SIGGRAPH 2006
Course 25

RenderMan For Everyone

course organizer:

Rudy Cortes - *Walt Disney Feature Animation*

lecturers:

Hal Bertram

Tal Lancaster - *Walt Disney Feature Animation*

Dan Maas - *Maas Digital*

Moritz Moeller - *Rising Sun Pictures*

Heather Pritchett - *Walt Disney Feature Animation*

Saty Raghavachary - *Dreamworks Animation*

August 1st 2006

Course Information

When most 3D artists hear the word RenderMan they immediately think of the render that Pixar uses for its movies which is the same as the product that they sell commercially. This product is actually named Photo Realistic RenderMan or PRMan for short. The term RenderMan (note the capital M in man) refers to a 3d scene description standard or specification which was created and is still maintained by Pixar. This course will show that there have been and are other alternate RenderMan compliant renderers available. While PRMan has been for more than a decade the CG defacto standard renderer for film production, these other renderers might prove viable to smaller projects that can't afford PRMan.

Desired Background

This course is designed to allow everyone who is interested in learning RenderMan to get introduced to the specification or standard. Experience in programming or shading is not necessary, at least not for the first parts, however we do assume our attendees will have an advanced knowledge of 3d graphics and rendering concepts.

Suggested Reading Material

This course will start at a very basic level but wont go step by step into advanced levels. For those who are interested in covering the gaps here is a list of recommended books.

- **Essential RenderMan Fast** - Ian Stephenson - ISBN: 1852336080 - Publisher: Springer; 1 edition (January 31, 2003)
  This is small book that will get you up and running really quick in RenderMan. It does a lot of work on the RenderMan API and RIB. It also goes into shading. This is a good starting point for those new to RenderMan.

- **Rendering for Beginners** - Saty Raghavachary - ISBN: 0240519353 - Publisher: Focal Press (December 13, 2004)
  Another introductory book. This one goes very deep into all the different options, levers and knobs available on RenderMan renderers. Its full color with lots of images and explanatory graphics.

- **Advanced RenderMan : Creating CGI for Motion Pictures** - Anthony A. Apodaca & Larry Gritz - ISBN: 1558606181 - Publisher: Morgan Kaufmann; 1st edition (December 8, 1999)
  This is a more advanced book and a must have for anyone who decides to become a serious RenderMan

user. It covers in depth virtually every topic of producing images for film with RenderMan. Filled with lots of code and examples.

- **The RenderMan Companion** - Steve Upstill

  An oldie but a goodie. This book might be a little outdated, but it does cover a lot of material not included in any other RenderMan book.

Structure of this Course

This course is designed for people of different levels of knowledge. Its broken down into 3 segments. The first part of the course (the early morning) will be an introduction to RenderMan covering the basics of the standard and what can be expected while learning and using RenderMan. The middle of the course is an in-depth look into the standard. Covering the new additions and how you can create tools to use and manipulate data used for RenderMan. The last part of the course will be on tips and tricks used by major studios to bring us such beautiful images on film.

We will focus heavily on the RISpec standard, with special attention on the other available renderers (not PRMan) and how they can be used to create high quality images.

Acknowledgments

We would like to thank ... for their help and support. Our respective studios, Walt Disney Feature Animation, Dreamworks Animation, Rising Sun Pictures, for allowing us to pass the knowledge to the rest of the industry. We would also like to thank our technical editors...Also the Siggraph Course chair and board for accepting our course and allowing us to bring RenderMan back to Siggraph and finally to all the RenderMan professionals, enthusiasts and students of the world, for taking the interest and enduring the steep learning curve, always looking for new ways of creating beautiful images.

Lecturers

Rudy Cortes - Course Organizer

Is a Look Dev TD currently working on Glen Keane's **Rapunzel** at Walt Disney Feature Animation. Before that he was a shader TD at DNA Productions in Dallas, TX and a shader and lighting TD at The Orphanage in San Francisco. He has been an avid RenderMan user and self proclaimed RenderMan evangelist for the last four years. In 2003 he founded The RenderMan Academy, a free online learning resource for RenderMan. His film credits include The Day After Tomorrow, Sky Captain and the World of Tomorrow, The Ant Bully (2006), Meet The Robinsons (2007) and Rapunzel (2009).

Hal Bertram

Hal Bertram joined Jim Henson's Creature Shop in 1992, working on animatronic puppeteering technology. In 1993 he set up the Creature Shop's CG department, developing real-time CG character performance systems used in numerous film, TV and commercial projects. Since 2003 he has been working as a consultant on rendering issues, writing software for films including *Tomb Raider: The Cradle of Life*, *Troy* and *Charlie and the Chocolate Factory*.

Tal Lancaster - Walt Disney Feature Animation

Tal Lancaster is a senior technical director forWalt Disney Feature Animation, where he has been working since 1995. For the past three years there, he has been the main shader architect for the development of RenderMan shaders. His film credits include *Fantasia 2000*, *Dinosaurs*, *Chicken Little*, and *Meet the Robinsons* (2007). Other projects include *Mickey 3D*, and the theme park attraction *Mickey's Philharmagic*. Tal also is the maintainer of the “RenderMan Repository”: www.renderman.org, since its creation in 1995.

Dan Maas - Maas Digital

Is the founder of Maas Digital, a New York based animation studio specializing in animation for the aerospace industry. Dan created a custom RenderMan-based pipeline for rendering highly realistic animation of NASA's Mars Rover mission and other projects. Dan recently received an Emmy nomination for his work on *Mars, Dead or Alive*, a PBS/NOVA program, and his first IMAX feature film project, *Roving Mars*, premiered in 2006.

Moritz Moeller - Rising Sun Pictures

Moritz Moeller's is a Technical Director at Rising Sun Pictures. Before he worked as a Rendering Supervisor on

an Italian full CG feature film in India and helped designing the rendering pipeline for Norway's first full CG feature film, "Free Jimmy". Before that he worked on numerous commercials across Germany and Europe and a long time ago also in games. Moritz has been using RenderMan compliant renderers since 1996. He also is one of the administrators of Liquid, the open source Maya to RenderMan plug-in and developed Affogato, the open-source XSI to RenderMan plug-in for Rising Sun Pictures. His film credits include "Free Jimmy" (2006) and "Charlotte's Web" (2006) Moritz is a part time lecturer at the German Film School, Germany, where he teaches RenderMan..

Heather Pritchett - Walt Disney Feature Animation

Heather has been with Walt Disney Feature Animation since 1994 and a RenderMan user since 1992. While at Disney she's worked on a variety of projects, including park, shorts, live action films and various animated Features from Hercules to Chicken Little. She is currently the Look Development Lead on the 2007 release "Meet the Robinsons".

Saty Raghavachary - DreamWorks Feature Animation

Is a senior graphics software developer at Dreamworks Feature Animation which he joined in 1996. He has written software used in *The Prince of Egypt, The Road to El Dorado, Spirit:Stallion of the Cimarron, Sinbad: Legend of the Seven Seas, Shark Tale* and *Over The Hedge*. He is also the author of *Rendering for Beginners, Image synthesis using RenderMan* and has been using RenderMan since 1991 when he was at Ohio State. He is a part-time instructor at the Gnomon School of Visual Effects, USA where he teaches RenderMan and MEL programming, and at USC where he teaches CS480, an introductory course on 3D graphics.

Course Schedule

8:30	**Welcome and Introduction** Rudy Cortes - WDFA
8:45	**A Brief Introduction To RenderMan** Saty Raghavachary - Dreamworks Animation
9:45	**What The RenderMan Spec Never Told You** Dan Maas - Maas Digital
10:45	**Morning Break**
11:00	**Going Mad With Magic Lights** Moritz Moeller
12:00	**Lunch Time**
1:30	**Changes To RenderMan: PRMan 13 Preview** Tal Lancaster - WDFA
2:30	**RenderMan In Production at WDFA - Part 1** Heather Pritchett - WDFA
3:15	**Afternoon Break**
3:30	**RenderMan In Production at WDFA - Part 2** Tal Lancaster
4:15	**Production Rendering Acceleration Techniques** Hal Bertram
5:20	**Q & A Session - Wrap up and Make Up Time**
5:30	**Party Time - RenderMan Users BOF**

Table of Contents

Introduction

Rudy Cortes - Walt Disney Feature Animation

Welcome

Welcome to **RenderMan for Everyone** and to Siggraph 2006 in beautiful Boston, MA. This is the first RenderMan course offered at Siggraph since 2003, but with the leaps and bounds that technology takes it feels like a lot longer than that. We have a panel of very experienced RenderMan users, developers and lecturers that have worked really hard to put together the best course possible for our attendees. This course is designed to help introduce new TDs to the wonderful world of RenderMan while presenting new techniques to the more experienced TDs. We are also hoping to emphasize the fact that on this day and age, you don't need a large budget to purchase software for learning RenderMan, in fact we will present information on many apps that are very affordable or even open source. Anyone who wants to learn RenderMan can learn it, therefore the name of our course.

Why This Course?

Every year, hundreds of TD's flock to Siggraph to acquire knowledge on the latest techniques, discoveries, industry contacts, some free gifts and free drinks at one of the many parties. However for the last 2 years, RenderMan geeks (yes, we are all geeks, the sooner we accept that, the better off we will all be) all over the world have been at a loss because no courses have been presented on the use of RenderMan. Working in production is a lot of hard work as is organizing a Siggraph course. Having gone through the experience myself I

now clearly understand how 2 years could pass by so fast without a course being presented. It might not seem like a lot of time, but as you all know in the CG industry 2 years represent a lot of advances in techniques and software features. RenderMan is no exception to such trend, in fact the RenderMan spec has changed dramatically within the last couple of years, specially since the introduction of raytracing into the spec. Most of these features are briefly described in Pixar's RISpec and in their Application Notes. Neither the handful of available books on RenderMan nor the notes from all previous Siggraph RenderMan courses cover any of these new features. Add to that the gap in RenderMan courses at Siggraph and we end up with a lot of TD's who are just starving for information.

Who Is This Course For

This course is intended for TDs of all levels who are interested in either diving into RenderMan or expand their bag of tricks for rendering. We assume our attendees have a solid understanding of 3D graphics, how 3D renderers work, and the properties of 3D objects like UV spaces, surface normals and material properties. We also assume that our readers are not intimidated or perplexed by code or low level math. Programming experience is not necessary, but it does help. Even if it is a high level language or a 3d application scripting language such as MEL or MaxScript. The course is broken down into two segments. The morning segment will be an introduction to RenderMan in which we will cover the basics and some very necessary tricks to know, as well as those little caveats that are not described in the RISpec but every TD should be aware of. On the afternoon we will tackle more complex tricks that will hopefully expand your horizons and put things in perspective of what is possible with a RenderMan renderer.

Our Panelists

As previously mentioned we have put together a panel of industry professionals who will share with you a lot of insight into the use or RenderMan. We have people with different backgrounds on our panel, such as Heather Pritchett and Tal Lancaster from Walt Disney Feature Animation. Heather is the Look Development Lead for the upcoming Walt Disney Feature Animation film Meet The Robinsons and Tal is a senior technical director and one of the main architects of WDFA's shading development environment. Also from the feature animation field we have Saty Raghavachary who is a senior software engineer at Dreamworks Animation. Saty doesn't use RenderMan professionally at Dreamworks anymore (they have their own proprietary renderer) but he is still very involved with RenderMan as he teaches it at Gnomon and USC and is also authoring a book on writing RenderMan shaders. From the visual effects industry we have Moritz Moeller who is a Technical Director at Rising Sun Pictures and is one of the administrators of Liquid, the open source Maya to RenderMan plug-in. He also developed Affogato, the open-source XSI to RenderMan plug-in for Rising Sun Pictures. We also have

people like Dan Mass who doesn't work on the feature animation or visual effects industry, but still pushes RenderMan technology to the edge on generating animations for NASA such as the Mars Lunar Rover. We also have a rendering consultant, which I must say sounds like such a cool gig to have. His name is Hal Bertram and he has worked with many companies and became quite noticed at last years Stupid RenderMan Tricks when he presented his interaction trick. I have been to several of these SRT before and I have never heard so many people gasp in silence before erupting into applause and cheers. He will be presenting extra information on his interaction technique.

A Brief Introduction To RenderMan

Saty Raghavachary - Dreamworks Animation

Origins

Pixar's RenderMan software has its origins in the University of Utah during the 1970s, where Pixar founder Ed Catmull did his PhD work on rendering problems. From there, the scene shifted to George Lucas' Lucasfilm in California, where Catmull and few other graphics researchers were brought in to work on graphics software specifically for use in motion pictures (a part of Lucasfilm later became Pixar).

The researchers had the explicit goal of being able to create complex, high quality photorealistic imagery, which were by definition virtually indistinguishable from filmed live action images. They began to create a renderer to help them achieve this audacious goal. The renderer had an innovative architecture designed from scratch, incorporating technical knowledge gained from past research both at Utah and NYIT. Loren Carpenter implemented core pieces of the rendering system, and Rob Cook wrote the shading subsystem. Pat Hanrahan served as the lead architect for the entire project. The rendering algorithm was termed “REYES”, a name with dual origins. It was inspired by Point Reyes, a picturesque spot on the California coastline which Carpenter loved to visit. To the rendering team the name was also an acronym for “Render Everything You Ever Saw”, a convenient phrase to sum up their ambitious undertaking.

At the 1987 SIGGRAPH conference, Cook, Carpenter and Catmull presented a paper called “The Reyes Rendering Architecture” which explained how the renderer functioned. Later at the SIGGRAPH in 1990, the

shading language was presented in a paper titled “A Language for Shading and Lighting Calculations” by Hanrahan and Jim Lawson. In 1989 the software came to be known as RenderMan and began to be licensed to CG visual effects and animation companies. Also, the CG division of Lucasfilm was spun off into its own company, Pixar in 1983 and was purchased by Steve Jobs in 1986. The rest, as they say, is history..

Even though the public offering of RenderMan did not happen until 1989, the software was used internally at Lucasfilm/Pixar way before that, to create movie visual effects, animation shorts and television commercials.

In 1982, the Genesis Effect in the movie Star Trek II: The Wrath of Khan was created using an early version of RenderMan, as was the stained glass knight in the movie Young Sherlock Holmes released in 1985.

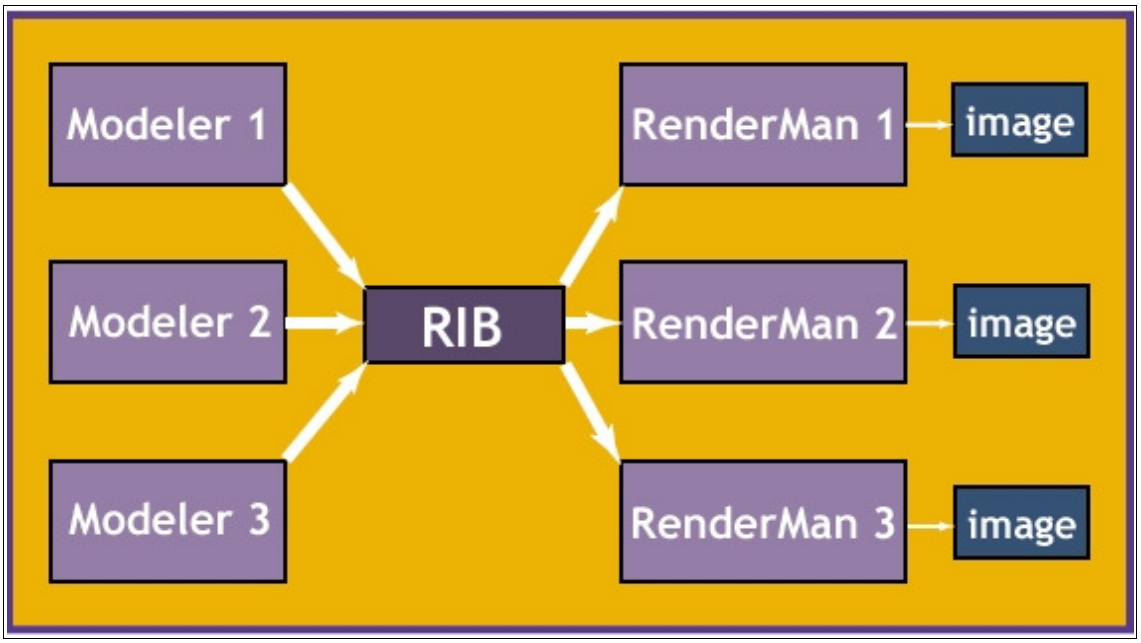
Today, leading animation and visual effects studios around the world routinely use Pixar's RenderMan thanks to its unsurpassed track record - it is fast, stable, efficient when it comes to handling large scenes with complex geometry, surface appearances and lighting. The output is high quality photoreal imagery, usable on its own (eg. in animated features) or ready for compositing with existing footage (eg. in live-action movies).

Spec

In the 'Origins' section above, we began by referring to 'Pixar's RenderMan'. This is because, strictly speaking, the word 'RenderMan' by itself denotes an interface description originated by Pixar, to provide a standard way for modeling/animation programs to communicate their scene descriptions to renderers. In other words, RenderMan is a formal specification. It is referred to as the 'RI Spec', where 'RI' stands for 'RenderMan Interface'. Pixar's own implementation of the specification was historically the very first one, so people loosely refer to it (the implementation) also as 'RenderMan'. The correct name for Pixar's version is 'PRMan' (short for Photorealistic RenderMan), and this is the name we will use for it from now on.

In a 3D graphics pipeline, rendering is the last step (after modeling, animation and lighting) that generates an image out of a scene description. Renderers are specialized, complex programs that embody a variety of algorithms which collectively lead to image synthesis. The RI Spec defines a clear separation (or boundary, or interface) between modeling and animation programs on one hand, and rendering programs on the other. The idea is that each side can focus on its own specialty, and formal 'handshake' protocol can lead to successful mixing and matching between the two. In practical terms, this means that if a piece of modeling/animation program were to output its scene description in an established format, that scene description should be able to serve as input to a variety of renderers that handle that format. All the renderers would produce pretty much the same output image from the scene description, regardless of how their internals are structured. This is because the interface specifies what to render (via geometry, lights, material and camera descriptions) but not how. The

'how' is up to the individual implementations to handle - they can freely employ scanline algorithms, ray-tracing, radiosity, point-based graphics or any other technique to render the output.



The Spec was authored by Pixar and was endorsed by leading graphics vendors at the time such as Sun, Apollo, Prime and NeXT. The hope was that the Spec would give rise to a variety of implementations. As a proof of concept and to seed the marketplace, Pixar themselves created PRMan, the first-ever RenderMan implementation. Shortly thereafter, Larry Gritz wrote his freeware BMRT which also adhered to the Spec. As a case in point, BMRT featured ray-tracing and radiosity which were only recently (in 2002) added to PRMan. RenderDotC from DotC Software was also an early implementation which continues to be sold to this day.

Fast-forwarding to more recent times, there have been several other RenderMan implementations since the early days. Exluna Corporation (founded by Larry Gritz and others) created and sold Entropy, a commercial version of BMRT (however, due to an unfortunate lawsuit, Exluna was shut down and Entropy/BMRT were taken off the market). Air, Aqsis, Angel, Pixie and 3Delight are contemporary implementations which should be of interest to attendees of this course. Together with RenderDotC, they provide alternatives to PRMan. While PRMan remains the industry's gold standard for RenderMan implementations, it also happens to be more expensive than the alternatives (many of which are free!).

Here is a brief tour of the Spec, which is divided into two parts. Part I, 'The RenderMan Interface', begins by listing the core capabilities (required features) that all RenderMan-compliant renderers need to provide, such as a

complete hierarchical graphics state, camera transformations, pixel filtering and antialiasing and the ability to do shading calculations via user-supplied shaders written in the RenderMan shading language. This is followed by a list of advanced/optional capabilities such as motion blur, depth of field and global illumination. The interface is then described in great detail, using procedural API calls in C/C++ and their corresponding RIB (RenderMan Interface Bytestream) equivalents. RIB can be regarded as a scene description format meant for use by modeling programs to generate data for RenderMan-compliant renderers. Part II of the Spec, 'The RenderMan Shading Language' (RSL), describes a C-like language (with a rich set of shading-related function calls) for writing custom shading and lighting programs called *shaders*. This programmable shading aspect is one of the things that makes RenderMan enormously popular, since it gives users total control over lighting and appearances of surfaces and volumes in their scenes.

