freedom to mix in our own light sources when necessary.

Examples: Tyrannosaurus Rex from *Jurassic Park*

In this sequence, the primary intent was to create a mood that was highly-charged and dangerous. Because the script called for a stormy night, the scene was shot with the addition of lightning and rain, which we had to mix in to the CG lighting design.

There was extensive use of “hidden” light sources. We decided to light this scene as if we were using real lights with a real creature, so as it moved it travelled through them. This helped to give the TRex an impression of great size while at the same time giving it a sense of moving through physical space.

Looking at Light

In lighting a scene, a cinematographer has to make two critical choices: what kind of light to use, and where to place it. To match the look of natural light in a CG environment, the artist needs to understand what is important about those choices, and how best to simulate them.

The relationship between the light and the camera reveals the most to us about the forms of the objects we are viewing. Light coming from the camera direction, for example, imparts a flat tone to the scene. As we vary this angle, we begin to see more shadows, which reveal more to us about the shapes of the objects we are lighting and their relationships to each other in space. In traditional film lighting, the light that sets this tone for the scene is called the *key light*. So if we are rendering CG elements into a background plate, the first step in matching the lighting is to recognize the placement of the key light.

Once the key light is in position, the cinematographer may use a *fill light* to allow us to see detail in the shadow areas. Often, this light is bounced indirectly from either the same or a different light source. In fact, it may become apparent that indirect light provides most of the additional light in the scene. If this is the case, we may have to resort to blending multiple single sources to create this same effect.

We encounter a similar difficulty in trying to simulate any source in which the light is either bounced off another surface or passed through some diffusion material before reaching the subject. This causes the light to produce broad highlights on the surface and soft, sometimes indistinct shadows. To duplicate this in CG, we need to use either areas light sources or solve for radiosity; unfortunately, neither option are available to us in the current version of PRMan.

Light and Shadow

With light comes shadow — at least in the natural world. In CG, however, we can choose to separate them. In the real world, the quality of a light and the quality of its shadow are interrelated.

Real shadows exhibit several characteristics that are difficult to capture in CG. First, real shadows areas will usually have some light remaining in them. Even in a scene with only one light source, nearby adjacent surfaces will scatter some of that light back into the shadow area, and into the surfaces that are hidden from the light. RenderMan allows you to cheat this effect by rendering a shadow that is not fully dense, so that some of the light leaks through. This works sometimes, although a better method is to use a fill light in this area; the tradeoff is the expense of rendering additional lights.

We can divide CG shadows into two categories by function: *self-shadows* and *cast shadows*. Self-shadows are those that are rendered within the scene; for example, the shadow of an arm across a torso, or of one creature on another. Cast shadows are those shadows that a CG object would cast on a real object that already exists in the background plate.

Since self-shadows are contained within the rendered image, their treatment is determined by the lighting on the CG subject. Cast shadows, on the other hand, need to be composited into the background in such a way that they appear to have existed at the time the scene was photographed. In this case, matching the placement of the key light in space becomes critical, as does matching any real objects that the shadow needs to fall across. It is also important to match the quality of the cast shadow to that of real cast shadows in the plate, paying particular attention to the shadow density and edge quality.

Photorealism

In using computer graphics to create visual effects for feature films, we are often presented with the criteria of making them appear “photoreal”. Put simply, this means creating an image in which all of the elements appear to have been photographed in the scene by the real camera. Achieving this effect calls for an understanding of all the elements that comprise an image on film.

While careful lighting design is important in creating the look and feel of a scene, a great deal of that work is “invisible” to the viewer. That is, the audience sees the light only in the ways in which it reflects from the surfaces it is illuminating. Thus it is important as well to capture the distinct qualities of those materials. Creating the look and feel of a creature, character, or object requires a multi-layered approach.

Often, the first place to start is with the color, since that is usually the most recognizable attribute of a surface. One of the temptations of having a powerful shading language like that in RenderMan is to try to develop complex algorithms to define the look and feel of a surface. However, a much more intuitive approach to color is to rely on painting texture maps. RenderMan allows the user to map a texture to a surface in a number of different ways, so the choice of which is appropriate depends on the type of object you are creating. For example, placing a label on a bottle may be done with a cylindrical projection. For a more complicated model, however, such as a creature, a more ideal method would be to use a 3D painting tool that allows the artist to paint directly on the CG model, in much the same way as painting a physical model. The output of this program is then passed into RenderMan, and the painted textures are applied during the rendering.

We can use the same tools to create the surface texture by painting bump or displacement maps for the model. Again, the explicitness of this approach allows the artist a fast way to design and create the desired texture.

Extending this technique, then, we can paint a series of maps to create variations in surface quality. For example, specular maps can be used on a creature to show variations where the skin might be worn smooth in some areas and rougher in others.

We can return to the idea of using procedural shaders to create the fine detail, such as cracks in the skin or wetness in the tongue. Especially when seen close-up, this last layer of detail provides the visual richness that is characteristic in photographs of real images. Also, procedural shaders come to the forefront when creating a dynamic effect, such as rain, which would be difficult to paint.

Finally, when we put all these elements together, we need to think about how they would look on film. In any photograph, the image captured on film looks different than the real subject. The same applies for CG images output to film - they will look different than they do on the monitor. This is due to a variety of factors, ranging from the color response of the monitor to the properties of the film stock. Since film is our final output medium, the artist needs to become aware of these effects, and learn how to take them into account during the lighting.

Creating Computer Images for Film Using RenderMan

Ellen Poon
Industrial Light and Magic

The Production Pipeline

RenderMan has been used for many productions at ILM since the mid 1980's on films such as *The Abyss*, *Terminator 2*, *Jurassic Park* and *The Mask*. It offers features such as motion blur, depth-of-field, shadows and a very flexible shading languages which allow us to generate photorealistic quality computer graphics images that are difficult to obtain otherwise.

A computer graphics shot is typically broken down into the following phases. When designing the approach of a shot, RenderMan is being taken into consideration at each and every one of these phases:

Modelling

Modelling can often be simplified when some of the contouring of an object can be achieved through the use of displacement or bump maps. Also certain parts of the object might not even have to be modelled when a color map that gives the equivalent appearance is applied instead.

Animation

Procedural animations are often required in many of our effects shots. They can be generated by various techniques such as using a particle system, or alternatively, we can design a shader bearing that effect in mind and by varying the values of the parameters to that shader, it gives us the animation we are looking for.

The motion-blur facility in RenderMan is crucial when it comes to creating realistic motions. Motion blur gives the animation its believability and life.

Lighting and Creating Surface Materials

A lot of what we do is in fact recreating reality inside the computer. In particular, we first have to match our lightings to the lightings designed by the Director of Photography of the film on the set. If it is an outdoor scene, we will need to light it so that it looks as though it is lit by the sun with a lot of bounce lighting from the ground and general ambience. Second, the look of the CG creatures and props to a great extent have to resemble the look found in photographic references or the actual props that were used by the actors on the set when they were shooting the movie.