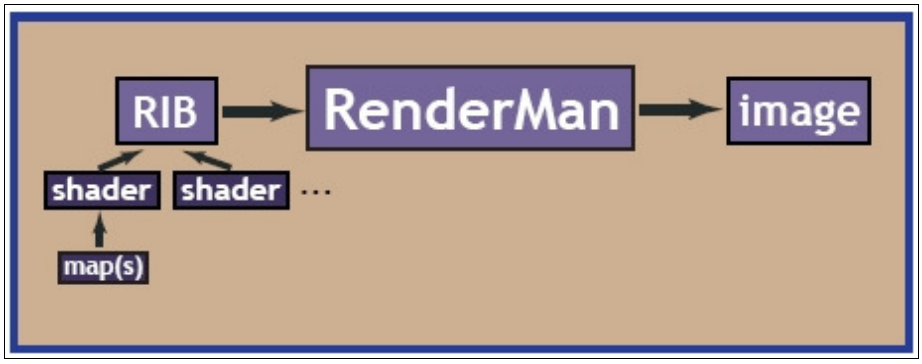
The latest version of the Spec is 3.2.1, revised in November 2005. You can find the 688K, 226 page document (which happens to make for enjoyable reading!) at Pixar's site: https://renderman.pixar.com/products/rispec/index.htm.

Be sure to get a good understanding of what is in the RI Spec - it will help you know what to expect in a typical RenderMan implementation (any renderer that calls itself 'RenderMan-compliant' will by definition be bound by the interface laid out in the Spec). In addition the Spec will serve as your reference for RIB and procedural API syntax and also for the large set of built-in functions of the RSL.

Pipeline (RIBS, shaders, maps)

In this section we will look at how you create RenderMan images in practice, using your renderer of choice (PRMan/RenderDotC/Air/Aqsis/Angel/Pixie/3Delight). While the overall pipeline is same for all these renderers, you will need to consult your particular renderer's manual for implementation-dependent details and minor syntax variations in RIB, etc.

Here is the overall flow of data to generate (render) output:



The renderer accepts a RIB file containing scene description that can be in ASCII or binary format, usually containing data for rendering just one frame. It reads the RIB file and renders a result image. The output image can either be displayed on a screen window or written to an image file, eg. in TIFF format. The RIB file will most likely contain references to shaders that describe shading, displacements or light sources. Such shader files are themselves external to the RIB file. Additionally, shaders can sometimes reference map files (eg. texture maps) which are in turn external to the shaders. The combination of a RIB file and its associated shaders and maps is what gets rendered into an output image.

The RIB file is input to the renderer via a simple command line invocation. Eg. in PRMan, the call would look like this:

```
render teapot.rib
```

'render' invokes PRMan, which will read 'teapot.rib' and render an output image according to the scene described in 'teapot.rib'. (In windows you need to use prman instead of render)

```
# teapot.rib
# Author: Scott Iverson <jsiverson@xxxxx.edu>
# Date: 6/7/95
#

Display "TeapotAfter.tif" "framebuffer" "rgb"
Format 600 400 1
Projection "perspective" "fov" 30
Translate 0 0 25
Rotate -22 1 0 0
Rotate 19 0 1 0
Translate 0 -3 0
WorldBegin
LightSource "ambientlight" 1 "intensity" 0.6
LightSource "distantlight" 2 "intensity" 0.6 "from" [-4 6 -7] "to" [0 0 0]
"lightcolor" [1.0 0.4 1.0]
LightSource "distantelight" 3 "intensity" 0.36 "from" [14 6 7] "to" [0 -2 0]
"lightcolor" [0.0 1.0 1.0]
Surface "plastic"
Color [1 0.6 1]
### Spout ###
AttributeBegin
Sides 2
Translate 3 1.3 0
Rotate 30 0 0 1
Rotate 90 0 1 0
Hyperboloid 1.2 0 0 0.4 0 5.7 360
```

```
AttributeEnd
### Handle ###
AttributeBegin
Translate -4.3 4.2 0
TransformBegin
Rotate 180 0 0 1
Torus 2.9 0.26 0 360 90
TransformEnd
TransformBegin
Translate -2.38 0 0
Rotate 90 0 0 1
Torus 0.52 0.26 0 360 90
TransformEnd
Translate -2.38 0.52 0
Rotate 90 0 1 0
Cylinder 0.26 0 3.3 360
AttributeEnd
### Body ###
AttributeBegin
Rotate -90 1 0 0
TransformBegin
Translate 0 0 1.7
Scale 1 1 1.05468457
Sphere 5 0 3.12897569 360
TransformEnd
TransformBegin
Translate 0 0 1.7
Scale 1 1 0.463713017
Sphere 5 -3.66606055 0 360
TransformEnd
AttributeEnd
### top ###
AttributeBegin
Rotate -90 1 0 0
Translate 0 0 5
AttributeBegin
Scale 1 1 0.2051282
Sphere 3.9 0 3.9 360
AttributeEnd
Translate 0 0 0.8
AttributeBegin
Orientation "rh"
Sides 2
Torus 0.75 0.45 90 180 360
AttributeEnd
Translate 0 0 0.675
Torus 0.75 0.225 -90 90 360
Disk 0.225 0.75 360
AttributeEnd
WorldEnd
```

Image 1: Output of teapot.rib

Let us now briefly look at some characteristics of RIB files, shaders and maps.

RIB files

Sources of RIBs include the following:

- executable programs created using the procedural API calls from the Spec. When such a program is run, its output will be RIB statements, which can be redirected to a file for submitting to the renderer. Eg. a DNA rendering program would contain RI calls to create spheres of various sizes and colors, placed at specific locations as dictated by the DNA molecular structure.
- translator plugins that are part of mainstream animation packages which create RIB descriptions of scenes. These plugins would make RI calls corresponding to scene elements. Eg. for Maya, MTOR, MayaMan and Liquid are all plugins that output RIB.
- converter programs that read scene descriptions for other renderers and create corresponding RIB calls. Eg. 'mi2rib' is a Mental Ray to RenderMan converter.
- several modeling/animation programs (eg. Blender) natively output RIB, ie. without a translator plugin.
- simple RIB files can be hand-generated by the user, or output using scripting languages such as MEL. The RI API calls are bypassed, and RIB statements are directly output by the script, using appropriate syntax for each RIB call.

Here is a comparison (taken from 'RenderMan for Poets') that shows a C program that outputs RIB and the corresponding RIB statements.

```
#include <math.h>
#include "ri.h"
void main (void)
{
static RtFloat fov = 45, intensity = 0.5;
static RtFloat Ka = 0.5, Kd = 0.8, Ks = 0.2;
static RtPoint from = {0,0,1}, to = {0,10,0};
RiBegin (RI_NULL);
RiFormat (512, 512, 1);
RiPixelSamples (2, 2);
RiFrameBegin (1);
RiDisplay ("t1.tif", "file", "rgb", RI_NULL);
RiProjection ("perspective", "fov", &fov, RI_NULL);
RiTranslate (0, -1.5, 10);
RiRotate (-90, 1, 0, 0);
RiRotate (-10, 0, 1, 0);
RiWorldBegin ();
RiLightSource ("ambientlight", "intensity", &intensity,RI_NULL);
RiLightSource ("distantlight", "from", from, "to", to, RI_NULL);
RiSurface ("plastic", "Ka", &Ka, "Kd", &Kd, "Ks", &Ks, RI_NULL);
RiTranslate (.5, .5, .8);
RiSphere (5, -5, 5, 360, RI_NULL);
RiWorldEnd ();
RiFrameEnd ();
RiFrameBegin (2);
RiDisplay ("t2.tif", "file", "rgb", RI_NULL);
RiProjection ("perspective", "fov", &fov, RI_NULL);
RiTranslate (0, -2, 10);
RiRotate (-90, 1, 0, 0);
RiRotate (-20, 0, 1, 0);
RiWorldBegin ();
RiLightSource ("ambientlight", "intensity", &intensity,RI_NULL);
RiLightSource ("distantlight", "from", from, "to", to, RI_NULL);
RiSurface ("plastic", "Ka", &Ka, "Kd", &Kd, "Ks", &Ks, RI_NULL);
RiTranslate (1, 1, 1);
RiSphere (8, -8, 8, 360, RI_NULL);
RiWorldEnd ();
RiFrameEnd ();
RiEnd ();
}
Format 512 512 1
PixelSamples 2 2
FrameBegin 1
Display "t1.tif" "file" "rgb"
Projection "perspective" "fov" 45
Translate 0 -1.5 10
Rotate -90 1 0 0
Rotate -10 0 1 0
WorldBegin
```

```
LightSource "ambientlight" 1 "intensity" 0.5
LightSource "distantlight" 2 "from" [ 0 0 1 ] "to" [ 0 10 0 ]
Surface "plastic" "Ka" 0.5 "Kd" 0.8 "Ks" 0.2
Translate .5 .5 .8
Sphere 5 -5 5 360
WorldEnd
FrameEnd
FrameBegin 2
Display "t2.tif" "file" "rgb"
Projection "perspective" "fov" 45
Translate 0 -2 10
Rotate -90 1 0 0
Rotate -20 0 1 0
WorldBegin
LightSource "ambientlight" 1 "intensity" 0.5
LightSource "distantlight" 2 "from" [ 0 0 1 ] "to" [ 0 10 0 ]
Surface "plastic" "Ka" 0.5 "Kd" 0.8 "Ks" 0.2
Translate 1 1 1
Sphere 8 -8 8 360
WorldEnd
FrameEnd
```

While the RI Spec only discusses API calls for C/C++, such API 'bindings' have been created by RenderMan users for other popular languages such as Java, Perl, Python and Tcl. So you can write standalone programs in those languages that use the API calls and output RIB by running the programs.

Shaders

Shaders can be created in two ways. The first way is to type them in by hand, using a text editor or source editor such as emacs, vi, etc. This is like writing a program in any other language such as C++ or Java, where many programmers prefer hand-editing source files compared to using more structured development environments (IDEs). The second way to create shaders is to use a graphical interface that lets you "connect the blocks" by visually hooking up pieces of shader functionality to output RSL source. Popular shader creation environments include Slim (which comes with the Maya MTOR plugin) and ShaderMan, a free standalone shader generator.

Maps

A map is a pre-existing piece of data in the form of a file, which your shader can access for use in its calculations. The most common example is a texture map which is used for adding detail to surfaces (eg. a floral

pattern on to a shirt). What are the sources of maps? Texture maps can be a hand-painted image, a scanned piece of artwork or even a digital photo (image processed or used as is). Many other types of maps are generated by RenderMan itself, in a prior 'pass'. The idea is to render the scene once to create a map, and render it again to create the desired output, taking into account the map created in the previous step. For example shadows can be created this way, by first creating a shadow map with a first pass and then rendering again to use the shadow map to compute a shadow.

To use images as texture maps, they need to be pre-processed (usually this means converting them to an image pyramid or MIP map). This is done with a texture creation program that ships with the renderer. For example in PRMan, a simplified invocation of the texture generator is this: txmake flower.tiff flower.tex The input to the txmake program is a standard TIFF image, and the output is a new .tex texture file which can then be refered to in a shader, using a texture() RSL function call.

Here is a list of maps used in PRMan. Non-PRMan renderers can also use many of these - consult your documentation to see which ones are applicable in your case.

- texture maps - used to add surface detail, and to colorize surfaces, add pre-computed lighting, affect transparency, etc.
- environment maps - these are used to mimic very shiny materials that reflect their environment
- reflection maps - to fake flat reflections, it is customary to render the scene from behind the reflecting surface (eg. a mirror) and reuse the 'reflection map' by projecting it on to the mirror in screen space
- normal maps - these are used with low resolution polymeshes to alter vertex normals, thereby giving the appearance of a high resolution model
- shadow maps - these are used to create shadows
- deep shadow maps - these contain more data than standard shadow maps, enabling richer-looking shadows (eg. colored, partially transparent shadows that exhibit motion blur)
- photon maps - used to create caustic patterns
- irradiance caches - useful for creating subsurface scattering effects
- occlusion maps - useful for creating visually-rich ambient lighting
- brick maps - these are in a PRMan-specific format (a form of tiled 3D MIP map), useful for storing calculations related to radiosity

Ways to extend RenderMan

- shaders can be considered "appearance plugins" since they are external programs which the renderer

invokes, to carry out shading calculations.

- the 'Procedural' RIB call (and the corresponding RiProcedural() API call) make it possible to write standalone programs and DSOs (dynamically linked plugins) which can output RIB fragments. This facility can be used to create custom geometry (eg. fire sprites, foliage, characters..) and to render specialized primitives which are not described in the Spec.
- 'display drivers' DSO plugin files can be used to output the rendered pixels using custom image formats.

RIB - syntax, semantics

In this section we will take a brief look at the syntax of RIB files and how scene data is organized in a typical RIB file.

A RIB file contains a sequence of requests to the renderer, organized in the form of a keyword, often followed by additional data. Some of the data are specified as attribute/value pairs. Here are some examples:

```
Projection "perspective" "fov" 25
Rotate 19 0 0 1
TransformBegin
Surface "plastic" "Ka" 0.1 "Kd" 0.05 "Ks" 1.0 "roughness" 0.1 "specularcolor" [1
.4 .1]
```

Lines beginning with a # denote a comment and are ignored by the renderer, while lines beginning with a ## are renderer 'hints'. Examples:

```
# Correct for monitor gamma of 2.2
Exposure 1.0 2.2
##Shaders PIXARmarble, PIXARwood, MyOwnShader
##CapabilitiesNeeded ShadingLanguage Displacements
```

RIB files do not contain loops, branches or function declarations. In other words, it is not a programming language. It is more like a declaration language for specifying scene elements.

RIB files typically contain attributes and transformations expressed hierarchically, in the form of nested blocks (using AttributeBegin/AttributeEnd pair of keywords or TransformBegin/TransformEnd). Examples:

```
TransformBegin
 Translate -1.3 1 0
 Scale .5 .5 .5
 TransformBegin
 Color [.22 .32 .58]
 ConcatTransform [1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]
 PointsGeneralPolygons [1] [3] [2 1 0] "P" [0 0 0 1 0 0 1 0.7265 0]
 TransformEnd
TransformEnd
AttributeBegin
Surface "rmarble" "veining" 5 "Ka" 0.25 "roughness" 0
Color [.96 .96 .04] #[0.52 .52 .58]

TransformBegin
# Translate 0 0 0
Disk 0 0.5 360
TransformEnd

TransformBegin
Translate 1 0 0
Disk 0 0.5 360
TransformEnd
AttributeEnd
```

By nesting transforms and attributes this way, complex geometry and material setups can be specified. Underlying this hierarchical specification are the notions of a graphics 'state' and a 'current transformation matrix', and the attribute/transform blocks manipulate these by pushing and popping data off stacks that maintain the current state.

Geometry, lights and materials are specified inside a WorldBegin/WorldEnd block, while camera and image parameters come before the WorldBegin:

```
Format 750 750 1
Display "SquareSquaresRel.tiff" "framebuffer" "rgb"
Imager "background" "color" [.57 .22 .13]
Projection "perspective" "fov" 90
Projection "orthographic"
ScreenWindow -20 20 -20 20
LightSource "distantlight" 1
Translate -16 -16 0
Translate 0 0 1
WorldBegin
AttributeBegin
# 0,0
Color .25 .25 .25
```

```
Scale 18 18 1
Polygon "P" [0 0 0 1 0 0 1 1 0 0 1 0]
AttributeEnd
AttributeBegin
# 18,0
Color .5 .5 .5
Translate 18 0 0
Scale 14 14 1
Polygon "P" [0 0 0 1 0 0 1 1 0 0 1 0]
AttributeEnd
AttributeBegin
# 18,14
Color .75 .75 .75
Translate 18 14 0
Scale 4 4 1
Polygon "P" [0 0 0 1 0 0 1 1 0 0 1 0]
AttributeEnd
WorldEnd
```

In the above example, the Display statement specifies a framebuffer "driver" (destination) for the output image. Image resolution is declared in the Format statement.

RIB files do not include shader files in them. Rather, they call out (or reference) shaders, which are separate from the RIB file and must be accessible to the renderer. Here are a couple of examples of shader specification:

```
# surface shader specification
Surface "wood" "Ka" .1 "Kd" 1. "grain" 12 "swirl" .75 "darkcolor" [.1 .05 .07]
# ...
# displacement shader call
Displacement "noisydispl" "ampl" .06 "freq" 5
# ...
```

RIB files can include other files (usually containing blocks of data in RIB syntax) using the ReadArchive statement. This is good for reusing assets (which can be kept in separate files ready for including in a master scene file), thereby keeping scene file sizes small.

```
WorldBegin
LightSource "distantlight" 1
LightSource "ambientlight" 2
Opacity [0.5 .5 .5]
Surface "showN"
# Costa_semi.dat contains polymesh data for a section of the Costa minimal
```

```
surface
ReadArchive "Costa_semi.dat"
WorldEnd
```

Note that the RIB specification does not have a provision for specifying time-varying parameters, ie. animation curves. This means that for an animation sequence, each frame should be a self-contained block of RIB capable of producing an output image. While the RIB spec. does allow for multiple frames of data to be stored in a single RIB file (using FrameBegin/FrameEnd blocks), it is more common for a RIB file to contain just one frame's worth of data. This allows RIB files for an image sequence to be distributed to multiple machines on a render farm, where each machine receives a single RIB file from the sequence and generates a single image from it. This also means that RenderMan renderers do not output animations in movie formats such as MPEG or Flash video. Animations are simply sequences of still images. The stills need to be post-processed using compositing and editing programs to create movie files for television, DVD production, etc. For movie production, post-processed stills (eg. composited with live action footage) are output to film using a film recorder.

Shader writing

Here are some useful things to know about writing shaders. As mentioned before, RenderMan's powerful shading language (RSL) lets users write custom programs called shaders to completely define surfaces and their interaction with light sources. These shaders are referenced in RIB files where they are used to specify materials, light sources, etc. In that sense, shaders can be thought of as RIB plugins.

Highlights of RSL syntax:

- C-like language.
- types include float, string, color, point, vector, normal, matrix; these can be arrays too.
- usual set of operators, eg. +,-,*,/,%,== etc.
- control statements include if() and for().
- a rich collection of built-in functions that include mathematical, trigonometric, lighting, color transformation and map access calls. It is this powerful function library that makes RSL shader-writing a joy. Consult the Spec and your renderer's manual for a list of available functions.
- user-defined functions can be used to isolate and reuse blocks of code.
- computation-expensive calls can also packaged into "shadeops" which are compiled DSOs (eg. custom noise functions, Fourier transform code etc. can be coded up as shadeops). This is mostly done to gain

execution speed.

- preprocessor directives are available, eg. #include, #define, etc.

You would write an RSL shader "blind", ie. without knowing exactly where on a surface it is going to invoked, how many times or in what order. Also, you do not have access to information about geometry that surrounds your shading neighborhood. What is available to you in the form of global variables is detailed information about the current point being shaded, for instance, its location, surface normal, texture coordinates, etc.

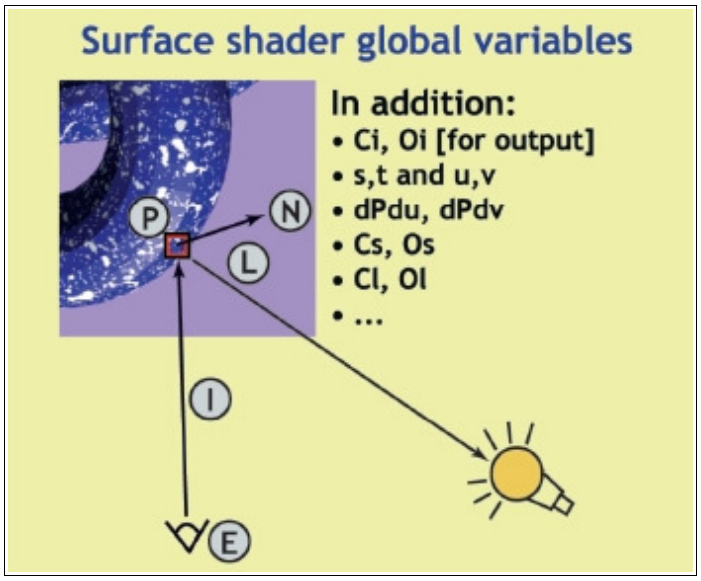


Image 2: Surface shader global variables

Image 3 shows a list of global variables pertaining to surface shaders. Consult the Spec for similar lists for other shader types.

Information based on the RiSpec 3.2. Copyright Pixar Animation Studios

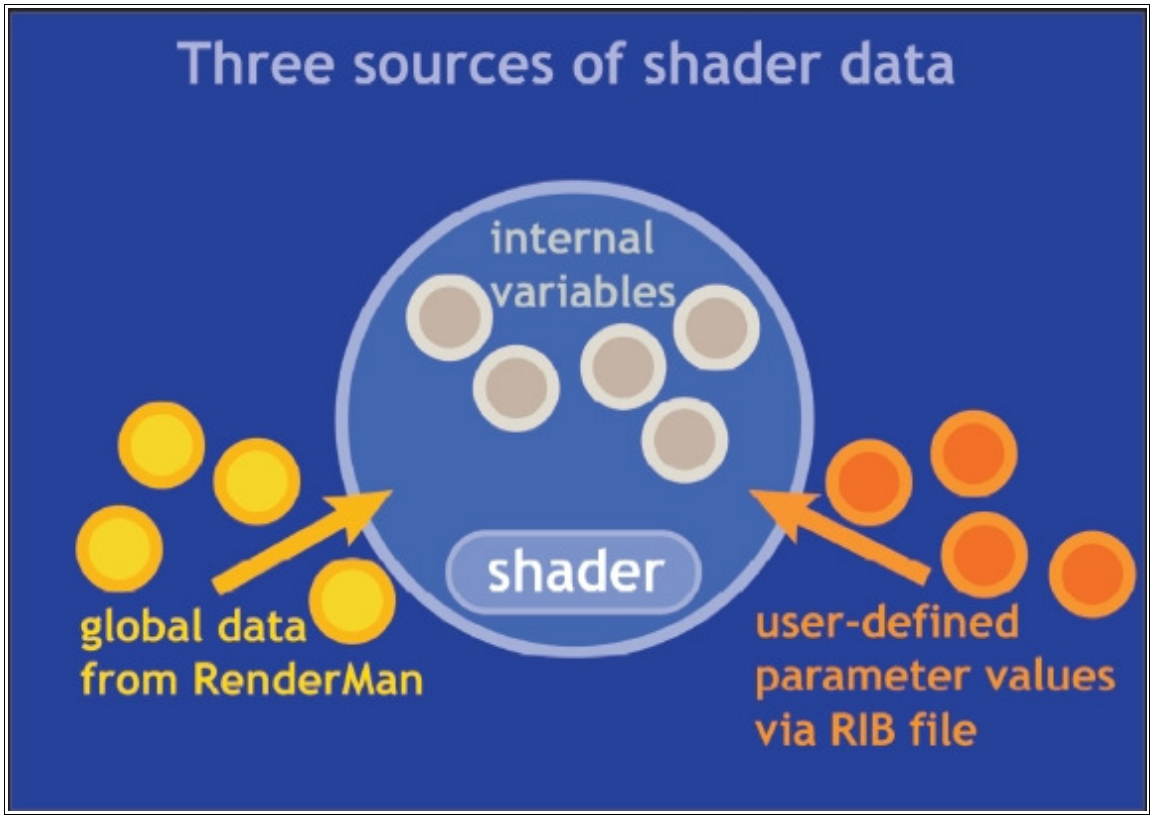
Surface Shader Variables

Variable Name	Type	Storage class	Description
Cs	color	varying	Surface Color described on the RIB file
Os	color	varying	Surface opacity described on the RIB file
P	point	varying	Position of shaded surface
dPdu	vector	varying	Derivative (tangent) of the surface position along u
dPdv	vector	varying	Derivative (tangent) of the surface position along v
N	normal	varying	Surface shading normal
Ng	normal	varying	Surface geometric normal
u,v	float	varying	Surface parameters
du,dv	float	varying	Change in surface parameters
s,t	float	varying	Surface texture coordinates
L	vector	varying	Incomming light ray direction*
Cl	color	varying	Incomming light ray color *
Ol	color	varying	Incomming light ray opacity*
E	point	uniform	Position of the eye or camera
I	vector	varying	Incident ray direction. Direction vector going from the camera to the current shading point
ncomps	float	uniform	Number of color components
time	float	uniform	Current shutter time
dtime	float	uniform	Amount of time covered by this shading sample
dPdtime	vector	varying	How the surface position P is changing per unit time, as described by motion blur in the scene.
Ci	color	varying	Shader output color
Oi	color	varying	Shader output opacity

* Available only in illuminance statements

Image 3: Global shader variables specified by the RiSpec 3.2

In addition to the global variables, data can come into your shader via its argument list, where values for the parameters in the list are specified in the RIB call(s) for your shader. Also, data can be calculated, stored and used inside a shader via local variables



As for shader creation tools, these have already been mentioned. With Maya, you can use Slim or MayaMan. Standalone shader generators include ShaderMan and Shrimp. The Air renderer comes with Vshade, a visual tool for creating shaders. But the most popular method of creating RSL shaders is to type them in source code form into a text or programmers' editor. The resulting source file needs to be compiled in order for the renderer to be able to load and invoke the shader. Eg. for PRMan, the compilation command is shader <myshader.sl> The result will be <myshader.slo>, which needs to accessible by PRMan when it renders a RIB file that makes a call to <myshader>.

With RSL you can create five types of shaders:

- surface
- displacement
- light
- atmosphere
- imager

Here are six shaders (and sample results) to give you a taste for RSL.

```
surface showN()
{
  point PP;
  normal Nf;
  vector V;
  color Ct, Ot;


  normal NN = normalize(N);
  vector posNorm = 0.5*(vector(1,1,1)+NN);
  color posCol = color(comp(posNorm,0),comp(posNorm,1),comp(posNorm,2));
  color posGCol = 0.25*color(comp(posNorm,1),comp(posNorm,1),comp(posNorm,1));
  Oi = Os;
  Ci = posCol;
}
```

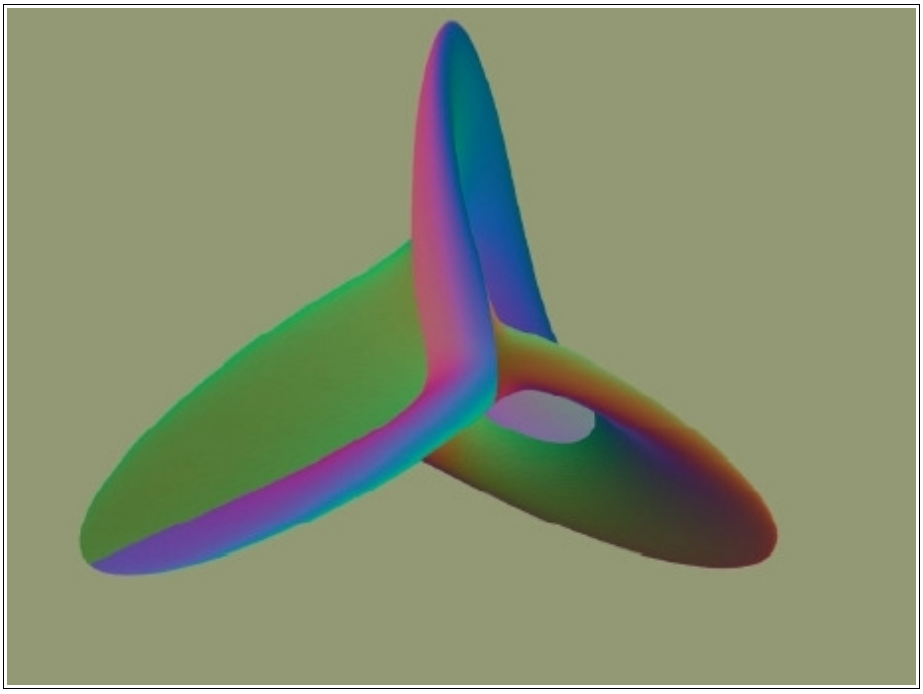


Image 4: showN surface shader output

```
/* Straightforward texture lookup */
surface tex(string tmap="generic.tex";)
{
  float alpha;

  /* get base color from map */
  if(tmap!="")
  {
    color Ct = color texture(tmap,s,t);
    alpha = texture(tmap[3],s,t);
    Oi = alpha;
    Ci = Oi*Ct;

  }

}
```

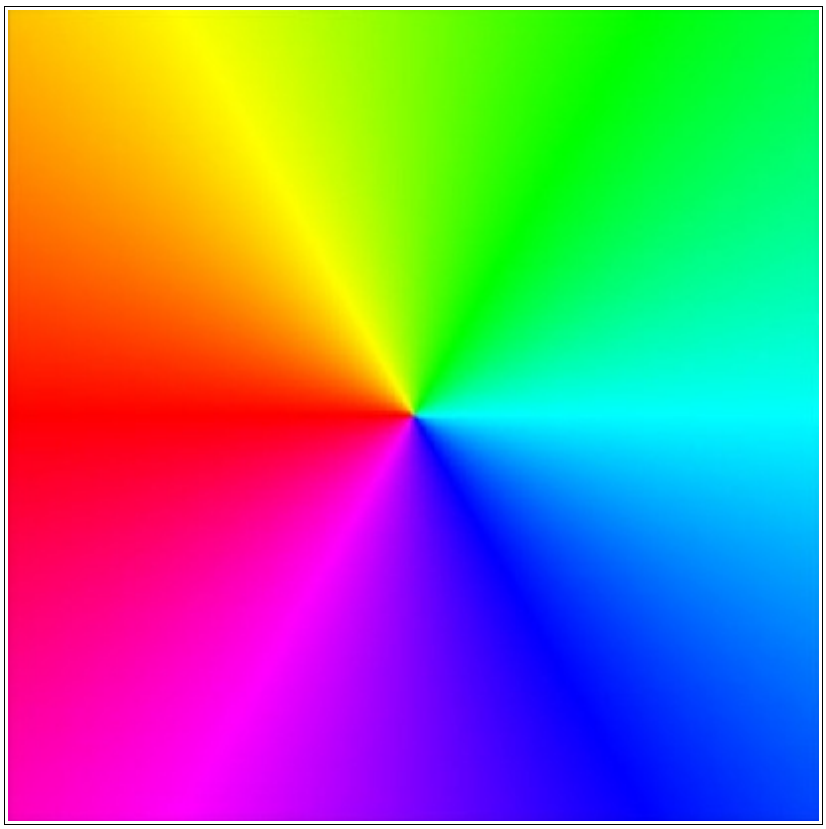


Image 5: Texture to be used for tex surface shader

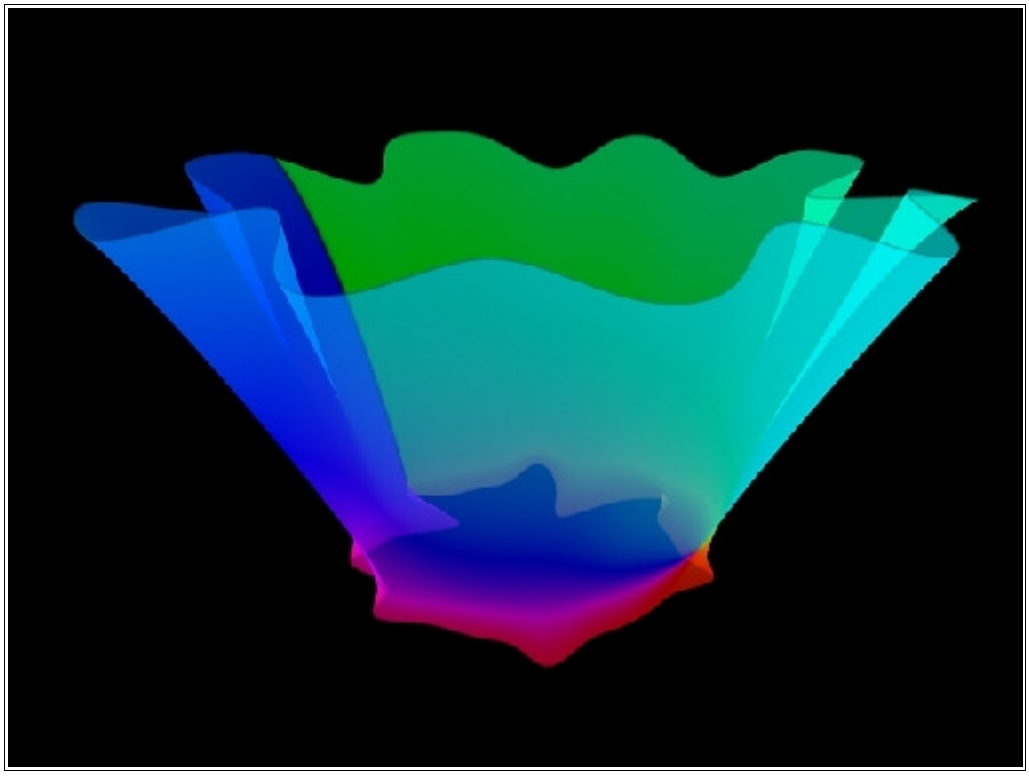


Image 6: tex surface shader output

```
displacement sinewaves(float freq=1.0, ampl=1.0, sphase=0, tphase=0, paramdir=0)
{
  // displace along normal, using sin(s) or sin(t) or both
  if(0==paramdir)
    {
      P += ampl*sin(sphase+s*freq*2*PI)*normalize(N);
    }
  else if (1==paramdir)
    {
      P += ampl*sin(tphase+t*freq*2*PI)*normalize(N);
    }
  else
  {
      P += ampl*sin(sphase+s*freq*2*PI)*sin(tphase+t*freq*2*PI)*normalize(N);
  }
  N = calculatenormal(P);
}// sinewaves
```

Image 7: Displacement shader output

```
light
Kessonlt(
    float  intensity=1 ;
    color  lightcolor=1 ;
    float freq=1.0, coneangle=PI/2;
)
{
  point Pt = freq*transform("shader",Ps);
  vector ldir = 2*noise(freq*Pt) - 1;
  solar(ldir,coneangle)
    {
      Cl = intensity * lightcolor;
    }
}
```

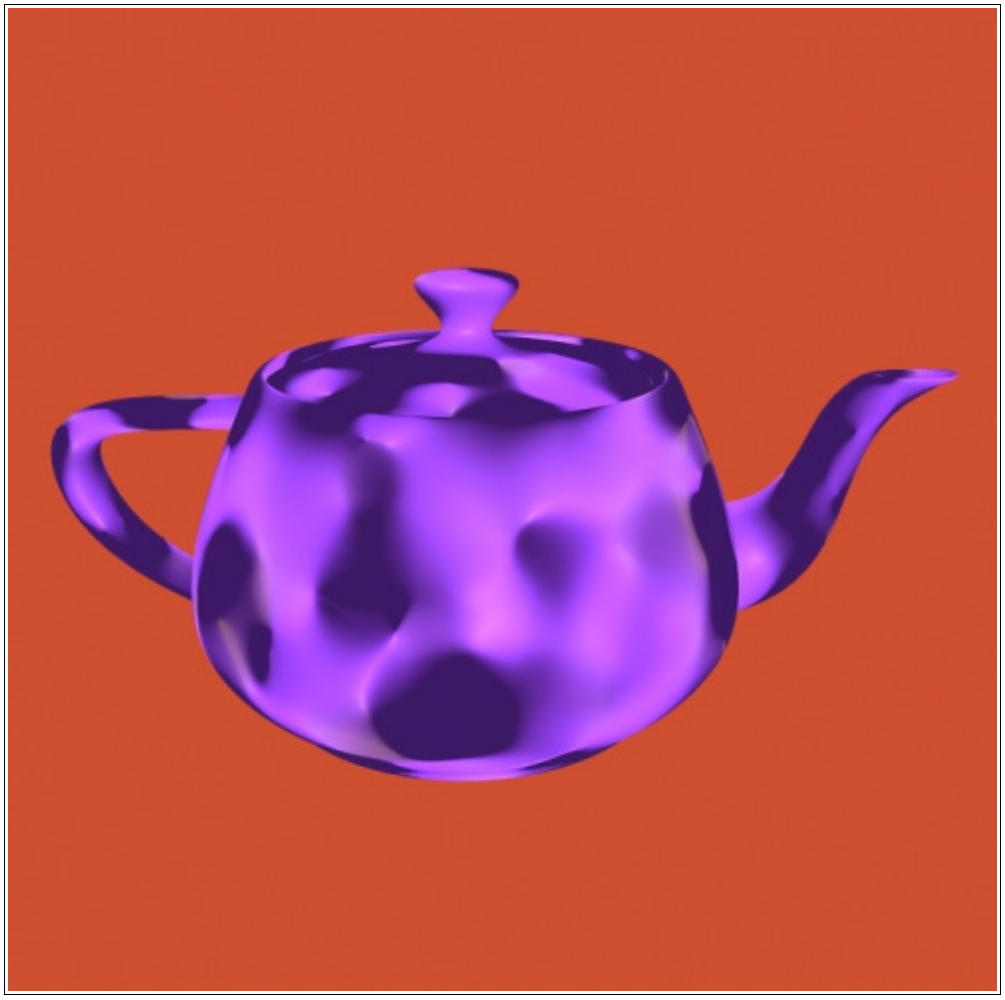


Image 8: Light shader output

```
volume underwater(
    float mindist=0, maxdist= 1;
    color fg=1, bg=1;
    float inten=1, gam=1, mixf=0.5;
    )
{
    color c;
    float d;

    d = length(I);
    if(d<=mindist)
      c = fg;
    else if(d>=maxdist)
      c = bg;
    else
      {
```

```
        d = (d-mindist)/(maxdist-mindist);
        d = pow(d,gam);
        c = mix(fg,bg,d);
      }

    Ci = inten*mix(Ci,c,mixf);
    Oi = mix( Oi, color (1,1,1), d );
}// underwater()
```

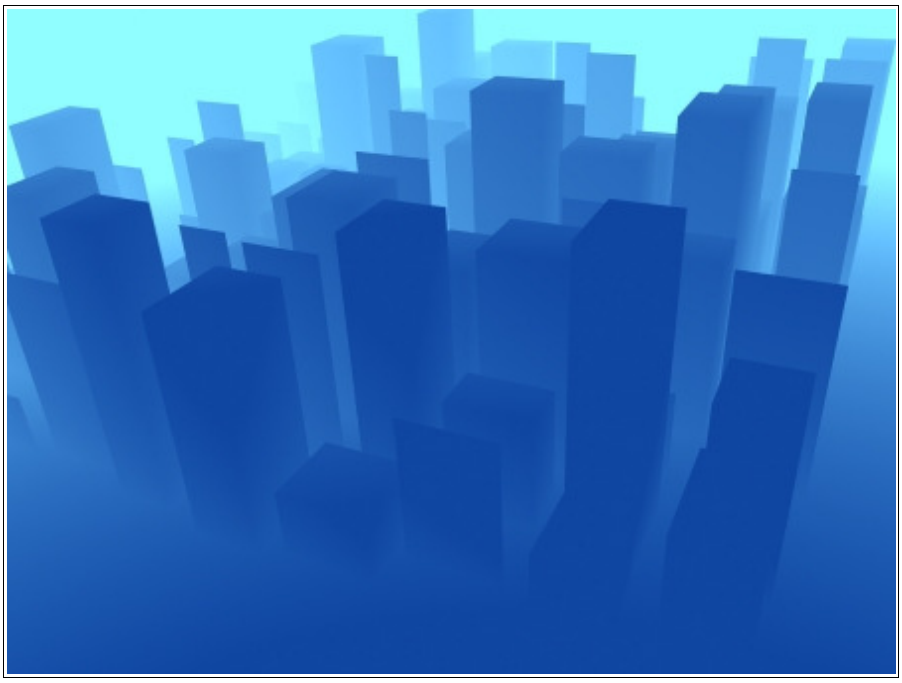


Image 9: Volume shader Output

```
imager Imager_ramp
(
 color ctop = color(1,1,1);
 color cbot = color(0,0,0);
 float gam=1.0;
 )
{
 float curr_y;
 float rez[3];
 color rampcol;
 float mixf;

 option("Format",rez);
```

```
  curr_y = ycomp(P)/ rez[1]; // 0 to 1, top to bottom
  curr_y = pow(curr_y,gam);

  rampcol = mix(ctop,cbot,curr_y);
  Ci += (1-Oi)*rampcol;
  Oi = 1.0;

}
```