To this end, custom written shaders are used extensively which enables us to create complex looking surfaces and to simulate realistic lighting situations. The main categories of shaders are light, surface, displacement and atmosphere. One of the many techniques we employ is to combine these different kinds of shaders together to get us a wide range of looks and materials which in turn gives us the creative freedom when it comes to designing the appearance of our CG creatures and objects.

Rendering

Many test takes have to be rendered before the shot can be taken to the final stage and eventually be scanned and filmed out. By adjusting rendering parameters such as the resolution of the image, the

shadow buffer size, the pixel sampling rate and the shading rate, we have a fairly flexible system to produce rough takes as well as final takes.

Painting

Painted textures maps are very often employed to give the objects their colors and textures, instead of creating them procedurally. The shader would make use of the `texture()` function and it has the added benefit of prefiltering the image and therefore eliminating aliasing artifacts. Other enhancements to the look of the image can be facilitated by creating transparency maps, specular maps, bumps maps etc. in painted forms.

Using painted textures is thus a very powerful tool because it gives an object a realistic look very quickly.

Rotoscoping

Since most of our shots have to be combined with live action elements, roto mattes are necessary to place certain part of the live action bg plates in front of the computer generated elements. However, a RenderMan “matte object” can be used to block out certain parts of the foreground CG objects when rendering the scene so that minimal roto-scoping is performed.

Compositing

An alpha channel is generated together with the three color channels in all the rendered images, which allows compositing to be performed rather painlessly. Another interesting feature in PRMan is that it can be made to produce depth maps for the rendered image, making depth-based compositing possible.

Case Studies : *The Mask*

Specially written RenderMan shaders were used for most of the shots in *The Mask* to create interesting looks and effects such as the tornado, the Mask itself, and the superdog Milo.

Milo the Dog

The shader that gives it the mischievous look is essentially a plastic shader with color mapping and bump mapping features incorporated. Color maps give Milo’s head the skin tone and the bump maps give it the short hairy look that is found in Jack Russell Terriers. This saves us a lot of rendering pain because there is no hair rendering involved!

Tornado

The tornado was modelled as a bunch of ribbons which were animated to spin around. The noise function in the tornado shader provides the key to giving it the wispy look.

The Mask Wrap

The hard mask itself was first modelled without cracks, and then cracks were put in through the use of bump maps. In the mask wrap sequence where Jim Carrey puts the mask on for the first time, the lightning effects coming through the cracks were achieved using a combination of animated snake-like objects which were then rendered to be used as guides to reveal the lightning map animations. He also struggles with the magical Mask and digs his fingers into it whilst it is trying to get ahold of his face. The effect could have been done with a lot of painful CV animations, but in fact it was achieved by creating a set of animated maps that follow his hands and fingers movements which

were then applied as displacement maps in the shader to give the illusion that grooves were formed beneath where his fingers are.

Tricks I Use

Avoid complex modelling of geometries whenever you can by using various different textures such as bump maps and color maps.

Sometimes intricate animations such as shape distortions are better performed within the shader through the use of displacement maps.

Always look for photo or video references before proceeding, for both lighting work and materials design.

Don't think you will get a perfect shader the first time round. Try using a simple shader like a plastic shader to start off with and keep enhancing it.

Keep functions as modules so you can cut and paste them into the "new" shader that you are working on.

Painted textures vs. procedural ones.

Make use of multiple texture co-ordinates to give you more freedom when doing texture mapping, s and t and $s2$ and $t2$.

Energy Ribbon Effect for *Star Trek: Generations*

Habib Zargarpour
Industrial Light and Magic
Copyright © 1995 Lucas Digital

Abstract

The following is a description of the techniques used to create the energy ribbon effect for David Carson's film *Star Trek: Generations*. Also discussed is the creation of the computer generated Enterprise-B. The Energy Ribbon was a space phenomenon on a planetary scale that traveled at high speeds through the galaxy, trapping and destroying objects it encountered.

Introduction

Frequently, computer generated effects are used to recreate existing or tangibly real objects or creatures. They mimic life and attempt to be substitutes for the real thing. But sometimes there is no earthly equivalent of the effect required, and even art work and arm waving can just begin to describe what it might be. Creating abstract effects are very difficult and require good art direction and concept work to be successful. The energy ribbon sequences in *Star Trek: Generations* involved such an abstract phenomenon: a rip between two different universes, a tear filled with tense energy. There was really no way of obtaining references for any part of it. Of course on the positive side, since no one had ever seen one of these energy ribbons, it could have looked like anything. But the endlessness of possibilities easily made up for the apparent freedom of creation by making it very difficult to visualize and find the best method possible to execute the effect convincingly.

Conceptualization and Art Work

In order to make sure that the concept is what the director wants, it is critical to generate art work that tries to portray the nature of the beast, and is approved by the director, especially in the case of intangible effects. This simple step can save weeks, if not months, of difficult and lengthy research and development that may eventually end up being the wrong idea, i.e., not what the director had in mind. After some preliminary ground work and lots of research, art work was made for the energy ribbon showing it to be an airfoil cross-sectioned "wing" core that would travel at high speeds leaving a trail of debris in its wake. It generated a magnetic field perpendicular to its direction of travel and the field continued cylindrically along the entire length of the ribbon. The ribbon core was the brightest section and was where the temporal flux occurred. Behind the core was the ribbon tail. It was a thin sheet of energy that tapered away from the core. Erupting out of the head of the core were the energy tendrils that were constantly moving and arcing along the magnetic field paths that surrounded it. Behind this effect was the ember field—a swirling debris field of glowing energy particles left behind by the violent action of the ribbon.

Although the art work is a big step toward nailing down a look, it does not determine its dynamics and movement. The ribbon core was to have some form of undulation in space, and the type of move was to be discovered along the way. We knew the movement had to be graceful and self propagating

like waves in an ocean. The behavior of the tendrils started out as smooth magnetic flux activity that began in front of the core and grew up and over the wing to stretch out and disappear before reaching the tail. The director wanted a more violent look, so the motion was changed to be more electrical, yet maintained the swinging front to back behavior like fast energy whips lashing randomly and becoming visible several times as they roll back. Story boards were drawn up at this point along with cross-sectional physics diagrams of the thing so as to convince us that such a phenomenon really existed.