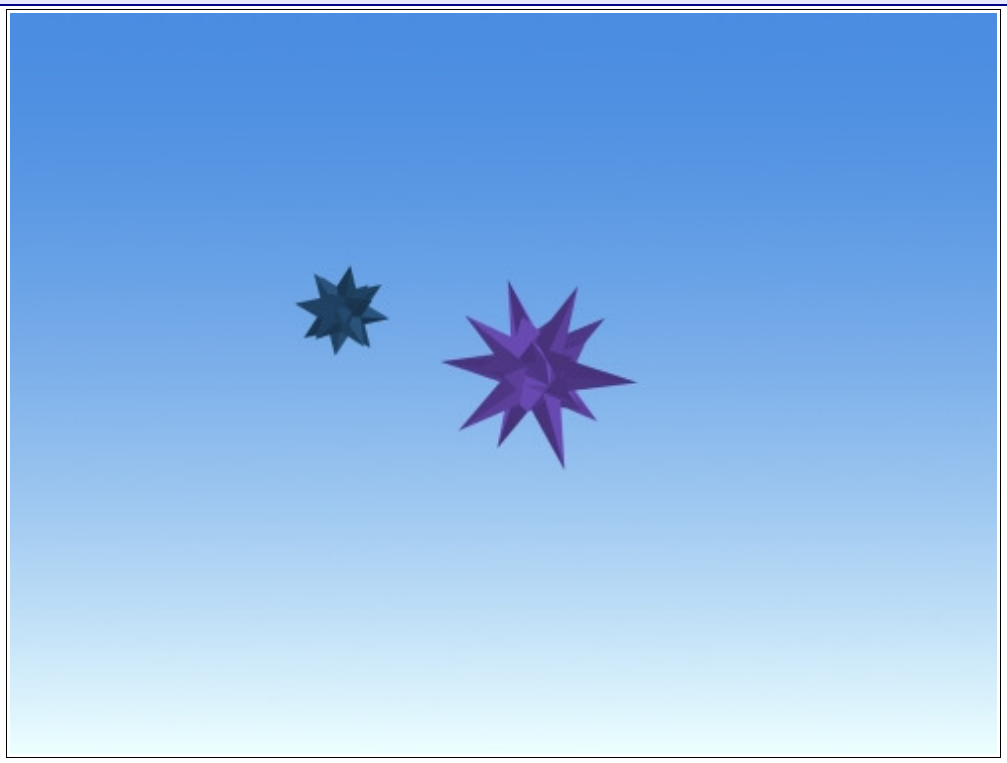


Image 10: Imager shader output

Resources

Since RenderMan has been around for a while, there are a lot of resources available to you for further exploration. Here are the main ones.

Books

- The RenderMan Companion - this was the first-ever RenderMan book. The first half deals with the C API for generating RIB, while the second half discusses shader-writing.
- Advanced RenderMan - lots of valuable info. on a variety of topics including lighting techniques and

shader anti-aliasing.

- Essential RenderMan Fast - an introductory book dealing with the C language API, shading language and RIB.
- Rendering for Beginners - another introductory book that documents RIB calls and has sections on cameras, image quality and shader-writing.
- Production Rendering - written for software developers, this book is a collection of good ideas for implementing a modern film quality renderer.
- Texturing and Modeling - this book is about RenderMan per se but does use the RSL too illustrate many examples.

Notes

- SIGGRAPH course notes - ever since RenderMan's inception there have been courses held at SIGGRAPH on various aspects of RenderMan, including shader-writing, global illumination, use in production, etc.
- Steve May's RManNotes - a nice collection of lessons on shader-writing, using a layering approach. RManNotes is at http://accad.osu.edu/~smay/RManNotes/rmannotes.html
- Larry Gritz's 'RenderMan for Poets' - a small document (19 pages) which lists and explains the basic procedural API and corresponding RIB calls. You can find it at http://www.siggraph.org/education/materials/renderman/pdf_tutorial/poets.pdf.
- Malcolm Kesson's detailed notes on RIB and RSL, at http://www.fundza.com.

Forums, portals

- http://www.renderman.org - this is the RenderMan Repository ("RMR"), containing a ton of information and links. Here you can find a lot of shader source code and also links to PDFs of past SIGGRAPH course notes.
- http://www.rendermania.com - News and info. about RenderMan, and a rich collection of links to other people's pages and sites.
- http://www.rendermanacademy.com - excellent PDF tutorials focusing on RIB and shader writing.
- Google Newsgroup - comp.graphics.rendering.renderman - a place to post questions and to read the c.g.r.r FAQ maintained by Larry Gritz.
- http://www.highend3d.com/renderman - another place for downloading shaders and posting questions.

- http://www.deathfall.com - a broad CG 'portal' with an active RenderMan section

In addition to the above, you can find more information/knowledge at these sites for specific RenderMan implementations:

- PRMan: http://www.pixar.com
- 3Delight: http://www.3delight.com/
- Air: http://www.sitexgraphics.com/
- Angel: http://www.dctsystems.co.uk/RenderMan/angel.html
- Aqsis: http://www.aqsis.com
- Pixie: http://www.cs.utexas.edu/~okan/Pixie/pixie.htm
- RenderDotC: http://www.dotcsw.com/

What The RISpec Never Told You

Dan Maas - Mass Digital

Here are a few short, non-obvious, but essential tips and tricks for getting the most out of RenderMan. Many of these ideas are hard to understand just by reading the spec, or are not covered in the spec at all

Geometry Tips

Understanding Sidedness

RenderMan offers three choices of "sidedness" for graphics primitives: Sides 1, Sides 2, and Sides 2 with the extra "doubleshaded" attribute (`Attribute "sides" "doubleshaded" [1]`). These have two effects: they control what values of N are passed to surface shaders, and they control whether non-camera-facing primitives are displayed.

Sides 1: RenderMan shades each primitive once with its "natural" surface normals (i.e. those passed in the RIB, or the default normals of the primitive if none are specified). Primitives are displayed only where the surface normals face the camera.

Sides 2: RenderMan still shades each primitive only once with its "natural" surface normals, but displays the primitive regardless of whether the normals face the camera. This is the default setting.

Sides 2 with `"doubleshaded" 1`: RenderMan shades each primitive twice, once with its "natural" surface nomals and again with the opposite normals, and displays whichever part has camera-facing normals.

The "double-sided" settings on most other 3D packages correspond closely to Sides 2 with `doubleshaded`. RenderMan's default double-sided behavior is rather unusual, since the normal passed to the surface shader may actually be facing away from the camera. It's up to the shader to flip the normal if it wants to shade the camera-facing side. This is why you see faceforward() calls in most RenderMan shader examples.

However, faceforward() doesn't always work correctly.

Avoid faceforward(N, I)

faceforward(N, I) flips the surface normal, if necessary, to ensure that it points toward the camera. However, this can lead to rendering artifacts on silhouette edges and on very thin objects. A single micropolygon straddling the silhouette edge may yield opposite results for faceforward() at adjacent vertices. Interpolating the resulting colors across the micropolygon gives improper results, such as "leaks" of light around the rim of an object.

For this reason, I recommend that RenderMan users not rely on faceforward() to fix their surface normals. Instead, pick one of the two non-default sidedness settings: either use Sides 1 and ensure that your surface normals always point the right way, or use Sides 2 with `"doubleshaded" 1`. Either way you won't have to use faceforward() in your surface shaders.

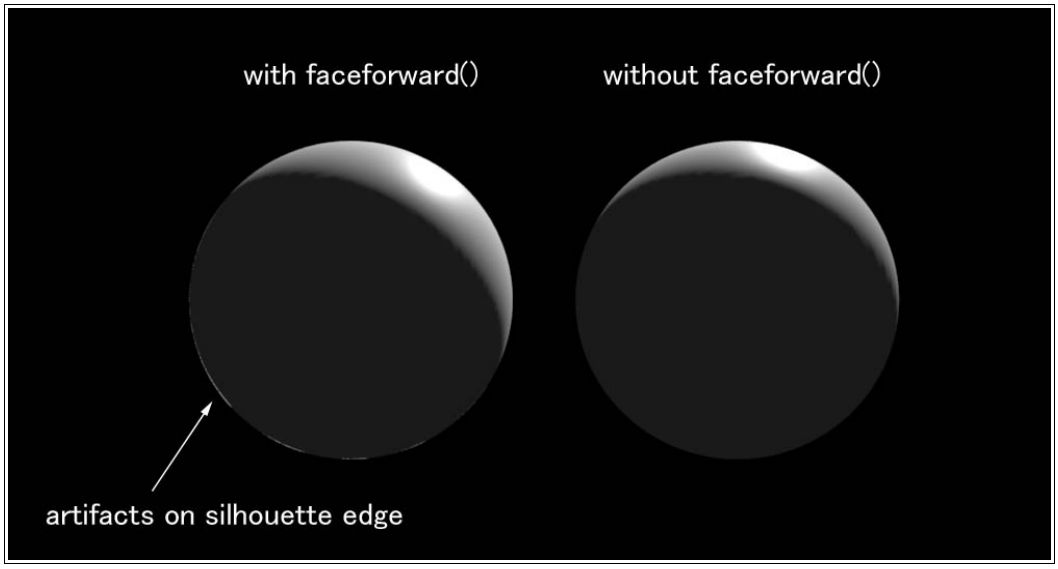


Image 11: side effects of using faceforward() in shaders

Bump mapping polygon primitives without faceting

New RenderMan users are often surprised when trying out a shader that uses calculatenormal() (like most bump and displacement shaders) on their polygonal objects. Instead of a nice smooth surface, they see faceting, as if their smoothed surface normals are being ignored. Well, they are.

calculatenormal() attempts to compute a surface normal by taking the cross product of the parametric derivatives dPdu and dPdv. On a polygon primitive, dPdu and dPdv run flat along the plane of each face. The resulting normal is therefore always perpendicular to each face, giving a faceted apperance. calculatenormal() doesn't take into account any smoothed normals you supplied in the RIB.

So how can you apply bumps and displacements to a polygonal object without faceting? One solution is due to Thad Beier, who suggests taking the difference between the original smoothed and faceted normals, then adding it back to the result from calculatenormal() after displacement [Beier 1999]. Note that the undisplaced faceted normal is always available in the Ng global variable.

```
// record difference between original smoothed and faceted normals
normal diff = normalize(N) - normalize(Ng);

// assume P has been moved by a displacement
P = P + some_displacement;

// calculate the new (faceted) surface normal
normal n_faceted = normalize(calculatenormal(P));

// add back the difference
N = n_faceted + diff;
```

I call this a "90% solution" because it usually works well, but if you look very closely you might still see artifacts. If you want a "100% solution" you'll have to use NURBS or subdivision surfaces; polygons just aren't that good for representing curved surfaces!

The importance of "clean" geometry

Although RenderMan is willing to accept any geometry you can pass through its interface, supplying clean geometry goes a long way towards avoiding rendering artifacts and poor performance. What do I mean by "clean"?

Ensure surface normals are consistent: If you take care to ensure that all your surface normals point in a consistent direction, you can use Sides 1, which allows the renderer to cull geometry more efficiently. Plus, you can eliminate faceforward() from surface shaders, avoiding artifacts and giving you a small speed-up.

Avoid self-intersections and overlaps: Self-intersecting geometry is problematic for REYES renderers. Micropolygons straddling the intersection may show interpolation artifacts due to the abrupt change in illumination as they plunge into the intersecting surface. Overlapping parallel surfaces are another source of trouble; they force you to use higher bias settings for shadow maps and ray tracing in order to avoid false intersections. High bias causes problems of its own, such as visibly detached shadows and reflections. Also, if you are using 3D point-cloud baking, it's possible for baked values to "leak" from the interior surface through to the exterior surface.

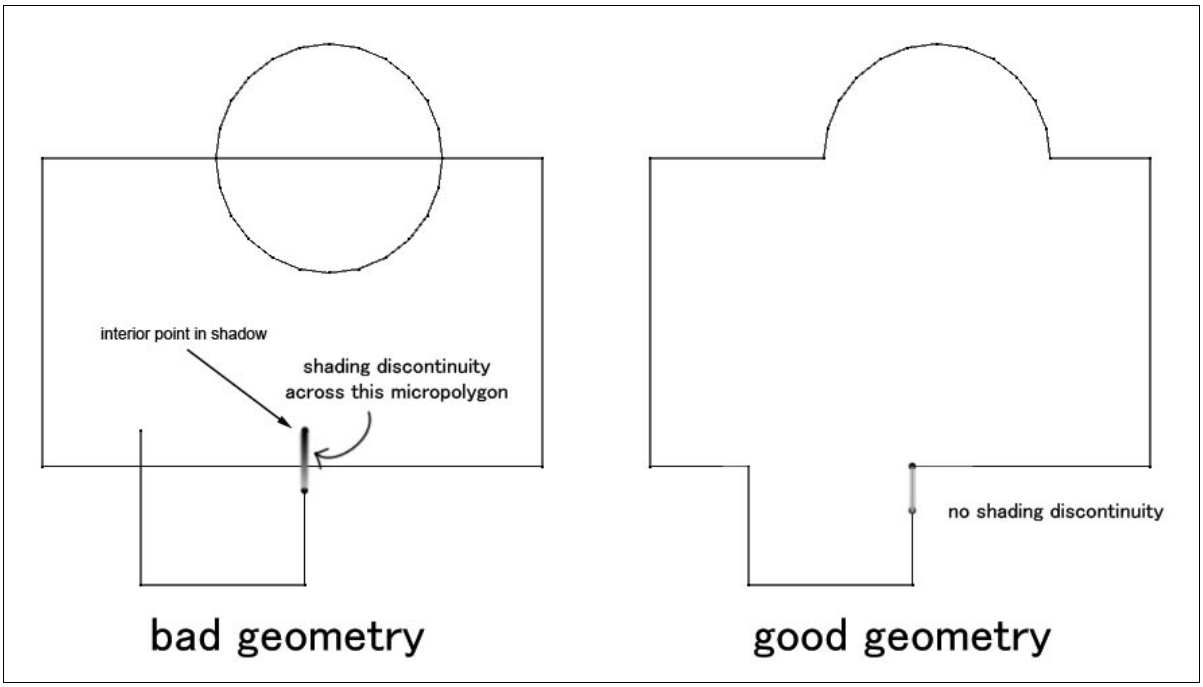


Image 12: Avoid geometry intersections

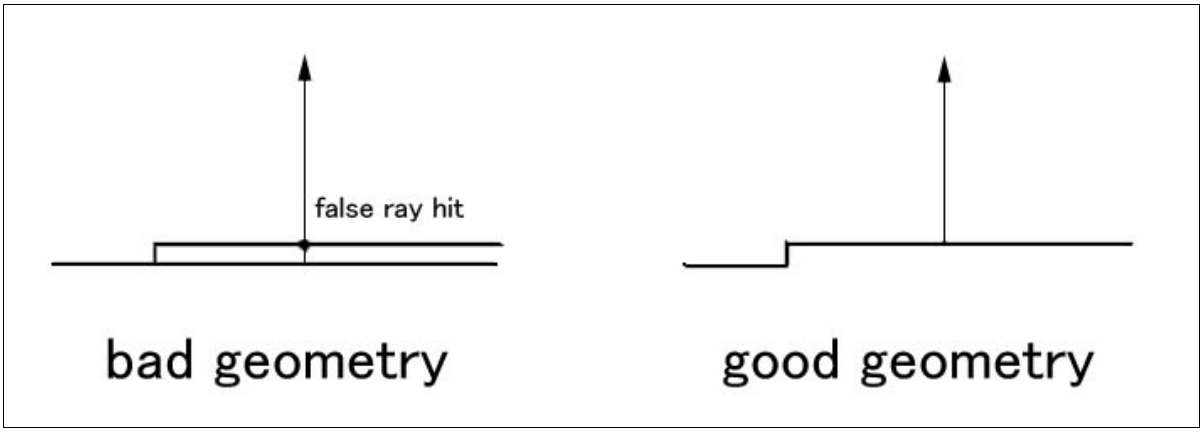


Image 13: Avoid geometry overlap

Avoid polygons with more than four sides: Most RenderMan implementations do not handle polygons with more than four sides very well. They usually must carve your polygons into three- or four-sided pieces in order to shade them, and rarely do a good job of it. Often you will find that the renderer splits large multi-sided polygons into thin shards or spiky fans, which waste resources and can lead to rendering artifacts. It's best to carve your multi-sided polygons into nicely-proportioned quads and triangles before passing them to the renderer, either by hand or by using a high-quality subdivision algorithm. Avoid T-junctions when doing this, since they tend to cause shading artifacts.

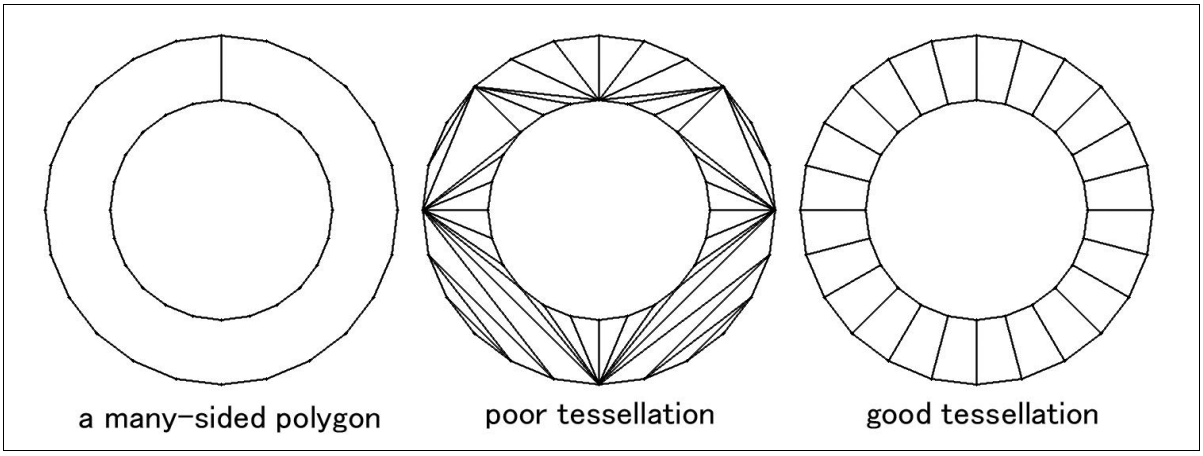


Image 14: Avoid using N-gons

The "micropolygon explosion"—and how to avoid it

Most RenderMan-compliant renderers dice your primitives into pixel-sized micropolygons prior to

displacement, shading, and motion blurring. Usually renderers dice on demand, keeping only a small number of micropolygons in memory as they produce the final image. However, if your displacement bound (maximum possible displacement) is large, perhaps more than a few tens of pixels on screen, or your object is undergoing a large amount of motion blur, it's quite possible for the renderer to exhaust huge amounts of memory storing micropolygons that are displaced or blurred across large areas on the image. This problem becomes much worse for high-resolution renderings, since the number of micropolygons that need to be stored is proportional to the square of image resolution.

There are several ways to deal with a "micropolygon explosion." The best approach depends on whether your explosion is caused by motion and depth-of-field blur, or displacement bounds:

If your explosion is due to a large amount ofmotion or depth-of-field blur, try setting GeometricApproximation "motionfactor" to a larger value. This reduces the shading frequency for strongly blurred objects, which decreases the number of micropolygons with little or no loss in image quality. The default value is zero, meaning no reduction at all. I always set this to 1 by default for my scenes.

Rendering foreground, mid-ground, and background objects as separate passes might help by reducing the amount of geometry that must be kept in memory.

And finally, if all else fails, you can always render the scene at a lower resolution. With large motion- and depth-of-field blurs, this shouldn't affect the image quality too much.

If your explosion is due to a large displacement bound, especially on large planar objects like terrain, try rendering the scene "sideways." Most REYES renderers march across the image top-to-bottom, left-toright. This causes problems with displaced terrain, since a huge amount of geometry must be diced and retained as the renderer marches across the distant, strongly foreshortened horizon. By tipping the camera 90 degrees (and adjusting the height, width, and ScreenWindow settings appropriately), we can force the renderer to march down the terrain in columns rather than across it in rows. In one of my scenes, this change alone dropped the memory usage from 9GB to 800MB per frame.

A further approach for handling displacement is to reduce your displacement bound by "baking" part of the displacement into the geometry. For example, if you have a terrain consisting of a plane displaced upwards by a texture map, you could translate the plane upwards by half the maximum displacement, and subtract the same amount in the displacement shader. Now you can use a displacement bound that is only half as large as before. Sometimes it's possible to cheat by reducing the displacement bound lower than its theoretical maximum value. Some renderers like PRMan will tell you exactly when and how much the displacement exceeded the bound,

helping you nail down a safe value.

RunProgam procedural pitfalls

RunProgram procedurals allow you to generate geometry at render time from a command-line program invoked by the renderer. I won't go into full detail on how to write one of these, but I'd like to point out four common mistakes that trip up authors of these procedurals:

Your procedural must loop, reading input from stdin continuously until it receives an end-of-file result. Even if there is only one instance of your procedural in the scene, a multi-threaded or multi-process renderer might need to call the program more than once.

Don't forget to send the terminating byte (hex `0xff`) after you finish writing your RIB to the renderer.

Flush output buffers to force the RIB to be written out before your procedural goes back to waiting for input. In C this means calling fflush(). Other languages have similar functions—check your documentation.

Expect your program to be invoked concurrently. As more and more RenderMan implementations adopt multi-threaded and multi-process techniques, it will become typical for procedurals to be invoked in multiple concurrent instances. If your procedural accesses any kind of shared resource like a database, you'll have to take care that separate instances don't step on each other's toes.

Shading Tips

Texture projection methods

Let's review some of the different ways a shader can map a 2D texture onto a 3D surface:

Natural uv coordinates (NURBS, patches, and quadrics only)

These primitives all have natural uv parameterizations following from their mathematical definitions. Note that RenderMan always scales the parameters into the range 0–1, even for less-than-complete primitives, like portions of a sphere.

Manual assignment of st coordinates (subdivision surfaces & polygons only)

Subdivision and polygon primitives do have uv parameterizations assigned by the renderer, but these are usually

arbitrary, inconsistent between faces, and not useful for texturing. It falls to the modeler to assign appropriate texture coordinates to each vertex manually.

(note on terminology: most 3D packages refer to these user-assigned coordinates as “UV coordinates,” but in RenderMan they are called “st coordinates.” “UV coordinates” mean only the natural parameters of the surface in RenderMan parlance.)

2D planar/cylindrical/spherical projection

As with most other 3D packages, it’s easy to project textures in planar, cylindrical, or spherical projections. Look in any graphics textbook for the equations. You will need to transform P into a coordinate system that is anchored to the object prior to projection if you don’t want your texture to “swim.” Beware of discontinuities in the projection (see "Texture Filtering" below for details).

Perspective projection

This is useful for texturing a scene by projecting an image “through the camera.” For example, you could render a rough “screen-shot” from the camera’s perspective, paint details on it, and then map the painted image back onto the scene.

There is a good article on exactly how to do this by Tal Lancaster in the SIGGRAPH 2000 RenderMan course notes [Lancaster 2000].

I’ll just add two minor fixes to Tal’s technique. If you pull the camera back behind the projection location, you’ll see a double image of the projection behind the camera and a strange artifact along the eye plane. The double image appears because the perspective transform is symmetrical across the eye plane; you can get rid of it by fading out the texture based on the z coordinate of P after it’s gone through the projection matrix. The eye plane artifact is due to the texture filter going haywire at a discontinuity in the texture coordinates. You can get rid of it by clamping the filter width (see Texture Filtering below).

Image projection

Sometimes it’s necessary to map a 2D texture map directly onto the rendered image as if it’s stuck in front of the camera. For example, you might use this for a planar reflection element, or as a kind of mask on the image. This mapping is easy in RenderMan shading language: just transform P to NDC space and use the x and y components as your texture coordinates.

The difference between NDC and perspective projection is that NDC space always stays with the camera, whereas perspective projections can remain anchored to objects as the camera moves around the scene.

Passing geometrical quantities to shaders

New shader writers are often tripped up when they have a shader that takes a geometrical quantity like a point or vector as an argument:

```
surface mysurface(point texture_origin = point(0);) {...}
```

Say you invoke this shader as follows:

```
AttributeBegin
Translate 3 2 1
Surface "mysurface" "point texture_origin" [0 0 0]
...
AttributeEnd
```

Within the shader, texture origin is unexpectedly given a non-zero value! What is going on?

RenderMan always transforms geometrical quantities into “current” space before passing them to shaders. This goes for shader parameters as well as the standard variables like P and N. The unexpected value for texture origin actually is the point (0,0,0) at the Surface RIB call, but it’s been transformed into “current” space coordinates. If you want to recover the same numerical value as seen in the RIB, you’ll have to transform the point back to shader space manually. Or, pass its components as individual float parameters, which are never transformed.

Understanding surface transparency

When transparent surfaces overlap, RenderMan uses premultiplied alpha compositing to determine the resulting color. The formula is:

```
result = (1 - Oi) * background + Ci
```

Notice that the foreground color, Ci, is not diminished by the surface opacity, Oi. That's what "premultiplied" means. In other words, Ci is assumed to have been multiplied by the opacity already. This must be done manually within the surface shader. That's why you almost always see a statement like Ci*= Oi; at the end of a shader.

If Oi is zero, the foreground color is simply added to the background color. You can use this fact to create additive surfaces for special effects like fire or glowing objects. Just omit the multiplication of Ci by Oi from your shader. Or, you can set Oi to an intermediate value, greater than zero but less than Oi, to get partial glows.

Gamma in Texture Maps

Color values in RenderMan shaders usually represent light intensity linearly. This makes sense because lighting functions like diffuse() and specular(), as well as alpha compositing, all produce correct results only when operating on linear light quantities.

However, the vast majority of 8-bit-per-channel texture maps do not encode luminance linearly. Any 8-bit image that looks good when viewed on your computer monitor has, by definition, some kind of nonlinear gamma encoding baked into it. There are various standards, but all of them encode luminance roughly as the square of the 8-bit pixel value, after normalizing it into the range 0-1. A RenderMan shader that makes use of an 8-bit texture map must reverse this nonlinear encoding immediately after the texture lookup in order to produce correct results. (if you find that this is not the case, then you are not gamma-encoding your renderings correctly - see below for guidance!)

Here is some SL code you can use to interpret nonlinear values from 8-bit texture maps:

```
// decode gamma like this:
float tex = texture(...);
tex = cpow(tex, gamma); // where 'gamma' is typically ~2.0

// or, if you want to use the standard sRGB encoding
tex = sRGB_decode(tex);
```

Helper functions

```
// exponentiation for colors
color cpow(color c; float p)
```

```
{
    color d;
    setcomp(d, 0, pow(comp(c,0), p));
    setcomp(d, 1, pow(comp(c,1), p));
    setcomp(d, 2, pow(comp(c,2), p));
    return d;
}

// decode from sRGB luma to linear light
float sRGB_decode_f(float f)
{
    float lin;
    if(f <= 0.03928)
        lin = f/12.92;
    else
        lin = pow((f+0.055)/1.055, 2.4);
    return lin;
}

color sRGB_decode(color c)
{
    color d;
    setcomp(d, 0, sRGB_decode_f(comp(c,0)));
    setcomp(d, 1, sRGB_decode_f(comp(c,1)));
    setcomp(d, 2, sRGB_decode_f(comp(c,2)));
    return d;
}
```

Some non-RenderMan renderers play fast and loose with gamma encoding, improperly mixing linear and gamma-encoded quantities. While the results are sometimes passable, lighting controls and post-render compositing will never behave as predictably as in a completely linear environment. If you want to do things right, always shade in linear light. Only apply gamma encoding during final output to an 8-bit format.

By the way, if you plan to render images in linear light, you'll want to store them in a high-dyamic-range format, like floating-point TIFF or OpenEXR. The RIB command `Quantize 0 0 0 0` instructs the renderer to output floating-point pixels. Or, if you are rendering directly to an 8-bit format, apply the proper gamma encoding with the `Exposure` command (e.g. `Exposure 1.0 2.2`).

More than one shadow map per light

In RenderMan shading language you can call the shadow() function as many times as you want, so you aren't limited to only one shadow map per light. You can combine maps by taking the maximum value returned by each of the shadow() calls. This capability is useful when a single light source illuminates a wide area, but you

only need shadows in a few small, disconnected regions. A single shadow map would be impractically large, but several smaller maps could cover all the important areas.

I recommend against using multiple shadow maps that actually overlap. While it seems to work at first glance, you will inevitably see some artifacts along the edges where one map meets another. I'd be interested to know if anyone has come up with an artifact-free method for handling overlapped shadow maps!

Texture Filtering

The texture() function in RenderMan shading language is very powerful. Instead of simply sampling the texture at a single point, it actually filters the texture over an arbitrary quadrilateral area, which is great for preventing aliasing. But there are a few caveats to texture filtering:

When you call texture() with the coordinates of just a single point, RenderMan attempts to compute an appropriate filter area by looking at how the coordinates vary over a single micropolygon. This estimate is often too conservative; unless your texture map consists of a single-pixel checkerboard of maximum and minimum intensities, you can usually get away with a smaller filter area, giving your surface a sharper look without introducing aliasing. This counteracts the tendency of RenderMan's texturing to look slightly blurry compared to other renderers. It's very easy to do: just pass a value less than 1.0 as the "width" parameter. I find 0.7-0.8 usually works fine.

Any time you have a texture that repeats across a surface, remember to specify the "periodic" wrap mode for s and t coordinates where appropriate. Forgetting this step will result in an ugly seam at the edge of the texture.

Discontinuity in texture coordinates can also be a cause of seams. This frequently happens when using cylindrical and spherical projections. One of the texture coordinates jumps all the way from 1.0 back down to zero across a single micropolygon. This confuses the texture() function into filtering across the entire width of the image, so you get a single abberant row of micropolygons with the average color of the entire texture.

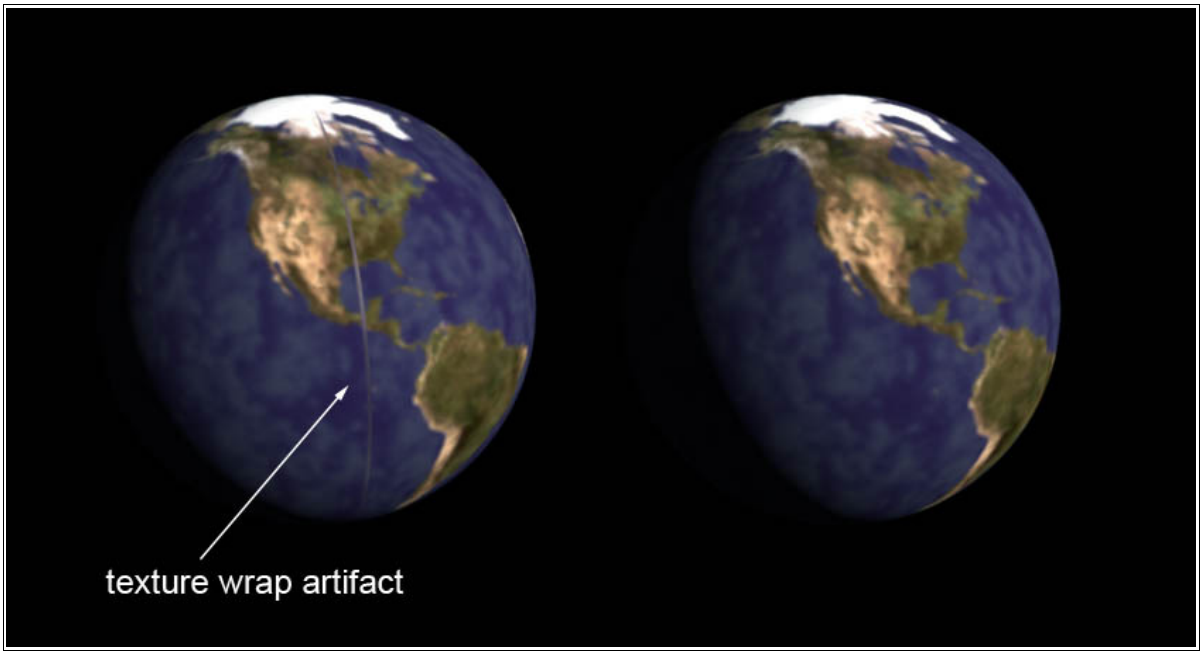


Image 15: Corrected texture lookup on seams

In some cases, like simple projections, it's easy to detect the wrapping situation and compute the correct filter width analytically. If that's impossible, or you're lazy like me, you can make a "quick fix" simply by clamping the texture filter area to a fixed maximum. This may reduce rendering speed and introduce some aliasing, but it is guaranteed to fix the artifacts.

Another solution, which applies to polygons and subdivison surfaces, is to use facevarying st coordinates, which will ensure that the seam lies cleanly on a micropolygon boundary.

```
// This code behaves exactly like the standard one-point texture()
// function, except it limits the maximum filter width to 'maxwidth'.
color clamped_texture(string texturename; float ss, tt, width, maxwidth)
{
  // calculate change in ss and tt across the micropolygon
  float ds_u = Du(ss) * du;
  float ds_v = Dv(ss) * dv;
  float dt_u = Du(tt) * du;
  float dt_v = Dv(tt) * dv;

  // scale coordinate deltas by filter width multiplier
  ds_u *= width; ds_v *= width;
  dt_u *= width; dt_v *= width;

  // clamp to maxwidth
  ds_u = clamp(ds_u, -maxwidth, maxwidth);
```

```
    ds_v = clamp(ds_v, -maxwidth, maxwidth);
    dt_u = clamp(dt_u, -maxwidth, maxwidth);
    dt_v = clamp(dt_v, -maxwidth, maxwidth);

    // find lower edge of filter region
    float ss2 = ss - (0.5*ds_u + 0.5*ds_v);
    float tt2 = tt - (0.5*dt_u + 0.5*dt_v);

    // four-point texture call
    return color texture(texturename, ss2, tt2,
                         ss2 + ds_u, tt2 + dt_u,
                         ss2 + ds_u + ds_v, tt2 + dt_u + dt_v,
                         ss2 + ds_v, tt2 + dt_v,
                         "filter", "gaussian", "lerp", 1);
}
```

This code sample also shows you how to duplicate the filter size computations performed by the one-point texture() function -- just ignore the clamp() statements.

By the way, always call texture() with the extra options "filter" "gaussian" and "lerp" 1. These select a high-quality filter and enable interpolation of mip-map levels. You'll get better image quality with only a tiny speed hit. Some renderers offer other high-quality filters beyond "gaussian" - check your documentation.

Implementing ambient occlusion—light source or surface shaders?

Ambient occlusion coupled with environment-based ambient illumination is a powerful technique for photorealistic lighting. While the basic idea is simple, some planning is necessary to implement ambient occlusion cleanly within a RenderMan-based pipeline.

Ambient occlusion involves two essential steps: first, computing the occlusion (and possibly caching it for later re-use), and second, looking up the ambient illumination in an environment map. Both of these steps have been covered in detail in a previous RenderMan course [Landis 2002] and in Pixar's application notes [Pixar 2004]. Here I will only discuss a shading pipeline issue: does ambient occlusion belong in light source shaders or surface shaders?

Both choices have advantages and disadvantages. The light source is a convenient place to compute occlusion and look up environment maps, since you don't need to modify every surface shader in your scene when adding ambient occlusion. However, it divorces the occlusion computation from your scene hierarchy, making it harder to cache occlusion on a per-object basis or to isolate certain objects from expensive occlusion computations.

Performing occlusion in surface shaders fixes these difficulties, but presents the problem of duplicating parameters like light intensity and environment map textures across many surfaces.

Due to the tight interaction between surfaces and lights, no matter which option you choose some amount of behind-the-scenes message passing will be necessary to get the shader information where it needs to go. (In fact, message passing blurs the distinction between surface and light source shaders to the point where it is feasible, although inconvenient, to implement ambient occlusion entirely in one or the other.) I recommend a split approach: compute the occlusion in surface shaders, then look up the environment maps in light shaders. This technique is quite flexible and can be implemented with a minimum of message-passing magic.

I usually compute ambient occlusion on a per-object basis, to allow for parameter tweaks on individual surfaces and to take advantage of object-space caching. All this computation boils down to determining two quantities inside the surface shader: Ka, the coefficient that multiplies the ambient lighting contribution, and what Landis calls the "bent normal," or the average unoccluded direction vector. Ka and the bent normal are computed either by ray tracing or by looking up values in a pre-computed per-object cache. One can also use image-space caching, as described in Landis' paper, although that technique would probably be simpler to implement in a light source shader. On background objects that do not need the full expense of ray-traced ambient occlusion, you can simply set Ka to a fixed value, or use some kind of heuristic, perhaps surface curvature, to give a cheap approximation of the occlusion.

The environment map lookup is performed by an ambient light source shader that is folded into the standard ambient() function. You can accomplish this by setting L = 0 in the light source shader, which instructs the renderer to treat it as an ambient source. This makes it easy to swap in different environment maps without modifying surface shaders, and you can apply more than one ambient light in the scene without duplicating the occlusion computation.

In order to implement this technique one needs a way to access the bent normal from within the light source shader. If you are computing ambient occlusion within the light source, you'll also need a way of finding the surface normal on the primitive being shaded (this isn't completely trivial: inside a light source shader, N is the normal on the light source, not the surface). Message passing is a good way to accomplish this.

(Pixar's renderer offers a function called shadingnormal() that retrieves the surface normal from within a light source shader, although this does not appear to be documented in the RenderMan standard.)

Here is a framework you can use to implement ambient occlusion as described:

Surface shader

```
surface mysurface(float Kd = 1;
                  float Ka = 1;
                  // this is the bent normal which we must pass to the
                  // light source shader
                  output varying normal bent_normal = normal(0,1,0);)
{
    // compute ambient occlusion (e.g. by ray-tracing or use a cache;
    // see Landis' paper)
    float ambocc = ...;
    normal bent_normal = ...;

    Oi = Os;

    // diffuse lighting term
    Ci = Kd * diffuse(normalize(N));
    // (add specular or other lighting terms here)

    // ambient lighting term
    // the light source shader will access bent_normal via message passing
    Ci += Ka * (1 - ambocc) * ambient();
    Ci *= Oi;
}
```

Light source shader

```
// environment map-based ambient light source
light env_light(string diffuse_tx = "";
float intensity = 1.0;
float gamma = 2.0;)
{
    // act as an ambient light
    L = 0;

    // obtain bent_normal via message passing
    varying normal bent_normal;
    if(!surface("bent_normal", bent_normal)) {
        printf("warning: surface did not pass bent_normal\n");
    }

    // rotate Y-up normal to Z-up for environment()
    // (this makes +Y the "north pole" of the environment map)
    bent_normal = yup_to_zup(bent_normal);

    // query environment map
    color envcolor = color environment(diffuse_tx, bent_normal);

    // gamma-decode
```

```
    envcolor = cpow(envcolor, gamma);
    Cl = intensity * envcolor;
}
```

Supplementary function

```
// transform a Y-up vector to a Z-up coordinate system
// this is handy for the environment() function, which treats +Z as the north
pole
vector yup_to_zup(vector refdir)
{
    return (vector rotate((point refdir),PI/2,point(0,0,0),point(1,0,0)));
}
```

Compositing Tips

Easy rendering of "global" AOVs using atmosphere shaders

RenderMan recently added a great new feature called Arbitrary Output Values (AOVs) that let you output any shader variable as a separate image along with the main rendering.