Narrowing Down the Problem: Computer Graphics vs. Traditional

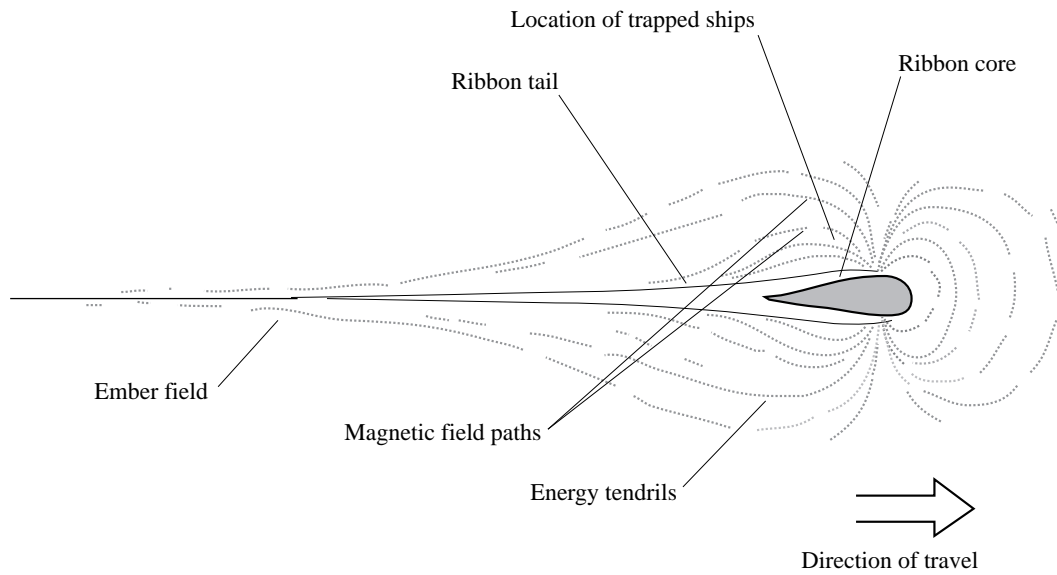
Of course before jumping into producing an effect, one has to carefully weigh all the various options and methods of execution. Not all effects are best done using computer graphics techniques. In the case of the energy ribbon, with the massiveness of the scales involved and the amount of control required over the look, computer graphics was the natural choice. There were also scenes in which various spacecraft, including the Enterprise-B, would be directly interacting with the energy ribbon, trying to escape its claws. There were to be particles streaming around the Enterprise as they were being deflected by the shields, and lots of interactive lights from the ribbon that would affect the ship. For these reasons, all the scenes involving the Enterprise with the energy ribbon were slated to use a computer generated model. Other scenes of the Enterprise would use the stage model. The main problem would be getting a close enough match between the two ships so they would be indistinguishable between cuts. The new ships, such as the Lakul, trapped within the core could be made digitally and would not have the intercutting issues.

Energy Ribbon

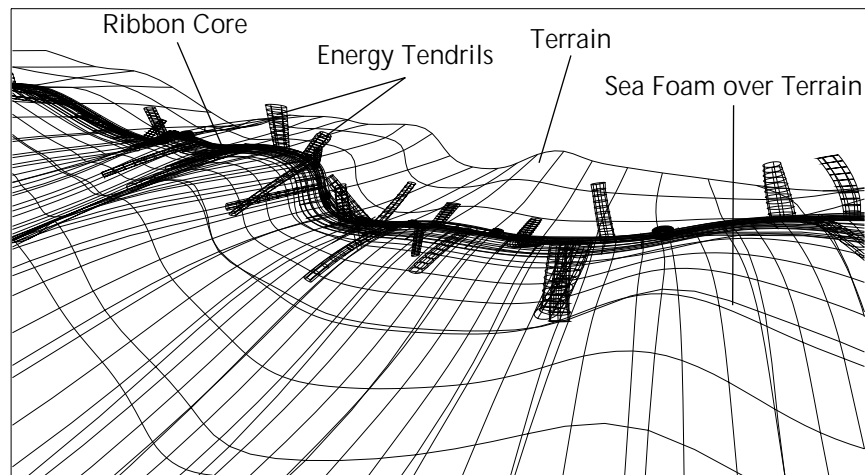
One of the key characteristics of the ribbon was the trailing wake of embers that streamed off the ribbon and followed the movement of the core just as it came off the back. It had to be like markers for where the core had been when it traveled through that area. We later came to call the ember trail “sea foam” as a good way to describe its look and feel.

After deciding to use computer graphics to tackle the problem due to the enormity of the scales required, various tests were done to try and obtain a look similar to that of the art work. The ribbon core could easily be executed using geometry with special shaders, but the trail of embers could be done using either particle animation or shaders on a surface. The decision was made to mainly use a surface given the thin sheet aspect of the trail, and use particles to enhance the details. The reason for this was that it was very tricky to fill the entire screen with particles and get a good feeling of depth and change of scale. There were storyboards that showed the camera looking directly behind the ribbon with the sea foam changing scale from a few thousand miles away to directly underneath. You would spend most of the efforts trying to disguise the particulate look and deal with the scale change issues. We later did use particle simulation to create plasma that erupted off the core due to its upward or downward motion and stayed relatively small in frame. The power of the RenderMan shading language and three-dimensional fractals were perfect for the sea foam element and its turbulent behavior.

To create the sea foam we tried various animation tools to try and simulate the effect of trailing materials using spline deformation in Softimage, but encountered several problems. If you animate a spline and then deform geometry to that spline, there is no way to control the perpendicular tangents which will change dramatically if the spline points get too close. The second problem was that there would always be a separation between the core and the leading edge of the sea foam because of the different patch resolutions needed for each element.



- ENERGY RIBBON CROSS-SECTION VIEW -



If you consider the animating spline at the center of the core as it travels through space and save the shape at regular intervals of time, you would be looking at key frames of the animation as it relates to time and space. Now if you connect those splines you will have a surface that represents the curved slice through space made by the ribbon core. Any material that trails off the ribbon would be positioned on that surface. This led us to using the technique of 'modeling' the key frames of the spline by simply shaping a patch to form a sort of terrain, then using the patch deformation tool in Softimage to make the energy ribbon core, sea foam, and tendrils all have their coordinate system warped to follow the terrain. We animated the ribbon to travel over the terrain, then counter animated the terrain to travel in the exact opposite direction to get an undulating ribbon that was not actually going anywhere. This made it easier to animate other elements such as the Enterprise relative to the core, and to let us use fixed lights for the ribbon effect. The patch deformation transformation is such that the world space coordinates x-z get mapped unto the patch u-v space, and the y axis becomes the normal at any point on the patch. All objects maintain their relative positions and track the surface closely.

Ribbon Core

The core was a wing foil cross-sectioned patch with a heavy amount of displacement in its shader. We had decided to use procedural shaders to their fullest extent because of the large amount of control and the ability to animate any and all values to obtain the look and movement the director wanted. Some painted textures were used in conjunction with the procedural fractals, but only if they did not need to be animated. In both cases of the ribbon core and the sea foam, several three-dimensional customized fractal functions were mapped onto the patch's s-t coordinates. This allowed the cloud forms to be animated through the surface and create a billowing effect. The same procedural values were then used to displace the surface along its normals. In the case of the core, the shader space was stretched along the wing cross-section so that any displacement would be streamlined uniformly all the way to the trailing end of the wing. To enhance the hot, flame-like surface, the edges were made to be more opaque than the center areas facing the camera. The displacement was tapered off along the cross-section towards the trailing edge of the wing in order to keep the patch seam from separating and also because of the need for a thin edge.

Energy Ribbon Tail and Ember Field

The sea foam used a similar shader to that of the core to keep a consistent feel and look. It eventually evolved into a much more complex version of the same idea. The art work had two very clearly distinct sea foam elements: a main orange-red element that formed the main trailing debris with large and small plasma fragments, and a smaller trail of yellow streaks that faded out quickly behind the core. Two separate render elements and shaders were made to create each effect. The shader contained many levels of three-dimensional fractals that determined the brightness, opacity, and displacement of the surface. In order to give the main sea foam element a feeling of depth, two completely separate layers of fractals were put in place. The first was used to render the main volume of plasma clouds trailing behind, and the second was very small fragments that traveled backwards at twice the speed, giving an illusion of depth to what was actually a single surface. Since the final film frames would not be viewed in stereo, and we never actually had to go through the sea foam, it did not make any difference whether these elements were rendered on completely separate surfaces, or if they used the same one with other visual cues such as scale and speed to distinguish them. The issue of color made several rounds throughout the project. Initially we were told to avoid the orange warm colors that were in the "approved" art work because it looked too much like fire, but the blues

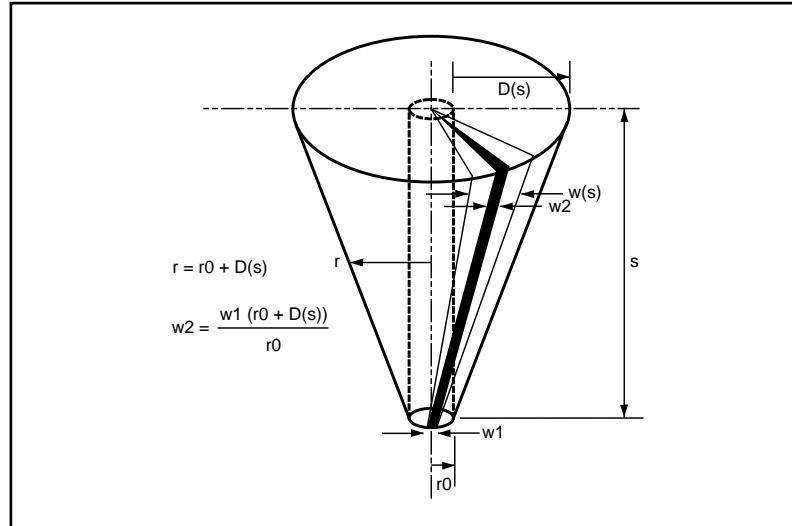
and purples were not so good either because a lot of the effects on the Star Trek films had used those colors, and green wasn't exactly pretty! Needless to say, the warm tones prevailed. Later another fractal variation was added to sea foam shader to vary the hue slightly over the surface. This feature turned out to be very touchy to control (its so easy to end up on the wrong side of the color wheel). All the various intensities were faded down as they got farther from the core, and a warp was added to the surface geometry so that the textures would start slow and speed up as they moved backward.

Energy Tendrils

This effect by far became the most difficult and challenging one to tackle. These were to be distinct thin whips of intense brightness that formed on the leading edge of the core, scaled up in length as they whipped backward over and under the core, and finally disappeared before reaching the surface of the sea foam. They also had to randomly appear and disappear as they rolled back from the front. After various attempts at rendering the tendrils as flat surfaces cut into many hundreds of sections, we decided to actually simulate the tendrils as though they were physical chains undergoing a lot of turbulence. We needed the tendrils to actually react to the movement of the core, adding side to side whiplash as a result of the wavy undulations. No matter what we would do with procedural shaders, we would not be able to obtain the liveliness of a physics simulation. During the course of production, we had divided the energy ribbon effects into two categories that could each be dealt with separately: the distant shots of the ribbon where you can see the sea foam trail, and the close up shots right on the surface of the core where the tendrils would actually break frame. For the tendrils used in the close up shots, there would only be six to eight tendrils which would allow for using actual cylinders that were spline deformed to shape animated splines in Softimage. A procedural technique was developed for inserting random key shapes into the splines every four frames. For the distant shots, there would have to be as many as 300 individually spline deformed tendrils. Not a pretty sight for any animation software. Also, the random shape animation worked well for the close up shots because the tendrils would extend from the bottom to the top of frame and did not need to react to the movements of the core and did not need to have any visible physics whiplash behavior.

Although the art work clearly defined the tendrils to be loners never traveling together, during one of the close up tendril tests some of them accidentally grouped together in twos and threes due to some mix-up with the spline deformation. We all agreed that made for an interesting look. To create the simulation for the tendrils used in the distant shots, we first animated about 100 cylinders in Softimage that would scale out from the front of the core and rotate backward, and start the whole cycle over again. They were also connected to the same patch deformation that ribbon core and sea foam are using. These were then exported into Wavefront's Dynamation software and converted into spring models with varying degrees of inertia that increased from the base towards to tips. This would cause the cylinders to follow the animation exactly at their roots and have their tips trailing behind and free to be affected by external forces. Turbulence, drag, and other forces were put in place and the resulting animation was exported into ILM's own file format for applying shaders and creating rib files for rendering. It was at this point that the decision was made to go ahead with the triple or double tendrils. This would mean we would have to create three times as many tendrils and find a way to keep them together in groups through the simulation. A RenderMan solution to this problem would save us a lot of agony and give us a great amount of control on such things as the width of the tendrils, the number of tendrils per group, and their electric activity. We could use the single cylinders and split them into multiple thin slices based on a similar technique used in the checkerboard shader in the RenderMan Companion. The cylinders were divided into two or three thin sections along their vertical axis, and a displacement animation in the shader added the necessary detail to

give a more electric feel to the tendrils. Because the groups would be part of the same cylinder, they would always stay together.