Let's say you want to output some "global" quantity, like xyz coordinates, from all the objects in your scene. But it's a big job to go and modify each one of your surface shaders to add the AOV.Wouldn't it be great if there were some kind of auxillary shader you could add to every surface, so you wouldn't have to modify each one?

It turns out there is: an atmosphere shader! Remember, in RenderMan an atmosphere shader runs right on every affected primitive right after the surface shader, and it has access to all the surface quantities like P, Oi, and Ci. So just write a single atmosphere shader to output your AOV and apply it to the entire scene. If your AOV isn't among the pre-defined global variables, you can probably obtain it by message-passing from the surface shader.

Note that if you want to assign a particular non-zero value to areas of the image where no foreground objects appear, you'll have to surround the scene with a large surface to give the atmosphere shader something to operate on (I use a large, inward-facing Sphere primitive with an opacity of 0.0001).

Transparency and AOVs: the real story

What happens to AOV values on semi-transparent surfaces? The story is a bit complicated. First, AOVs with the color data type are alpha-composited according to surface opacity, just like the main RGB channels. For other

data types, you can specify different algorithms like “min,” “max,” or “average”— check your renderer documentation for the details.

Remember, if you want your color AOV to be alpha-composited correctly, you have to multiply it by the surface opacity (Oi), just like you do for Ci.

Depth passes

Depth passes are useful for performing all kinds of post-render effects. But there are some subtleties to consider, such as how to handle motion blur and anti-aliasing. Let’s talk about two specific uses of depth passes:

Depth compositing: You are using the depth pass to determine whether one 2D element is in front of another. In this case anti-aliasing and motion blur make no sense; you just want the raw depth value. The best way to do this is to use the new “sub-pixel” output feature, which tells the renderer to give your the depth value for each spatio-temporal sample as an individual pixel in the output image. When you composite two sub-pixel images and the downsample, you’ll get basically the result as if you rendered the elements together. However, note that this doesn’t handle multiple layers of transparent objects; that would require something akin to a “deep depth map,” which hasn’t been standardized yet.

Post-render atmospheric effects: you are using the depth pass to apply fog or other atmospheric effects. In this case you can get away with a normal anti-aliased, motion-blurred render. The results won’t strictly be correct, but as long as your atmosphere changes smoothly with distance, the error should not be noticeable. I find it useful to output not the raw depth value but an “encoded” value consisting of $\exp(-c \cdot \text{depth})$. This handles alpha-compositing and anti-aliasing better than raw depths. In particular, it helps to avoid stair-stepping along silhouette edges where the depth “falls off a cliff” from the foreground to the background, overwhelming the anti-aliasing filter. Also, you don’t have to enclose your entire scene with a surface to set a background depth, since the encoded value for infinite depth is just zero.

Here’s a sample implementation:

```
// apply this as an atmosphere shader to output depth information for
// post-render atmospheric effects
volume get_depth(output varying color depth = 0;
                 // unit_depth = 1 / c
                 // set unit_depth to at least 1/13th the greatest
                 // depth value in the scene
                 float unit_depth = 1000;
                 float enable_warning = 1;)
```

```
{
    // get distance from surface in camera space
    float len = length(vtransform("camera", I));

    // rescale depth
    float x = -len/unit_depth;

    // warn if depth goes out of range
    if((x < -13) && (enable_warning != 0)) {
        printf("get_depth: warning: precision loss - "
               "increase unit_depth by a factor of at least %f\n",
        x/-13);
    }
    x = clamp(x, -13, 0);

    // encode depth
    depth = color(exp(x));

    // check for the user fog intensity attribute
    uniform float fog_amount = 1;
    attribute("user:fog_amount", fog_amount);
    depth = mix(color(1), depth, fog_amount);

    // multiply by Oi for proper alpha compositing
    depth *= Oi;
}
```

This shader checks for an optional user attribute called “fog amount” which modulates the encoded depth values, and thus the intensity of the atmospheric effect. To make use of this encoded depth pass in your compositing application, just invert the color values so that zero depth is black and infinite depth is white, then modulate your atmosphere with the resulting image.

This assumes an exponential falloff with depth which is physically correct for fog-like effects. You can control the depth scaling by applying a gamma operator to the encoded depth values, since exponentiating the encoded depth is the same as multiplying the raw depth by a constant. If you need to recover the raw depth values, just invert the exponential encoding.

Another common way to extract depth is to encode it as the hue of an HSL color. This has the advantage of returning an accurate depth value for motion-blurred edges.

References

BEIER, T., 1999. Deforming bumpy polygonal models. Usenet: comp.graphics.rendering.renderman, January.
LANCASTER, T., 2000. Rendering related issues on the production of Disney's Dinosaur.
ACM SIGGRAPH 2000 Course #40 Notes, July.
LANDIS, H., 2002. Global illumination in production. ACM SIGGRAPH 2002 Course #16 Notes, July.
PIXAR, 2004. Application note #35: Ambient occlusion, image-based illumination, and global illumination.
RenderMan Pro Server documentation.

Going Mad With Magic Lights

Moritz Mœller - Rising Sun Pictures

'Magic' lights have been around for a while. The term 'magic light' for a light that does other things than what the term *light* suggests in the first place, was introduced by Pixar a long time ago. Their *Alias to RenderMan* plug-in (*AtoR*) shipped with a few of those and they still can be found in *MtoR* nowadays.

However, there has never been any explanation on how they work and while it might be something obvious for RenderMan old hands, I quite well remember how puzzled I was by my first encounter with them almost a decade back.

This course will give you an overview of the concepts or the 'magic' behind those lights and illustrate their actual use in production with some examples.

What Lights Really Are

We know that surface shaders are essentially small (or nowadays often rather large) programs that get run by the renderer on geometric primitives.

So instancing a surface shader on a primitive guarantees its execution if that primitive or part of it is inside the camera's viewing frustum. Light shaders are different in that their invocation is triggered by the surface shaders based on a geometric criteria. Lights could be seen as subprograms or functions that the surface shader program calls.

The geometric criteria is the intersection of two cones, as specified to in the `illuminate()` resp. `solar()` and `illuminance()` constructs. An implication of this is that instancing a light doesn't guarantee its execution if there are no surface shaders in the scene that have any implicit (calling `diffuse()`, `phong()` or `specular()`) or explicit `illuminance()` loops. Examples of such shaders are the *defaultsurface* and *constant* standard shaders. Another pitfall are ambient lights because they don't have an illuminate or olar loop at all. We shall see later thiugh, that there is an easy way around this which gurantees their execution and still has them behave as ambient lights.

From a computer science point of view, we just have two fragments of code. The first fragment (the i`lluminance()` loop) triggers invocation of the second one (the light source shader's `illuminate()` or `solar()` construct). With more than one light source shader instance we have the first fragment trigger the invocation multiple times and with different numbers of light source shaders we have the `illuminance()` loop call different code fragments (multiple times).

The basic idea is to look at the whole thing from an abstract perspective. Read: one function calls another and both functions can do message passing back and forth to exchange information.

This goes beyond an ordinary function call where ones passes parameters in and has the function return values — lights can ask the surface about parameters and vice versa, anywhere in those constructs.

Message Passing

All RenderMan shader types can pass messages among them. Each primitive can have a multitude of shader types attached to it, but only one instance per shader type; except for lights — their number is unlimited.

The syntax to get the contents of any variable that another shader type attached to the primitive has evaluated, is:

```
shadertype("variableName", destinationVariable);
```

For example, if the variable whose value we are interested in was a vector called `warpDirection` which had been calculated by a *displacement* shader and we wanted to store it in a local variable called `dispWarpDir`, we would use a code fragment like this in our shader:

```
vector dispWarpDir = vector "world"(0, 0, 1);
displacement("warpDirection", dispWarpDir);
```

The message passing call to `displacement()` will only alter that value of the passed-in variable, however, if that variable exists in the destination shader. Read: if `warpDirection` was not declared in the displacement shader attached to the current primitive, the call to `displacement()` would not alter the value of `dispWarpDir` at all.

The above code snippet makes use of this fact and assigns a default value to `dispWarpDir` before calling `displacement()`. That is a good practice and you should stick to it.

Let's look at a more defined example:

```
surface myFancySurface(...) {
    ...
    float externalDispStrength = 0;
    displacement("dispStrength", externalDispStrength);
    color Cramp = spline(dispStrength, ...);
    ...
}
```

The a typical message passing call in the surface counterpart looks something like this:

```
displacement myFancyDisplacement() {
    ...
    float dispStrength = veryExpensiveFunction(P);
    ...
}
```

The idea is that we want to use the value that `veryExpensiveFunction()` returns in both shaders. We use it somehow in our displacement strength calculation in the displacement shader and we use it to also look up a `spline()` in the surface shader to color the surface. However, if we did not use message passing, we would have to evaluate `veryExpensiveFunction()` again in the surface shader. That isn't very efficient and it also requires to recompile not only two shaders instead of one when altering the function but also to move it into an external header file if we wanted to make sure that editing its inner workings affected both shaders.

Being able to share calculations between different shader types is one of the main reasons why shader message passing was introduced, originally. Suffice to say that people quickly found ways to do things with it, that its inventors likely had not thought of.

Now putting our thinking caps on, what can we do with this that goes beyond what lights already do by default?

Abusing Lights — Turning Ordinary Into Magic Lights

Since lights have access to almost the same set of predefined variables as surface or displacement shaders, we can do stuff with them that that commonly is done in the former two types of shaders.

Instead of using lights to shine light, we use them to 'shine' other kinds of information that we interpret in many (generally spoken: arbitrary) ways. We call these lights **'magic lights'**.

Here is a very basic magic light:

```
light magicColorLight(
    output uniform color  magicColor   = 1;
    output uniform float  magicOpacity = 1;
) {
    solar(vector "shader"(0, 0, 1), 0) {
        /* Calculate distance to light axis to create a disk pattern */
        point Pshader = transform("shader", Ps);
        float x = xcomp(Pshader);
        float y = ycomp(Pshader);
        float distance = sqrt(x * x + y * y);
        /* Create a disk, pre-multiply color */
        color __magicOpacity = magicOpacity * filterstep(1, distance);
        color __magicColor   = __magicOpacity * magicColor;
        /* Make sure we don't add any light */
        Cl = 0;
    }
}
```

And its matching surface shader part:

```
surface magicColorSurface() {
    color totalColor = 0;
    color totalOpaciy = 0;
    illuminance(P) {
        color magicColor = 0;
        float magicOpacity = 0;
        /* Get the color & opacity from the magic light */
        lightsource("__magicColor", magicColor);
        lightsource("__magicOpacity", magicOpacity);
        /* Compose this color *Over* our current color */
        float backgroundOpacity = 1 - magicOpacity;
        totalColor  = backgroundOpacity * totalColor  + magicColor;
        totalOpaciy = backgroundOpacity * totalOpaciy + magicOpacity;
    }
    Oi = totalOpaciy;
    Ci = totalColor;
}
```

Using this combination, we paint layer of layer of color dots on an object that uses the `magicColorSurface` shader. Think of solid-type single color layers in Photopshop. These two shaders are admittedly of not much practical use, but they should lay the basic principle out quite clearly.

The light declaration order in the RIB dictates the order in which our colors get layered over each other.

This is an important detail: 3Delight and PRMan guarantee that the evaluation order in the `illuminance()` loop! Is the same as the declaration order of the lights in the RIB!

While this is not documented or part of the RI spec., it is a known fact and we take it for granted here in. Most of the examples in the chapter will not work as expected if you try to use them with a renderer that doesn't guarantee that light instantiation order equals light evaluation order at render time.

You might have noticed that the light source shaders commonly use `Ps` as opposed to `P`. The reason is that `P` in a light is reserved for the surface position of an area light. 3Delight also defines `Ns` to be the normal of the surface and N to be the normal of the light emitting primitive at the point for which the area light is run, in the area light case. This is not RI spec. compliant though.

Using Magic Lights to Displace

As hinted at in this chapters introductory code snippets, one of the most fun things to do with lights is using them for displacement. Yes, you read correctly, for displacement!

Instead of accumulating light intensities in our `illuminance()` loop, we accumulate a displacement strength that we then use in our surface[1] shader.

Let's look at some shader code:

1 It is bad style to do displacement in a surface shader. That's what displacement shaders are for and there are good reasons to keep both separate. We only do this here to merely keep the examples as short and readable as possible.

```
light magicDisplacementLight(
           uniform float  Strength           = 1;
           uniform float  Radius             = 1;
           uniform string __category         = "magicdisplacement";
    output varying float  __magicDisplacement = 0;
) {
    /* Behave like a pointlight */
    illuminate(point "shader"(0, 0, 0)) {
        __magicDisplacement = Strength * (1 - smootherstep(0, Radius,
length(L)));
        Cl = 0;
    }
}
```

This shader calculates a `__magicDisplacement` value that we will later use in the surface shader to do displacement.

The only thing that should need explanation here is the use of a function called `smootherstep()` (note the 'er'). As the name suggests, this function is similar to the build-in `smoothstep()`. However, it has some properties that make it preferable over the latter when it comes to displacement and the resulting shading of the displaced surface. Under the hood this function, defined in *helpers.h*, is Ken Perlin's `smoothstep()` replacement from [3].

The matching surface shader looks like this:

```
#include "displace.h"

surface magicDisplacementSurface(
    output uniform float  Diffuseness²□□= 0.5;
           uniform float  Specularity = 0.5;
           uniform float  Glossiness  = 10;
) {
    normal Nn = normalize(N);

    float summedMagicDisplacement = 0;
    illuminance("magicdisplacement", P) {
        float magicDisplacement = 0;
        lightsource("__magicDisplacement", magicDisplacement);
        summedMagicDisplacement += magicDisplacement;
    }
    normal Nf = Displace(vector(Nn), "world", summedMagicDisplacement, 1);
```

2 I decided to abandon RenderMan nomenclature and give the parameters more artist-friendly names. My experience is that outside the literature, Kd and Ks don't get used much anyway because non-adepts in things RenderMan don't have a clue what they mean and studios commonly don't want to invest in training people in that sort of stuff, if they are not TDs. The default 'roughness' parameter is better suited to define the roughness of an Oren-Nayar diffuse term, I believe. Hence the switch to it's suggested reciprocal 'Glossiness' to define that parameter to specular(). The reason for capitalization is that I commonly use this to distinguish between parameters that should be available to the user. Any parameter that starts with a lowercase letter or underscore doesn't wouldn't show up in the user interface for this shader if it was built by a tool I wrote.
Call me a rebel.

```
    Nf = faceforward(Nf, I, Nf);

    Oi = Os;
    Ci = Oi * (Diffuseness * pickyDiffuse(Nf) +
               Specularity * pickySpecular(Nf, normalize(-I), 1/Glossiness));
}
```

This shader sums the `__magicDisplacement` values from all magic displacement lights and displaces the surface using the `Displace()` function defined in *displace.h*[3] from [2].

Note that we started using light categories in this example. Recall that `illuminance()` takes an optional first argument, a light 'category', that is used to limit it to calling only lights who declare a parameter called `__category` and whose value matches this argument.

The reason categories are introduced here is plain performance. If we called the regular `diffuse()` and `specular()` functions, these would call our displacement lights in their respective `illuminance()` loops and waste time evaluating them again —they can't 'know' that these don't cast any light.

We hence define two new functions, `pickyDiffuse()` and `pickySpecular()`, that behave exactly like the built-in ones, with the exception that they don't call any lights that have a non-empty category. This is accomplished using a simple wildcard syntax. The `pickyDiffuse()` function e. g. looks like this:

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

Now we can use some very simple geometry to get some rather interesting results. Below is a RIB that puts everything together. Note the use of negative 'Strength' in one *magicDisplacementLight* instance, to put a deep valley into the resulting surface. This is the displacement equivalent of a light with negative intensity.

```
Display "magicDisplacement.tif" "tiff" "rgba"
PixelSamples 3 3
```

3 http://www.renderman.org/RMR/Books/arman/arman-shaders.tar.gz

```
# Camera
Projection "perspective" "fov" [ 41 ]
Transform [ 0.74457998 -0.42889986 -0.51151306 0 -6.777348e-16 0.76627343
-0.6425146 0
    -0.6675333 -0.47840347 -0.5705519 0 -0.31376336 2.6090056 24.540062 1 ]

WorldBegin
  # Ordinary lights
  Transform [ 0.85236544 0.50748367 -0.12622791 0 -0.43932875 0.56397056
-0.6992335 0
      0.28366073 -0.65145796 -0.7036612 0 11.0775147 -5.0834957 -1.1839303 1 ]
  LightSource "distantlight" "key" "float intensity" [ 1 ]

  Transform [ -0.7954117 -0.27529154 0.53993965 0 -0.4642206 0.84948384
-0.2507517 0
      0.38964016 0.45010195 0.8034854 0 10.0775147 -6.0834957 -2.1839303 1 ]
  LightSource "distantlight" "rim" "float intensity" [ 1 ]
    "float __nonspecular" [ 1 ]

  # Magic displacement lights
  Transform [ 2 0 0 0 0 2 0 0 0 0 -2 0 3.7961976 0.46999147 0 1 ]
  LightSource "magicDisplacementLight" "dent0" "float Strength" [ -2 ]
    "float Radius" [ 2 ]

  Transform [ 3 0 0 0 0 3 0 0 0 0 -3 0 2.9825936 0.4094044 1.8328775 1 ]
  LightSource "magicDisplacementLight" "dent1" "float Strength" [ 1 ]
    "float Radius" [ 3 ]

  Transform [ 5 0 0 0 0 5 0 0 0 0 -5 0 -1.5186014 1.9849439 1.8328775 1 ]
  LightSource "magicDisplacementLight" "dent2" "float Strength" [ 2 ]
    "float Radius" [ 5 ]

  Transform [ 7 0 0 0 0 7 0 0 0 0 -7 0 1.1336353 0.65621733 -1.943809 1 ]
  LightSource "magicDisplacementLight" "dent3" "float Strength" [ 6 ]
    "float Radius" [ 7 ]

  Transform [ 5 0 0 0 0 5 0 0 0 0 -5 0 1.1336353 0.65621733 -1.943809 1 ]
  LightSource "magicDisplacementLight" "dent4" "float Strength" [ -10 ]
    "float Radius" [ 5 ]

  # Geometry
  Identity
  Attribute "displacementbound" "string coordinatesystem" [ "world" ]
  Attribute "displacementbound" "float sphere" [ 10 ]
  Surface "magicDisplacementSurface"
  Patch "bilinear" "vertex point P" [-8 0 8 8 0 8 -8 0 -8 8 0 -8 ]

WorldEnd
```

And here is the image this produces (Figure 1). I rendered it twice, once with a modified *magicDisplacementLight* shader that uses the built-in `smoothstep()` function. The shading discontinuities in that version should make it quite obvious now, why we used Ken Perlin's version of that function.

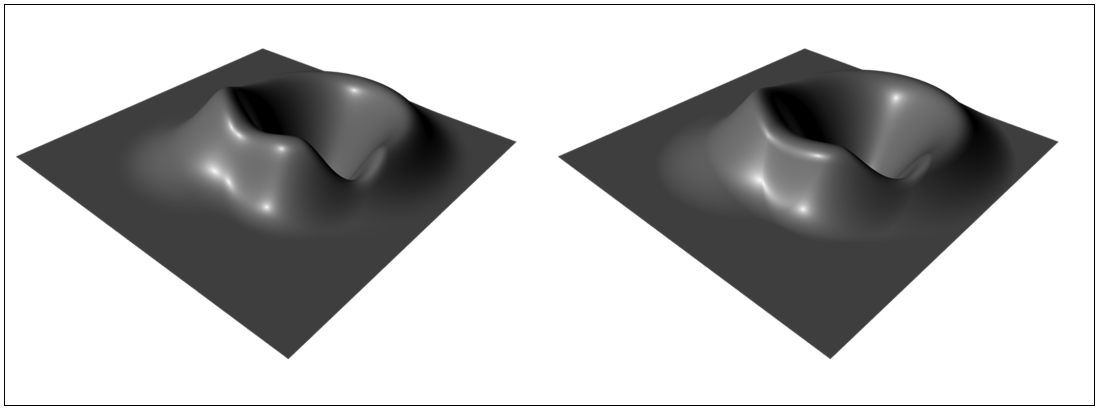


Image 16: Using five magicDisplacementLight instances to imprint a crater-like shape on a bilinear patch.(a) using our own smootherstep(), (b) using good old smoothstep()

In the last section we displaced geometry in the surface shader using an accumulated value from all magic lights. What if we displaced the surface for each light separately? Would that give us a different result? The answer is: quite likely! Imagine that after each displacement we get a new normal. Now displacing again, we use that normal altered normal from the previous displacement. This requires only a very minor change in our magicDisplacementSurface shader:

```
...
illuminance("magicdisplacement", P) {
    ...
    Nn = Displace(vector(Nn), "world", summedMagicDisplacement, 1);
}
normal Nf = faceforward(Nn, I, Nn);
...
```

We simpy move the line that displaces inwards, so the next light uses the new normal Displace() returned.