In order for the displacement to work properly, we needed to compensate for the increase or decrease in the radius of the cylinder as a result of the displacement because the splitting was a function of the parametric coordinate 's' along the cross-section. This compensation was figured out using geometry, and turned out to be simply an expansion of the perimeter as a function of the increase in the radius, and is directly proportional to the radius. We encountered a problem with the geometry not being divided into enough micro polygons to give us a smoothly displaced model. The reason for this was that the cylinders would start out fairly thin, and the shading rate would determine a subdivision based on the screen size of the geometry before it got displaced so that we had to use ridiculously small shading rates, like 0.05, to start making a difference. The solution was simple: instead of taking a thin cylinder and displacing it outward, we would start with a fat cylinder and displace it inward. That way there would be more than enough micro polygons to go around since they would be based on the screen size of a larger object. An intermediate width was finally used so as to not affect the simulation drastically and the surface displaced in both directions. The fractal function was animated to move through the surface (slices of the 3-D fractal) and also move up along the length of the tendril. The benefit of using displacement within the shader to obtain surface detail and electric animation is the small amount of geometry required. Obtaining similar results using just geometry would require a staggering amount of modeling and animation data. It would be like animating at micro polygon resolution.

The tendrils also had to randomly appear and disappear. In order to simplify the process, a texture was painted that looked like a bar code image. The cylinders were all lined up vertically, and s-t texture coordinates were created. For this, the cylinders were shuffled randomly so as to hide any recognizable patterns, and they were scaled to have zero width so that the entire object would turn on or off at the same time. The tendrils were to flash on very bright at first, then stay on for 8 to 20 frames and fade off. This variation in intensity was painted into the bar code texture and the tendril's

's' coordinate was then animated over the texture width to determine its intensity. The tendrils had to be rendered with the ribbon core as a matte object to hide tendrils that were behind the displaced surface of wing.

Plasma Particles

To create the rising gaseous plasma that came off the core and dissipated quickly we decided to use particle simulation. This element would be mostly small in frame and would be perpendicular to the view not requiring a simultaneous change of scale in the scene. The ribbon core was exported into Dynamation and its movements and velocities used to emit particles that would inherit its surface velocities and erupt at locations with a lot of sudden movement. The particles were then treated in the digital compositing stage to have a gaseous look. For the close up energy ribbon shots this effect was created using multiple sheets of geometry textured with fractal shaders combined with texture maps to create an illusion of thick atmospheric depth.

Compositing

The final stage of incorporating all the elements turned out to have a life of its own when it came to creating flashes of light within the core that would actually light up entire sections of the ribbon. Two dimensional disks were animated to zigzag up and down the frame crossing the ribbon core along their path. Intensities higher than a certain level were automatically amplified to give bright white flashes of light. Afterwards the actual 3-D elements of the core and the sea foam were then lit with intense spot lights that were positioned to go off at the right time and simulate interactive light raking the surface and bringing it to life.

Enterprise 1701-B Excelsior Class

The most critical issue in using two different models is that of intercutting. Each must be able to convincingly substitute for the other. In this case one of the models belonged to the real world and the other was virtual. The digital model was built to match the stage model, then textured using a combination of photographed image projections from the real model and digital paint work.

One of the most important features of the surface quality of the real model was its specular textures. In order to preserve accuracy, the same frisket masks that were used to paint the specular patches on the model were scanned and used to paint digital specular maps for the CG model. This was a key element to show the scale of the ship. Bump maps were used very scarcely only to break up evenness in the model, but not for adding any groove or metal plate seam details because although some shots would get very close to the ship, at that scale everything was very smooth on the stage model. All the shots involving the energy ribbon and the Enterprise used the CG model to take advantage of the interactions of lighting and particle physics together with the massive scales involved.

The CG model consisted of 231 pieces of geometry created using Alias software, with 94 texture maps for color, bump, transparency, specular, and incandescent mapping in the 72 different materials. The detailing on the ship continued on into the production of several shots, each needing added detail from a different angle. It was easy to just keep working endlessly on detailing, but there is a point of diminishing returns. What soon became apparent was memory and disk space limitations because of the 244 megabytes of textures required that were being read over the network from the various rendering machines. We eventually had to make local copies of all the textures on those machines in order to speed up the render times. On average the shots involving the model at close range would take about 2.5 hours to render with motion blur. One of the shots ended up bringing the model close enough to the camera that if the real model was being filmed, the camera lens would have hit

the model. The CG model was scaled properly in relation to the energy ribbon to resolve any discrepancies.

Specific shaders were developed for some of the lights used to light the ship to simulate the rippling sparks and glows of the energy ribbon using noise functions to vary the intensity. Particles were created in Dynamation and charged with negative force fields and blown across an also negatively charged Enterprise model, causing the particles to avoid the ship and swirl around it. The Excelsior class Enterprise had a unique feature where the main fuselage connected to the saucer section via a neck piece that was divided into lots of thin fins. The real model had been made extremely well so as to make all the fins parallel and with the same thickness. But despite this, there were still some minor imperfections that were eventually put onto the CG ship's perfect array of fins by assigning a very subtle displacement to it just to add some warp to the pieces. A noise function was also used to create a procedural bump mapping that roughened the surface. In some instances where parts of the ship were lit on the real model by what seemed to be lights coming from the ship itself, such as the saucer light that delineates the 1701-B on the top and bottom of the disk, were actually being lit from points outside the ship using special cut outs to make the shapes. Some of these were re-created using the slide projection feature from lights, and others were added as incandescent maps for easier control and line-up.

Scene Integration

The importance of interactive lighting cannot be more emphasized. It is the one queue that ties otherwise separate elements together and creates an illusion of oneness. Scale change issues are things to watch for in the compositing stage. There was one shot of the energy ribbon where the camera followed the Enterprise as it closed in on the ribbon almost doubling in size. All the flashes of light from the ribbon core were then lit in 3-D to match the location, size and intensity, and also used to light the starship from the same sources. The scale change on the ribbon meant that all the 2-D compositing values that were being used to integrate and soften the elements had to be animated correctly. It is important to concentrate and emphasize the important aspects of each shot and not get bogged down by detail that would not be visible until the twentieth viewing.

The RenderMan Interviews

RenderMan Users in Their Own Words

Introduction

We surveyed several studios who use RenderMan regularly for high-end computer graphics animation production. We wanted to find out how RenderMan was used in each studio, so that everyone in the audience today (or reading the course notes later) would have a sense of how RenderMan fits into the whole of animation production, and how a little RenderMan knowledge will help them in the job market. We weren't able to survey every major studio, but we got as many as we could (under terrific time pressure just as these course notes were going to press). I think you will find that there are a lot of similar themes, but that each studio has its own special needs and its own special way of approaching the problems it has to solve.

At the time of these surveys, most of the studios had only used Pixar's PhotoRealistic RenderMan product on actual production work, so unless they specifically mention using BMRT or some other tool, you can assume that they are writing about it whenever they refer to "RenderMan".

Notes on presentation: I was going to compile all answers to each question together, to make it look like there was sort of a round-table discussion going on. However, there really wasn't, and after reading through the answers, I decided that each one sounded more like an individual interview than a round-table (not surprising, since it was!). It made more sense to hear each story as a coherent whole rather than try to counterpoint each little bit of each interview together, so that's what I did. Items inside square brackets [like this] are comments added by the editor [Tony Apodaca] to elaborate or explain items that might not be clear to all readers.

Bill Reeves, Technical Director, Pixar Animation Studios

Pixar specializes in 3D character animation. Our work has been seen in several popular short films, many television commercials, and we are currently producing the world's first full-length computer animated feature film, *Toy Story*. Bill is the Technical Director (in the Hollywood film credit sense) of that film.

RenderMan Renderer

How long have you used RenderMan? Which implementations?

We first used RenderMan on Tin Toy in 1987. That was the Transputer version. Since then we have used it on CCI, SGI, Sun, NeXT platforms. We always have used the high-end version -- never used MacRenderMan or clones.

Of course, we used Reyes [the precursor program upon which the current PhotoRealistic RenderMan was based] before that, starting in 1981.

On which films (or other projects) have you used RenderMan?

Tin Toy, *KnickKnack*, numerous commercials, *Toy Story*.

In your opinion, what does RenderMan do well?

The shading language, motion blur, filtering, texturing.

What does it not do, that you need it to do?

It is slow, shadows are problematic. We have the occasional need for ray tracing, and the occasional need for radiosity.

How do you work around things that RenderMan can't do?

We buy lots of machines because it's slow, write lots of shader and source code hacks for shadows. We use reflection maps for ray tracing, and dirt maps for radiosity.

What's your average rendering time for a typical shot?

3 hours for a 1536x922 image at 4-by-4 PixelSamples.

Do you also use other renderers besides RenderMan?

Nope [need we ask?]

RenderMan Savvy Tools

What front-end tools do you use to drive RenderMan?

Menv [Pixar's proprietary modelling and animation environment. The system is based a procedural modeling language which is compiled into RIB just moments before rendering.]

How much tweaking is necessary to get "good RIB" from those tools into RenderMan?

Lots of sed/awk/perl massaging of RIB files, but this is mainly due to managing separate elements (reflections, shadows, etc.) and not due to "bad RIB".

What back-end tools do you use on RenderMan images?

Comet [Pixar's in-house compositing and image processing system], and the Amazon paint system.

RenderMan in Your Projects

What's the best (in your opinion) effect you've done with RenderMan?

All of *Toy Story*.

With the Shading Language?

There are many to pick from. [Pixar tends to use complex custom shaders extensively to generate the look of its images, rather than rely on complex geometry, so nearly all the good stuff is special shaders in one way or another.]

What's the most invisible effect that you've done with RenderMan?

Everything we do is visible.

How much SL programming do you do on average shots?

Lots!

How much custom tool writing?

Not much.

RenderMan in Summary

How much work is saved because you use RenderMan for rendering?

I wouldn't want to paint those images.

How much extra work do you incur because you use RenderMan?

Because it's slow, iteration is hard and time consuming.

What's your motivation for using RenderMan? (e.g. cost, quality, breadth, speed, programmability, robustness, standardization, etc.)

Quality, breadth, programmability.

Employment

How many technical graphics people does your group employ?

30 in *Toy Story* production, 8 more in Pixar Shorts

What percentage are "RenderMan savvy"?

80%

What percentage are expert RenderMan shader writers?

50%

Do you often have job openings for RenderMan experts? Currently?

Yes, yes.

What experience/knowledge do you need from your job candidates?

A Masters degree in CG is about ideal for fresh out of college. You don't really need RenderMan experience - you can pick it up.

Tom Williams, Executive in Charge of Digital Production, Industrial Light & Magic

ILM has been a full-service motion picture special effects company since 1978, when it did its first work on *The Empire Strikes Back*. Since then it has won 14 Academy Awards for Visual Effects, including 5 for films that included significant 3D computer graphics effects.

RenderMan Renderer

How long have you used RenderMan? Which implementations?

Since I was born. No, not exactly. Personally, I used ChapReyes [a Pixar Image Computer version of the Reyes rendering algorithm] about 10 years ago [It couldn't have been more than 8, but who's counting!]. I started using RenderMan as soon as it was commercially available.

On which films (or other projects) have you used RenderMan?

Oh jeez, I don't have the whole list handy, but you can safely assume that RenderMan was used on every ILM film production since *The Abyss* and certainly ever since I started on *Terminator 2*.

In your opinion, what does RenderMan do well?

Super. It's the best.

What does it not do, that you need it to do?

It doesn't do ray tracing well, it doesn't do area light sources, it doesn't handle shadows very well, and it's not fast for very simple renders. Unfortunately, most people do comparisons of RenderMan vs. rather simple rendering engines. With complicated shaders and if you know how to tweak some of the performance parameters it does quite well with time and memory usage comparisons.

How do you work around things that RenderMan can't do?

You can force RenderMan to do a lot more than you think, but for some types of rendering or

effects, we will write our own renderers or compositing hacks to create the effect we're looking for.

What's your average rendering time for a typical shot?

1 hour at film resolution. It can vary greatly as most users know.

Do you also use other renderers besides RenderMan? If so, how do you choose what to use for a particular show?

Well, as I mentioned, most of the other renderers we use are home-grown, so usually we know which renderer is best for which type of effect. We also use the built-in SoftImage renderer or SGI hardware for quick motion tests.

RenderMan Savvy Tools

What front-end tools do you use to drive RenderMan?

We write our own bridges between our internal file format and other tools such as SoftImage or Alias. Our formats understand how to output RIB and how to give users access to the important RIB parameters before rendering.

What back-end tools do you use on RenderMan images?

Again, we write a lot of our own software, so most of the front end processes and back end compositing is proprietary software.

RenderMan in Your Projects

What's the best (in your opinion) effect you've done with RenderMan?

That's a hard question. Certainly some of the projects we are currently working on have better effects than ever before but, I really like *Jurassic*, personally, for the sheer shock it gave me that the artists on the project could take these crude tools and hardware and create the believable illusion of life.

What's the most invisible effect that you've done with RenderMan?

A lot of the birds flying through scenes in *Gump* were computer generated, but the *Jurassic* jeep, which the Trex mangles (with the kids inside) was completely CG. That was great because it was right their staring you in the face and I couldn't tell the difference (unless you look at the hubcaps!).

How much SL programming do you do on average shots?

A lot. Depends on the project.

How much custom tool writing?

We have a pretty amazing staff of programmers that do this full time, so I guess you could say we do a lot of it.

RenderMan in Summary

How much work is saved because you use RenderMan for rendering?

I think it's more of a question of how much work we can do, that we wouldn't of been able to do without RenderMan. I would think that, safely, you could say that without the history of RenderMan and RenderMan images [i.e. the marketability of their services due to the look they can achieve], not to mention the tool itself, we would be doing at least 2 less film projects a year.

What's your motivation for using RenderMan? (e.g. cost, quality, breadth, speed, programmability, robustness, standardization, etc.)