We can easily create horns and other non-convex features on a surface using these technique. Furthermore, if lights take into account the angle to the surface or a vector tangent to that to bias the displacement direction, we can use lights as magnets that bend, twist or do other deformations to already existing displacement that magic lights, having run before them, have done. This has its limits though. The μ-polygons get torn and twisted and chances are they are far off shading rate by the time our shaders are done with them.

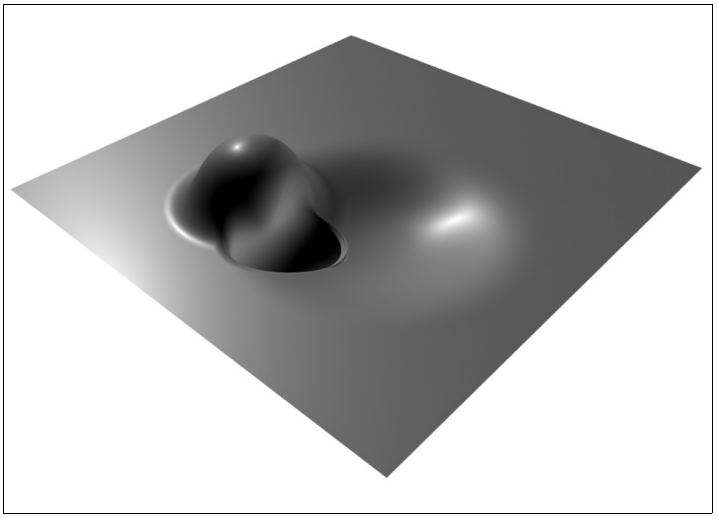


Image 17: Patch displaced using accumulated changes to the displacement direction. Note the concave sag and the surface wrinkling on itself.

Using Magic Lights to Alter Surface Shading & Displacement

Now putting this together, lets create something more useful. Below is a bullet hole shader that puts holes into a surface and displaces it inward away from the hole so that it seems to have a thickness.

```
#include" helpers.h"

light magicBullet(
                uniform floatBulletSize                     = 1;
    output varying      float  __magicOpacity               = 0;
    output varying      float  __magicDisplacement      = 0;
    output varying      vector __magicDisplacementDir   = 0;
) {
    float bulletRadius = BulletSize / 2;
    /* Calculate an ID for our bullet to use with pnoise() later */
    float bulletId = 10 * cellnoise(transform("shader", "world", point(0.1, 0.1,
0.1)));
    solar(vector "shader"(0, 0, 1), 0) {
        /* Get distance and angle to the bullet path for current point */
        point Pshader = transform("shader", Ps);
        float x = xcomp(Pshader);
        float y = ycomp(Pshader);
        float distance, theta;
        normalizedPolar(x, y, distance, theta);
        /* Add a little bit of angular noise */
        distance += BulletSize * 0.1 * (pnoise(bulletId + theta * 10, 10) - 1);
        __magicOpacity = filterstep(bulletRadius, distance);
        /* Distance to the edge of the bullet hole */
        float bulletHoldEdge = distance - bulletRadius;
        __magicDisplacement = 0.5 * pow(1 - linearStep(0, BulletSize,
```

```
bulletHoldEdge), 3);
        __magicDisplacementDir = L;
        Cl = 0;
    }
}
```

The matching surface shader is almost the same as the *magicDisplacementSurface* from the last section; it just takes opacity from the magic light into consideration.

```
#include "brdf.h"
#include "displace.h"

surface magicBulletSurface(
                uniform      float  Diffuseness                    = 0.5;
                uniform      float  Specularity                    = 0.5;
                uniform      float  Glossiness                     = 10;
) {
    normal Nn = normalize(N);
    normal Nf = faceforward(Nn, I, Nn);

    float totalOpacity = 1;
    float totalDisplacement = 0;
    vector totalDisplacementDir = 0;
    illuminance( P, Nf, PI/2 ) {
        float magicOpacity = 1;
        lightsource("__magicOpacity", magicOpacity);
        totalOpacity = min(totalOpacity, magicOpacity);
        float magicDisplacement = 0;
        lightsource("__magicDisplacement", magicDisplacement);
        if(totalDisplacement < magicDisplacement) {
            totalDisplacement = magicDisplacement;
            vector magicDisplacementDir = 0;
            lightsource("__magicDisplacementDir", magicDisplacementDir);
            totalDisplacementDir = magicDisplacementDir;
        }
    }
    Nf = Displace(totalDisplacementDir, "world", totalDisplacement, 1);

    Oi = Os * totalOpacity;
    Ci = Oi * (Diffuseness * pickyDiffuse(Nf)
            + Specularity * pickySpecular(Nf, normalize(-I), 1/Glossiness));
}
```

A RIB with two ‘shots’:

```
Display "magicBullet.tif" "tiff" "rgba"
PixelSamples 3 3

# Camera
Projection "perspective" "fov" [ 40 ]
Transform [ 0.91010595 -0.3496773 -0.22233516 0 6.1322713e-10 0.5365547
```

```
-0.8438656 0
  -0.41437558 -0.7680071 -0.4883216 0 0.5603471 0.6653321 13.629389 1 ]

WorldBegin  # {
  # Ordinary lights
  Transform [ 0.97997517 0 0.1991197 0 0.13710144 0.7252004 -0.67474998 0
    0.1444017 -0.68853783 -0.7106784 0 0 0 0 1 ]
  LightSource "distantlight" "key"

  Transform [ 0.9999286 -3.8114063e-17 -0.0119498997 0 -0.0035433948 -0.9550264
-0.2964997 0
    0.01141247 -0.2965209 0.9549582 0 0 0 0 1 ]
  LightSource "distantlight" "rim"
      "float __nonspecular" [ 1 ]
      "float intensity" [ 0.2 ]

  # Magic displacement lights
  Transform [ 0.9627261 -0.22942821 -0.14325182 0 -0.2521274 -0.5694557
-0.78240145 0 -0.0979294 -0.78935606 0.606075 0
        1.90821 0 1.4780138 1 ]
  LightSource "magicBullet" "bullet1"

  Transform [ 0.98275686 0.116909855 -0.14325182 0 -0.08911498 -0.37934948
-0.92095197 0 0.16201086 -0.91783773 0.36238987 0
        -2.0679639 0 -1.1101599 1 ]
  LightSource "magicBullet" "bullet2"

  # Geometry
  Identity
  Attribute "displacementbound" "string coordinatesystem" [ "shader" ] "float
sphere" [ 2 ]
  Surface "magicBulletSurface"
  Patch "bilinear" "vertex point P" [-4 0 4 4 0 4 -4 0 -4 4 0 -4 ]
WorldEnd
```

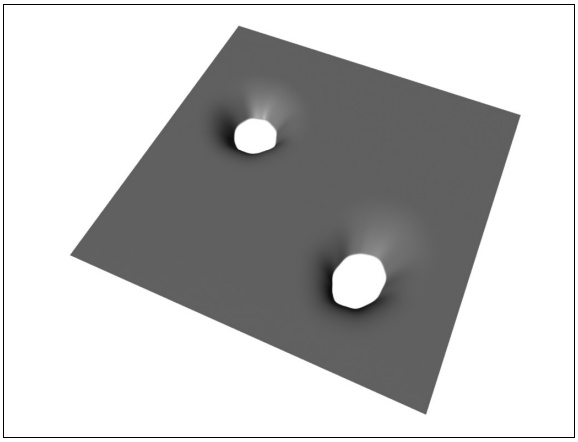


Image 17: Bullet holes' cast by lights.

Instead of the surface normal, we use the light direction to displace. That demonstrates the possibility mentioned at the end of the last section about displacement in directions other than the surface normal.

Also note that we call `filterstep()` in the light first, then displace in the surface shader, thereby changing the filter size and thus getting an inferior result than if we did the `filterstep()` call for the opacity calculation in the surface shader after displacing. But since we want to encapsulate as much into our lights as possible, we ignore this little glitch.

Magic Lights For Texturing

Texturing is the original application that magic lights were used for in Pixar's *AtoR*. We use directional lights to project texture maps ('decals') onto surfaces and layer them in light execution order from bottom to top. This extends the first example shader in the chapter to something more useful.

The magic decal light:

```
#include "helpers.h"
light magicDecalLight(
    /* User parameters */
           uniform color  DecalColor            = 1;
           uniform string DecalMap              = "";
           uniform float  DecalScaleX           = 10;
           uniform float  DecalScaleY           = 10;
           uniform float  DecalFlipX            = 0;
           uniform float  DecalFlipY            = 1;
    /* Category */
    output uniform string __category            = "magicdecal";
    /* Output parameters */
    /* Default color black */
    output varying color  __CmagicColor         = 0;
    /* Default alpha full opaque */
    output varying float  __magicOpacity        = 1;
)
{
    solar(vector "shader"(0, 0, 1), 0) {
        uniform float channels = numTextureChannels(DecalMap);

        /* If this texture exists */
        if(0 < channels) {
            /* Get P in light space */
            point Plight = transform("shader", Ps);

            /* Calculate horizontal texture coordinate */
            float sDecal = remapCoordinate(xcomp(Plight), DecalScaleX,
DecalFlipX);

            /* Calculate vertical texture coordinate */
            float tDecal = remapCoordinate(ycomp(Plight), DecalScaleY,
DecalFlipY);

            /* Make sure we assign a default if the DecalMap gets used as Opacity
*/
            __magicColor = DecalColor;
            getColorAndOpacity(DecalMap, sDecal, tDecal,
                               __CmagicColor, __magicOpacity);
        } else {
```

```
            /* Make sure we don't add any color if the DecalMap is not valid */
            __magicOpacity = 0;
            /* Just in case the shader instance overwrote the default */
            __CmagicColor = 0;
        }
        /* Make sure we don't add any light */
        Cl = 0;
        Ol = 0;
    }
}
```

The function `numTextureChannels()` is defined in *helpers.h*. It is more clever than the plain old `if("" != texturename)` test. This test returns true if the string is non empty, but doesn't actually tell you if the texture exists, physically. The function returns the number of channels present in the given texture file or 0 if the texture doesn't have any channels (which means the resp. file doesn't exist or is invalid) or if the texture file name string it got passed was empty.

The surface counterpart for this magic light is almost as simple as the initial example in this chapter:

```
#include "brdf.h"
#include "helpers.h"
surface magicDecalSurface(
          uniform float  Diffuseness          = 0.5;
          uniform float  Specularity          = 0.5;
          uniform float  Glossiness           = 10;
) {
    normal Nn = normalize(N);
    normal Nf = faceforward(Nn, I, Nn);

    color Cdecal = 0;
    illuminance("magicdecal", P) {
        /* Get the opacity and multiply the background down */
        float magicOpacity = 0;
        lightsource("__magicOpacity" , magicOpacity);
        Cdecal *= 1 - magicOpacity;

        /* Get the (pre-multiplied) color and add it */
        color CmagicColor = 0;
        lightsource("__CmagicColor" , CmagicColor);
        Cdecal += CmagicColor;
    }

    Oi = Os;
    Ci = Oi * (Cdecal * Diffuseness * pickyDiffuse(Nf) +
              Specularity * pickySpecular(Nf, normalize(-I), 1/Glossiness));
}
```

We put everything together in a little RIB:

```
Display "magicDecal.tif" "tiff" "rgba"
PixelSamples 3 3

# Camera
Projection "perspective" "fov" [ 40 ]
Transform [ 0.89384144 0.37837515 0.24058211 0 7.558656e-9 0.5365547 -0.8438656 0
  0.4483832 -0.754282 -0.4795948 0 -0.33846625 1.3288873 15.698347 1 ]

WorldBegin  # {
  # Ordinary lights
  Transform [ 0.8087541 0.11217631 -0.57735025 0 -0.53797985 0.5378031
-0.64911134 0
    -0.23768586 -0.8355742 -0.49529905 0 1 1 1 1 ]
  LightSource "distantlight" "key"

  # Magic decal lights
  Transform [ 1 0 0 0 0 0 -1 0 0 -1 0 0 0 0 0 1 ]
  LightSource "magicDecalLight" "decal1"
    "string DecalMap" [ "brickColor.tif" ]
    "float DecalScaleX" [ 10 ]
    "float DecalScaleY" [ 10 ]
    "float DecalFlipY" [ 0 ]

  Transform [ 0.82074975 -0.012851611 0.57114326 0 0.51356954 0.45450153
-0.72778756 0
    0.25023227 -0.89065326 -0.37963224 0 1 1 1 1 ]
  LightSource "magicDecalLight" "decal2"
    "color DecalColor" [ 1 1 1 ]
    "color DecalOpacity" [ 0.867 0.867 0.867 ]
    "string DecalMap" [ "shadowFaces.tif" ]
    "float DecalScaleX" [ 8 ]
    "float DecalScaleY" [ 8 ]

  Transform [ 0.6873237 0.0272209 0.72584104 0 0.7175644 0.12951945 -0.6843435 0
    0.11263897 -0.9912032 -0.06948903 0 -1.2446777 1 -0.8752074 1 ]
  LightSource "magicDecalLight" "decal3"
    "color DecalColor" [ 0.02 0 0.29 ]
    "color DecalOpacity" [ 0.863 0.867 0.686 ]
    "string DecalMap" [ "badBunny.tif" ]
    "float DecalScaleX" [ 6 ]
    "float DecalScaleY" [ 6 ]

  # Geometry
  Identity
  Surface "magicDecalSurface" "float Diffuseness" [ 1 ]
  Patch "bilinear" "vertex point P" [-5 0 5 5 0 5 -5 0 -5 5 0 -5 ]

WorldEnd  # }
```

Image 18: Three texture maps projected through magic decal lights.

One could argue that you can do the same thing with traditional texture map painting, aligning your maps in an image editing application. This requires texture coordinates to be present though or, if absent, to be created 'somehow'. It also prone to wastes a lot of texture space in most circumstances. Note how small the bunny is in respect to the surface it gets projected on. If we had everything to be pre-aligned in the texture space of the bicubic patch, a lot of space in those two stencil graffiti decals would be empty.

It also is less convenient as you can't know how well the alignment, size and color of the decals will work in the rendered image. Magic lights are a lot more immediate in that respect. With a well designed magic light rig in one's 3D package of choice, a look artist has a good idea of how things will come out at the end, without being forced to do preview renders all the time.

Magic Lights For Matte Painting

When creating matte paintings, it is often convenient to block out the painting in 3D using simple shapes, then project textures through the camera on that geometry and then use basic lighting to alter the look of the basic shapes before moving onto the next stage which might be a paint application.

A spotlight is very much like a camera, so it is possible to magic spotlights to project layers of textures and shader properties. This is often quite handy with approaches were photogrammetry and 3D is combined with classic digital matte painting

Magic lights can also be used like those adjustment layers artists are already familiar with from Photoshop. The most common types of filters are probably hue/saturation and gain/gamma. In general, using lights to 'filter' properties of your shaders (or even other lights), is another use.

To have lights influence other lights, you need to run the illuminance() loop with their category first, record the information that comes from them, then illuminance() loops with other categories can offer this data to light asking for and reacting to it.

Using DarkTrees With Magic Lights

DarkTree[4] is an application to build procedural shade trees through an artist friendly interface. While is has it's problems in terms of filtering, it is a good alternative for smaller shops that either simply can't or want to afford shader writing-savvy TDs but do want to look into procedural textures more.

DarkTree is particularly well suited for being used with magic lights as it exposes complete access to its internal engine through a DSO. The engine then evaluates the shade tree, almost like a `texture()` call but with the benefit of having a full shader (including a BRDF) inside this 'texture'.

The approach we are going to take in this example is to have as few lines of code in our lights as possible. On first sight, it might look as if we can get away with merely using our magic lights as parameter containers. However, there are subtle problems when it comes to filtering and the SIMD shader execution model of most RenderMan renderers.

We only evaluate contribution in the light, all else happens in the surface shader. Think of contribution as a light intensity clamped between 0 and 1.

```
#include "brdf.h"
#include "displace.h"
#include "helpers.h"
#include "SimbiontRM.h"

surface magicDarkTreeSurface() {

    /* Setup basics */
    normal Nn = normalize(N);
    normal Nf = faceforward(N, I, N);
    /* The next two are needed or DarkTree */
    normal Nshader = ntransform("shader", Nf);
    vector Ishader = normalize(vtransform("shader", I));
    Ishader = normalize(vtransform("shader", I));
    vector V = -normalize(I);

    /* Define all our layers */
    color CdarkTreeColor = 0;
    color CdarkTreeSpecularColor = 0;
    float darkTreeDiffuseness = 0;
    float darkTreeSpecularity = 0;
    float darkTreeGlossiness = 0;
```

4 More information can be found at http://www.darksim.com/

```
    float darkTreeLuminosity = 0;
    float darkTreeDisplacement = 0;

    /* Since these light are 'texturing', we would
     * get wrong results if we specified a cone.
     */
    illuminance("magicdarktree", P) {

        /* Interrogate current magic light */
        point magicShadingPoint;
        lightsource("__magicShadingPoint", magicShadingPoint);

        float magicFilterWidthP;
        lightsource("__magicFilterWidthP", magicFilterWidthP);

        string magicDarkTree = "";
        lightsource("__magicDarkTree", magicDarkTree);

        /* This is the hand-shake call for the DarkTre DSO */
        float context = SimRM_GetContextID(magicDarkTree);

        /* The opacity comes from the */
        float darkTreeOpacity = 1;
        lightsource("__magicOpacity", darkTreeOpacity);
        if(0 < darkTreeOpacity) {
            /* We define a few inline helper functions to make the code more
compact */
            /* Looks up a float variable from DarkTree */
            float simbiontFloat(uniform float type) {
                extern float context, magicFilterWidthP;
                extern point magicShadingPoint;
                extern normal Nshader;
                extern vector Ishader;
                return SimRM_EvalFloat(context, type, magicShadingPoint, Nshader,
Ishader,
                                       magicFilterWidthP, 0, 0, 0, 0);
            }

            /* Multiply our opacity by the DarkTRees alpha, now we are good to go
*/
            darkTreeOpacity *= simbiontFloat(DTE_EVAL_ALPHA);

            /* Looks up a float variable from DarkTree and mixes
            *  it with the inout value based on darkTreeOpacity
            */
            float simbiontFloatMix(varying float input; uniform float type) {
                extern float darkTreeOpacity;
                return mix(input, simbiontFloat(type), darkTreeOpacity);
            }

            /* Looks up a color variable from DarkTree */
            color simbiontColor(uniform float type) {
                extern float context, magicFilterWidthP;
                extern point magicShadingPoint;
                extern normal Nshader;
                extern vector Ishader;
                return SimRM_EvalColor(context, type, magicShadingPoint, Nshader,
Ishader,
                                       magicFilterWidthP, 0, 0, 0, 0);
            }
```

```
            /* Looks up a colorr variable from DarkTree and mixes
            *  it with the inout value based on darkTreeOpacity
            */
            color simbiontColorMix(varying color input; uniform float type) {
                extern float darkTreeOpacity;
                return mix(input, simbiontColor(type), darkTreeOpacity);
            }

            /* Get most of the data from the DSO */
            CdarkTreeColor          = simbiontColorMix(CdarkTreeColor,
DTE_EVAL_COLOR);
            float darkTreeMetalness = simbiontFloat(DTE_EVAL_METAL_LEVEL);
            color CtmpSpecularColor = mix(simbiontColor(DTE_EVAL_SPECULAR_COLOR),
CdarkTreeColor,
                                          darkTreeMetalness);
            CdarkTreeSpecularColor  = mix(CdarkTreeSpecularColor,
CtmpSpecularColor,
                                           darkTreeOpacity);

            darkTreeDiffuseness     = simbiontFloatMix(darkTreeDiffuseness,
DTE_EVAL_DIFFUSE_LEVEL);
            darkTreeSpecularity     = simbiontFloatMix(darkTreeSpecularity,
DTE_EVAL_SPECULAR_LEVEL);
            darkTreeGlossiness      = simbiontFloatMix(darkTreeGlossiness,
DTE_EVAL_GLOSSINESS);
            darkTreeLuminosity      = simbiontFloatMix(darkTreeGlossiness,
DTE_EVAL_LUMINOSITY);

            /* Scale the displacement strength on a per-light basis */
            float tmpDisplacement;
            lightsource("__magicDisplacement", tmpDisplacement);
            tmpDisplacement         *= simbiontFloat(DTE_EVAL_ELEVATION);
            darkTreeDisplacement    = mix(darkTreeDisplacement, tmpDisplacement,
darkTreeOpacity);
        }

        /* End DarkTree DSO context */
        SimRM_ReleaseContextID(context);
    }
    /* Do displacement */
    Nf = Displace(Nn, "shader", darkTreeDisplacement, 1);

    color CplainColor = Cs * CdarkTreeColor;
    /* Diffuse & specular contributions */
    color Cdiffuse = darkTreeDiffuseness * pickyDiffuse(Nf);
    color Cspecular = darkTreeSpecularity * CdarkTreeSpecularColor *
                      pickySpecular(Nf, V, max(0, 1 - darkTreeGlossiness));
    Oi = Os;
    Ci = Os * (CplainColor * (darkTreeLuminosity + Cdiffuse) + Cspecular);
}
```

Magic lights are particularly well suited for painting wear & weathering effects on primitives. The idea of using lights to model natural forces that cause these kind of surface imperfections was first described in [5]. I suggest this paper as further reading. I also believe that it is the first paper to mention what later became ‘ambient occlusion’.