Cost, reliability, speed, efficiency, flexibility. Not necessarily in that order.

Employment

How many technical graphics people does your group employ?

Depends on how you define technical graphics people. I'd guess about 110.

What percentage are "RenderMan savvy"?

Depends, again, on what you mean by savvy. 40%??

What percentage are expert RenderMan shader writers?

10%?????

Do you often have job openings for RenderMan experts? Currently?

Yes, and Yes.

What experience/knowledge do you need from your job candidates?

It helps to be from the same primordial gene pool as Darwyn Peachey. Ummmmm, seriously, from the RenderMan user perspective you have just got to have produced moving images with a computer before we could qualify someone. There are a lot of really good people out there that haven't but, it's really difficult to put them on a project unless they have pervious examples.

Jim Rygiel, Visual Effects Supervisor, Boss Film Studio

Boss Films is a full-service special effects studio which produces traditional and computer generated special effects for feature films and television commercials. Boss's boss, Richard Edlund, has received 6 Oscars for Best Visual Effects, including *2010*, the first Oscar for a film with significant CG.

RenderMan Renderers

How long have you used RenderMan? Which implementations

Approximately 4 years, primarily for rendering out quality film-like images, relying on its motion blur and depth-of-field principles, and also for the quality of the render.

On which films (or other projects) have you used RenderMan?

Batman Returns, Cliffhanger, Aliens III, Outbreak, Drop Zone, Species, Waterworld

In your opinion, what does RenderMan do well?

The quality of the render has a visual appearance unlike any other render. The detail, color and visual feel is outstanding.

What does it not do, that you need it to do?

Shadows are somewhat unruly, although when they work, they do work well.

How do you work around things that RenderMan can't do?

We use a full compliment of packages, everything from Wavefront, SoftImage, Alias to Elastic Reality and Amazon Paint. Our motto is "Wherever possible, cheat!"

What's your average rendering time for a typical shot?

20 minutes for a 2K image, with image shading rate of 1 and sampling anywhere from 3 to 6.

Do you also use other renderers besides RenderMan? If so, how do you choose what to use for a particular show?

Yes. For instance, it was much easier for us to render shadows in Wavefront, and the object in RenderMan, and do a simple over composite. Also, in cases where we might not need motion blur or depth-of-field, using another render might be advantageous. Alias has some nice built-in lighting effects, for instance.

RenderMan Savvy Tools

What front-end tools do you use to drive RenderMan?

WaveMan we rely on heavily. We do all of our setup in Wavefront, preview, lighting, etc., and then port it thru WaveMan.

How much tweaking is necessary to get “good RIB” from those tools into RenderMan?

WaveMan has been very very good to us. It seems to do somewhat near the correct thing 95% of the time.

What back-end tools do you use on RenderMan images?

Wavefront Composer is usually the last step for us. All of our images end up in composite mode in here.

RenderMan in Your Projects

What’s the best (in your opinion) effect you’ve done with RenderMan?

Its the standard effects answer! Anytime we render things in RenderMan and to ourselves the look is invisible, we consider it our best effect and compliment.

With the Shading Language?

Poly normal renderings, which can be useful in a 2-D situation for strange image processing effects

What’s the most invisible effect that you’ve done with RenderMan?

Rendering helicopters for *Outbreak* and a test on a sailing ship, where when shown to Directors and Producers, they have to ask us repeatedly what did we do in the scene. Even after we explain it to them, they seem a little befuddled.

How much SL programming do you do on average shots?

Probably for each show we might write on an average of 20 shaders.

How much custom tool writing?

About 25% of our tools on a show might be custom.

RenderMan in Summary

How much work is saved because you use RenderMan for rendering?

We save more in the rendering mode, where we don’t have to think about the subtleties of motion blur or depth-of-field, but where we can literally turn in on and be somewhat assured that it will work and act correctly. Also in the lighting and rendering quality.

How much extra work do you incur because you use RenderMan?

The RenderMan RIB file can become a monster, sometimes growing larger than the actual image it will render out. This becomes a problem in disk space and I/O time.

What’s your motivation for using RenderMan? (e.g. cost, quality, breadth, speed, programmability, robustness, standardization, etc.)

The cost is certainly an issue. And of course, this because relative with the quality of the prod-

uct. Without giving you guys a big head and then having you jack up the price, I think it is well worth itself many times over.

Employment

How many technical graphics people does your group employ?

60

What percentage are “RenderMan savvy”?

25

What percentage are expert RenderMan shader writers?

5

Do you often have job openings for RenderMan experts? Currently?

We are always looking for those people that can push our effects onto that ever-moving next level.

What experience/knowledge do you need from your job candidates?

The ability to prove your capabilities, in either the form of video tapes or published papers, although production experience is always the first plus mark in our evaluation.

Antoine Durr, Senior Technical Director, VIFX

VIFX is a full-service visual effects company. Although our main bent is computer generated effects, we also do traditional effects work. I do a bit of everything. Sometimes I animate, sometimes I program, ideally a bit of both. Lately I’ve been designing effects using a combination of PRISMS and RenderMan.

RenderMan Renderer

How long have you used RenderMan? Which implementations?

2 years, pretty full time. Exclusively PRMan. I have toyed with BMRT, but that’s about it.

On which films (or other projects) have you used RenderMan?

Time Cop, Ghost In The Machine, Mighty Morphin’ Power Rangers - The Movie, Outbreak

In your opinion, what does RenderMan do well?

The reason we use it is because of the complete flexibility. It can create fantastic looks very easily. It’s motion blur facilities, although not perfect, are much more efficient than that of any other renderer I’ve encountered. Besides motion blur, one of the other main reasons to use it is the displacement capability.

What does it not do, that you need it to do?

Our biggest problem comes from trying to send stuff through the camera plane. We’ve gotten nailed with that one on just about every show we’ve done. When you have a “U” shaped object, and you try to fly the camera into it, you have to be very careful.

How do you work around things that RenderMan can’t do?

Fudge it. Break apart the models. Render with more layers, so that you have less data in each one.

What’s your average rendering time for a typical shot?

At NTSC res, most shots [a few seconds of animation] are renderable overnight, usually on a

single CPU. Sometimes more, sometimes less. I try to keep render times around a half hour total per output frame, i.e. sum of all layers.

Do you also use other renderers besides RenderMan? If so, how do you choose what to use for a particular show?

You bet. I've intermixed them in a single shot. PRISMS, our main animation tool, comes with a ray tracer (Mantra) and a scan-line renderer (Crystal2). The scan-line renderer is wicked fast, though the quality is not the best. The ray tracer makes creating cast shadows a snap.

RenderMan Savvy Tools

What front-end tools do you use to drive RenderMan?

PRISMS, mostly, sometimes Alias. We also have a couple of in-house tools that can spit out RIB.

How much tweaking is necessary to get "good RIB" from those tools into RenderMan?

From PRISMS, virtually never. There used to be some stuff that we changed via 'sed' on the fly. We never hand-edit RIB files, except occasionally when trying to determine errors, or prototyping an effect. RenderMan is becoming like PostScript - it's somewhere under there, but you don't have to touch it, although understanding it is very valuable.

What back-end tools do you use on RenderMan images?

Mostly compositing, occasionally color correction or contrast adjustment. Noise [e.g. film grain] sometimes has to be added.

RenderMan in Your Projects

What's the best (in your opinion) effect you've done with RenderMan?

Few effects are strictly RenderMan. They are usually a combination of modelling, animation, rendering, and compositing/image processing.

With the Shading Language?

The scanning electron microscope look for *Outbreak*. The whole scene used no lights -- it computed brightness depending on normals. Massive displacement with various kinds of noise functions made the surface look very realistic.

What's the most invisible effect that you've done with RenderMan?

That same effect - the scene never made it into the movie. I'd say that made it pretty invisible.

How much SL programming do you do on average shots?

Most of the shader writing is usually done at the beginning of the show. Then, during the course of the various shots, the shaders get tweaked, creating theme-and-variations. In total time, I might spend upwards of a couple of weeks just writing shaders, for a multi-month show.

How much custom tool writing?

I try to write things in such a way that they can be reused. Usually, though, we need a bunch of text filters that are show-specific.

RenderMan in Summary

How much work is saved because you use RenderMan for rendering?

If you now can create a look whereas before you couldn't, you've saved an infinite amount of time. If it lets me create a look with relative ease, then I save labor.

How much extra work do you incur because you use RenderMan?

Doing the basics is a bit of a pain. It seems that for every show we end up writing yet another texture-mapping shader.

What's your motivation for using RenderMan? (e.g. cost, quality, breadth, speed, programmability, robustness, standardization, etc.)

Quality, motion blur, flexibility. Film work almost always requires you to motion blur, so that you match the look of the background.

Employment

How many technical graphics people does your group employ?

About two dozen.

What percentage are "RenderMan savvy"?

Everyone who animates at VIFX sooner or later uses RenderMan. That doesn't mean that everyone can take a RIB file apart and know what's going on. I'd say 60-70% understand a good portion of the RIB files they generate.

What percentage are expert RenderMan shader writers?

Less than 10% are expert shader writers. Another 20% are pretty good shader writers.

Do you often have job openings for RenderMan experts? Currently?

RenderMan knowledge is definitely an asset, but not everything. Because everyone does a bit of everything, there is much more that needs to be known than just RenderMan.

What experience/knowledge do you need from your job candidates?

An understanding of how computer graphics and animation work is helpful. Production experience. Half of a computer animator's title is "computer", and the better they know this half, the more efficiently they can do the work.

Ron Moreland, Technical Director, Santa Barbara Studios

Santa Barbara Studios's focus is on high quality, realistic, or technically accurate computer imagery for education and entertainment. My primary responsibility is solving technical issues involved in generating and compositing CG images.

RenderMan Renderers

How long have you used RenderMan? Which implementations?

I have used RenderMan for almost 4 years, we currently use RenderMan v3.4e.

On which films (or other projects) have you used RenderMan?

I have used RenderMan on several films and television shows. Films: *Death Becomes Her* [in a prior life at ILM], *Star Trek: Generations*, *Cosmic Voyage* [an Omnimax movie currently in production for the Smithsonian Institute]; television: *500 Nations*, *Star Trek Voyager* (Opening Credits and several shows).

In your opinion, what does RenderMan do well?

I think RenderMan's strengths are its stability, meaning it has no or few flickerly artifacts (other than those introduced with bad shaders), it has no killer bugs, the shading language is very, very flexible, it has superb filtering capabilities. Its image out almost always "just looks great".

What does it not do, that you need it to do?

It doesn't ray trace, it has no soft shadows (area lights), it has no "easy to use" user interface. It has poor communication between components (i.e. lights shaders to surface shaders) [Fixed in v3.5, Ron! Get with the program! ;-)] A shader has little knowledge of the object it's shading other than the current point. No volume rendering. We could use 4-D Perlin noise.

How do you work around things that RenderMan can't do?

I often combine the output of other renderers with RenderMan for special reflections or effects. I hacked our Wavefront environment to work with RenderMan to provide a UI for RenderMan. I hide special data in texture-maps or user-defined variables to enhance a shaders information about the object being shaded.

What's your average rendering time for a typical shot?

This is a difficult question. I find 1hr/frame acceptable for most of our work, but this varies with shot content and length.

At about what resolution, sampling rate and shading rate?

Our film work is typically 2k horizontal resolution, our IMAX resolution is 4k and, video res is .72k. Sampling rate 2x2, shading rate 1.

Do you also use other renderers besides RenderMan? If so, how do you choose what to use for a particular show?

We use Wavefront Image for fast raytracing, occasionally for special shadows, and Dynamation for clouds and smoke. We have an in-house volume renderer for special cloud effects.

RenderMan Savvy Tools

What front-end tools do you use to drive RenderMan?

Proprietary Wavefront-to-RIB converters and lots of shell scripts.

How much tweaking is necessary to get "good RIB" from those tools into RenderMan?

Lots....

What back-end tools do you use on RenderMan images?

Wavefront composer, in-house image processing tools.

RenderMan in Your Projects

What's the best (in your opinion) effect you've done with RenderMan?

I don't write many "effect" shaders. I just wrote one for an exploding sun that's interesting.

With the Shading Language?

I like the shaders I wrote for 500 Nations the best, they made good use of surface and light.

What's the most invisible effect that you've done with RenderMan?

Probably the twisted neck on Meryl Streep for *Death Becomes Her* [this was when Ron worked at ILM].

How much SL programming do you do on average shots?

I usually modify 2 to 3 shaders per show.

How much custom tool writing?

I wrote a converter for Wavefront-to-RIB and we have several surface-to-RIB tools.

RenderMan in Summary

How much work is saved because you use RenderMan for rendering?

That's hard to say because we usually chose RenderMan for shots that have to look great (all of them) or that can't be done any other way.

How much extra work do you incur because you use RenderMan

We probably have a 15-25% efficiency hit because RenderMan is not integrated into our normal tools.

What's your motivation for using RenderMan? (e.g. cost, quality, breadth, speed, programmability, robustness, standardization, etc.)

Quality, breadth, programmability, robustness.

Employment

How many technical graphics people does your group employ?

3

What percentage are "RenderMan savvy"?

33% (HELP!)

What percentage are expert RenderMan shader writers?

33% (HELP!)

Do you often have job openings for RenderMan experts? Currently?

We are looking for shader freelancers.

What experience/knowledge do you need from your job candidates?

For shader writers, 1-2 years of steady work writing a variety of shaders.