If we look at lights as photon emitters, we realize quickly there there are a few similarities between particles carried by natural forces, like sand scratching on a metal surface, dirt settling in ridges (think about ambient occlusion) and so forth.

We categorize lights: a wind force (carrying dust which might settle on objects and possibly scratch surfaces) would be modeled through a directional light. A point light would act as a local source, like a settlement of dirt were moss starts growing or were the surface is damp and rust starts building up etc.

Below are an ambient light that affects the whole scene and a point light that has a `smoothstep()` falloff as this is better suited for textures than the power-based falloffs commonly found in lights.

```
#include "filterwidth.h"

light magicDarkTreeAmbient(
           uniform string DarkTree            = "";
           uniform float  Scale               = 1;
           uniform float  Displacement        = 1;
    output uniform string __category          = "magicdarktree";
    output varying float  __magicOpacity      = 1;
    output varying point  __magicShadingPoint = 0;
    output varying float  __magicDisplacement = 0;
    output varying float  __magicFilterWidthP = 0;
    output uniform string __magicDarkTree    = "";
) {
#ifdef DELIGHT
    normal Nn = normalize(Ns);
#else
    normal Nn = normalize(N);
#endif
    /* Assign output vars for message passing */
    __magicDarkTree = DarkTree;
    __magicDisplacement = Displacement;

    /* Get Ps in shader space */
    point Pshader = transform("shader", Ps * Scale);
    __magicShadingPoint = Pshader;
    /* Calculate filter size for DarkTree */
    __magicFilterWidthP = filterwidthp(Pshader);
    /* Calculate magic light opacity */
    __magicOpacity = 1;
    /* We need an illuminate construct or else our light will not get called! */
    illuminate(Ps + Nn) {
        Cl = 0;
    }
}
```

The first thing you might notice is that this ambient light still has an `illuminate()` block. The reason is simply that, as mentioned in the beginning, ambient lights don't get run by the `illuminance()` construct. The illuminate(Ps + Nn) is a hack but it does exactly what we want.

Recall that 3Delight uses `Ns` to distinguish between the surface normal of the point being shaded and the surface normal of a potential area light geometry the light is run on which is why we the pre-processor clause in there.

We use the expose the light's transformation to the surface shader through `__magicShadingPoint`. Since we are dealing with solid textures, we can use the light's transformation to align and scale the texture. We have to calculate the filter width of that point in the the light, as an `illuminance()` loop counts as a conditional block. If that comes a s a surprise, think about it being encapsulated in a huge `if(__category==...){ ... }`. As the `filterwithp()` macro from [2] uses an an area operator (namely `area()`), we can't call it in such a loop without risking that we get a bogus filter size.

The point light uses an Fbm from *noises.h* [2] to break up the edge of the light contribution area a little bit.

```
#include "filterwidth.h"
#include "noises.h"

light magicDarkTreePoint(
           uniform string DarkTree            = "";
           uniform float  Scale               = 1;
           uniform float  Radius              = 1;
           uniform float  Falloff             = 1;
           uniform float  Displacement        = 1;
    output uniform string __category          = "magicdarktree";
    output varying float  __magicOpacity      = 0;
    output varying point  __magicShadingPoint = 0;
    output varying float  __magicDisplacement = 0;
    output varying float  __magicFilterWidthP = 0;
    output uniform string __magicDarkTree     = "";
) {
    /* Assign output vars for message passing */
    __magicDarkTree = DarkTree;
    __magicDisplacement = Displacement;

    /* Get Ps in shader space */
    point Pshader = transform("shader", Ps * Scale);
    __magicShadingPoint = Pshader;
    /* Calculate filter size for DarkTree */
    __magicFilterWidthP = filterwidthp(Pshader);
    /* Calculate magic light opacity */
    illuminate(point "shader"(0, 0, 0)) {
        /* Calculate distance-based smoothstep falloff */
        __magicOpacity = smoothstep(Radius, Radius + Falloff, length(L));
        /* Roughen the shape up a little with noise */
        __magicOpacity = mix(1, smoothstep(0.25, 0.7, fBm_default(Pshader)),
__magicOpacity) *
                             (1 - __magicOpacity);
        Cl = 0;
    }
}
```

DarkTree comes with a huge library of free example 'dark trees'. Below is an example of the kind of result one can get in a few minutes by using some of these presets. The figure on the left shows the light source positions

and core sizes as semi-transparent spheres. I only used four patterns, all are full solid procedural textures. The one omitted in the image below is a kind of lichen that I is on the big magic light at the base of the hydrant.

There is one ambient magic light in there that casts the galvanized steel surface as a base on the whole object.

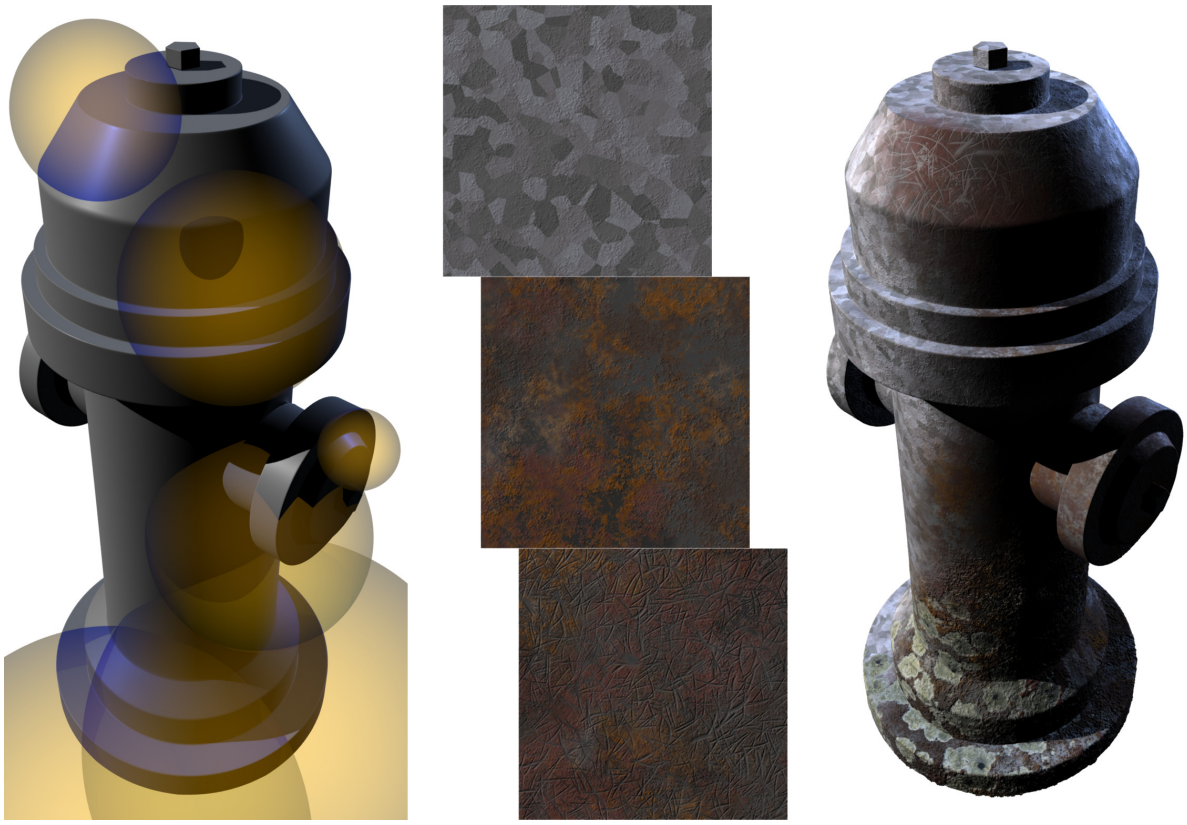


Image 19: Simple object textured & shaded quickly and painlessly using five point and one ambient magic light with dark trees.(a) object with magic lights rendered as spheres (b) subset of procedural shaders used © final result

Another advance of magic lights over traditional textures is that they are defined spatially in 3D. In the above example it means that unless light linking prevented it, the lichen would also show on the surface the hydrant sits on, e. g. A sidewalk. Using textures, one would likely need to touch at least two textures and have quite some trouble lining the edges up, unless all the is done in a full 3D paint application.

For any look-related set dressing task, magic lights are a gift of the gods. They also integrate nicely with an existing, texture based pipeline, as one can always bake everything into good old 2D texture maps, provided the geometry has texture coordinates in the first place. With large scenes, one find themselves quickly using dozens or even hundreds of magic lights. It is obvious that we don't want to evaluate them over and over again, particularly not if the geometry the texture is static. The next section has some suggestions what we can do

about that.

Baking

Lights are expensive. Particularly if we use many of them as suggested in the last section. The first thing to do to get our render times down is to link lights to those primitives they do their magic on. If you have 50 magic lights in your scene, chances are only a few affect all primitives. By linking lights to primitives through the use of the `Illuminate` RIB command, we can stop the evaluation of lights on surfaces they don't affect anyway.

The next thing to look at is the use of baking. Traditionally, baking was done using traditional 2D texture maps [6].

However, one of the big advances of using lights over traditional texture mapping is that we don't require texture coordinates. If we baked into 2D textures, we would need to create texture coordinates for each and every primitive — a very time consuming task that also would quite likely invalidate the resolution independence of the whole approach too.

A new option are brick maps. Brick maps are sparse 3D texture maps. Like traditional 2D textures, they are 'mip- mapped', so they filter beautifully when looked up at different filter sizes. They don't require any texture coordinates to be present, so they are ideally suited for our purposes.

On a side note, if you use DarkTree, brick maps also solve the problem of the non existing anti-aliasing of those procedural textures. Once everything is baked into brick maps (which you can super-sample, if you need, by lowering shading rate), the brick-map filtering will take care of the anti-aliasing automagically.

A Production's Perspective of Magic Lights

Usually, when an asset, like a character, prop or set is locked-off to get used in shots, so are it's textures and shaders. Magic lights provide a convenient way to change the look of an asset without interfering with this idea.

Imagine you work on a shot and you need to do a local modification to a character asset's look. That might for example be altering the specular value a little bit on one cheek to remove or push back a highlight the director doesn't like showing up in the particular shot. Traditionally, you needed to modify the texture map that steers this shader parameter. If it exists at all. If not (say the specular intensity was steered through a primitive variable or procedural texture), you might even need to add a slot for it into the shader, thus altering two parts of a locked-off asset. To do the latter, you create a local (as in local to the shot) copy of your asset and do the modifications, then make sure that that shot uses this local asset version, instead of the global one at render time.

If you are lucky, the asset management system you use supports this work flow. If not, in the worst case, you might find yourself in world of pain trying to do something your pipeline wasn't built to do. What's more, in the case of a shader modification, TD skills are required, so an 'ordinary' artist wouldn't be able to see the shot through anymore.

An often overlooked alternative that is kind of in the middle between traditional 2D texture maps with the I just touched and the ease of use of magic lights, are primitive variables. They have been available since ever, yet people seem to make very sparse use of them, probably because most 3D packages hide the functionality to deal with those per-vertex attribute maps quit well. They come with a smaller subset of the problems that I described above. On the pipeline side, they are as easy or hard to integrate as are texture coordinates. On the creation side though, thy can be quickly painted on, so the requirement to go to 3D paint ord digital imaging application is removed. And while no sophisticated unwrapping of a surface is needed, it still requires that some way exists, to attach those primitive variables to a locked-off asset afterwards.

Magic lights provide a convenient way around these potential problems. If all shaders are written so that they allow modifying most aspects of their look through such lights, most of the time artists working on shots will be able to do get look modifications done themselves. Neither requiring modifications to a branched copy of the asset, nor a TD to carry them out.

In our example, you would use a magic light to locally tweak the specular intensity. If the respective magic light's shape parameters wouldn't suffice, you could paint a map and project that through the light to get more refined local control. Using magic displacement lights, you can even subtly (or radically) alter the shape of a primitive, e. g. to improve a particular silhouette an asset shows from an angle the modeling artist didn't expect it to be looked at in a shot.

The possibilities are limitless and they allow particularly smaller shops, that don't have a sophisticated asset management system that does branches of versions of assets in a way that supports the 'old school' approach described above, to give artists working on shot a very refined level of control over the look at the latest stage in the pipeline possible.

Conclusion

I have given an overview of some of the many useful applications that magic lights have in VFX production. I didn't touch animation, but it should be obvious that they are also very well suited for complex effects shots where geometry has to deform in ways that are hard to tackle with traditional approaches or where surfaces have

to have their look react with their environment in way s that make it hard to use common ways like e. g. animated texture maps.

Magic light are on of the least invasive way to tweak the look of something at any stage in the pipeline, as long as shaders can be modified. Apart from this traditional application, we also touched the iceberg's tip of the fun stuff that one can do with them, with a little bit of imagination. Last but not least we looked at using them for look development.

If you are starting a new project, it is always wise to keep a back door open in your shaders so they can be tweaked through magic lights. You never know what the exact challenges of a show ahead of you might be and were they might come handy. If you implement the light side as well from the beginning, you will give the non-TDs at your place a powerful tool to help them push shots over the line on their own. Last but not least they open the door for some work flows that less TD-heavy places can look into to get along better with the few TDs they have.

I look forward to hear suggestions for much crazier use of magic lights than this paper touched.

References

[1] Steve Upstill. *The RenderMan Companion*. Reading, MA: Addison-Wesley, 1990.
[2] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan*. San Francisco, CA: Morgan Kaufmann Publishers, 2000.
[3] Ebert, Musgrave, Peachey, Perlin, Worley, Texturing and Modeling, A Procedural Approach Second Edition, Academic Press Professional, 1998.
[4] The RenderMan Interface Specification, versions 3.2 and 3.3 (draft)
https://renderman.pixar.com/products/rispec/index.htm
[5] Tien-Tsin Wong , Wai-Yin Ng and Pheng-Ann Heng. A Geometry Dependent Texture Generation Framework for Simulating Surface Imperfections. Proceedings of the 8th Eurographics Workshop on Rendering, St. Etienne, France, June 1997, pages 139—150.
[6] Larry Gritz, A Recipe for Texture Baking, RenderMan in Production, SIGGRAPH 2002, Course 16, pages 45—54

RenderMan In Production at WDFA

Heather Pritchett & Tal Lancaster - Walt Disney Feature Animation

Look Development at WDFA

Heather Pritchett

Production Pipeline

In the simplest of production pipeline flow charts, Look Development sits squarely between the Modeling department and the Shot Finaling department. The former feeds us models of individual elements, which we create looks for. The term “look” is our word for the final appearance created by the materials, textures and lighting properties. Shot Finaling takes these looks, along with the final animation, and lights, renders and composites them for film.

Image 20: Simple Flowchart

In reality, it is not that simple. Not only do we need the final model, we need reference for the visual development department. We also need some idea of how each element is actual used in the film. Do we see it

closeup? Far away? Is a character interacting with it? Is the effects department involved? How do we plan on lighting the sequence this element is in? Some of these questions are very difficult to answer and many of the answers change over time. Our looks can also be pulled in by multiple downstream departments, including effects and layout.

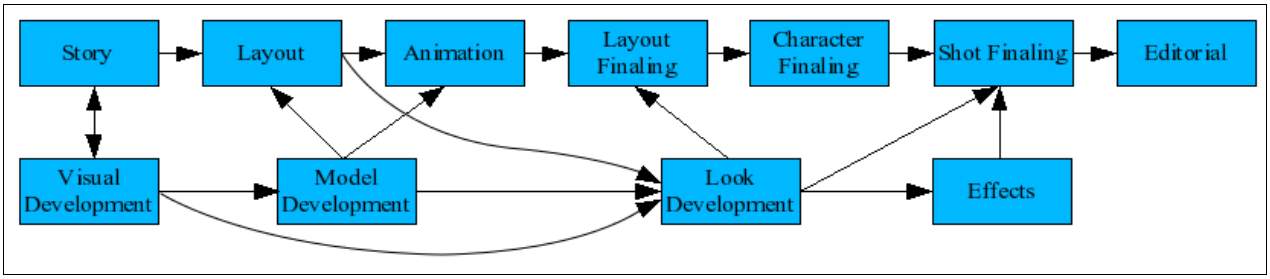


Image 21: More realistic flow chart.

The layout department, which provides us with definite answers as far as cameras and placement, is usually working concurrent to us. At the beginning of production, we have a heard start on them and are often making best guesses on final element usage based on story sketches. Toward the end of production, we can get very concrete information, including final cameras and sequence lighting. Unfortunately, the downside to having this information is that the sequences containing these elements are potentially waiting on your look.

Shaders

Our shader development approach is a modular one. Instead of having separate shaders per material or per element, we have shader modules. Look at a module as a single library of RSL code designed to handle one aspect of a complex shader.

The majority of our modules are lighting-based, for example, “Lambert Diffuse” , “Ambient”, “Blinn Specular” or “Greg Ward Larson Anisotropic Specular”. We also have modules for reflection and transparency, ray traced or map-based, as well as modules for alternate texture mapping methods, just to name a few.

The final .slo files are created using a complex Makefile system that assembles each shader out of a collection of modules based on the desired functionality. This system tracks all dependencies, and, in addition to the .slo files, creates custom .slim files for each of our shaders using an interface design code that is embedded in the comments following each parameter declaration.

For the most part, our final shaders are designed to fit a general need as opposed to a specific prop or character. For example, our workhorse shader is “s_basic”. This is used on well over half of the elements in “Meet the

Robinsons". It includes these modules:

- Color Layers
- Shading Expressions
- Basic Transparency
- Lambert Diffuse
- Blinn Specular
- Ambient
- Basic Reflection

We use another shader, "s_basicAniso" that is almost identical except it has the Ward Anisotropic module instead of the Blinn Specular. "s_basicPhong" similarly has the Phong Specular module.

We also have a concept of shader layers. While it's primarily driven by the Color Layer module, this allows a users to composite, in the shader, multiple versions of that module. The default number of layers is 2, but we have shaders with as many at 16 which are used for some of our more complex skin rendering. "s_basic" is often called the "s_basic" family because of all these cousins:

- s_basic1
- s_basic4
- s_basic8
- s_basic12
- s_basic16

A compact example is "s_constant". This shader is one of our smallest given the lacking of any lighting modules.

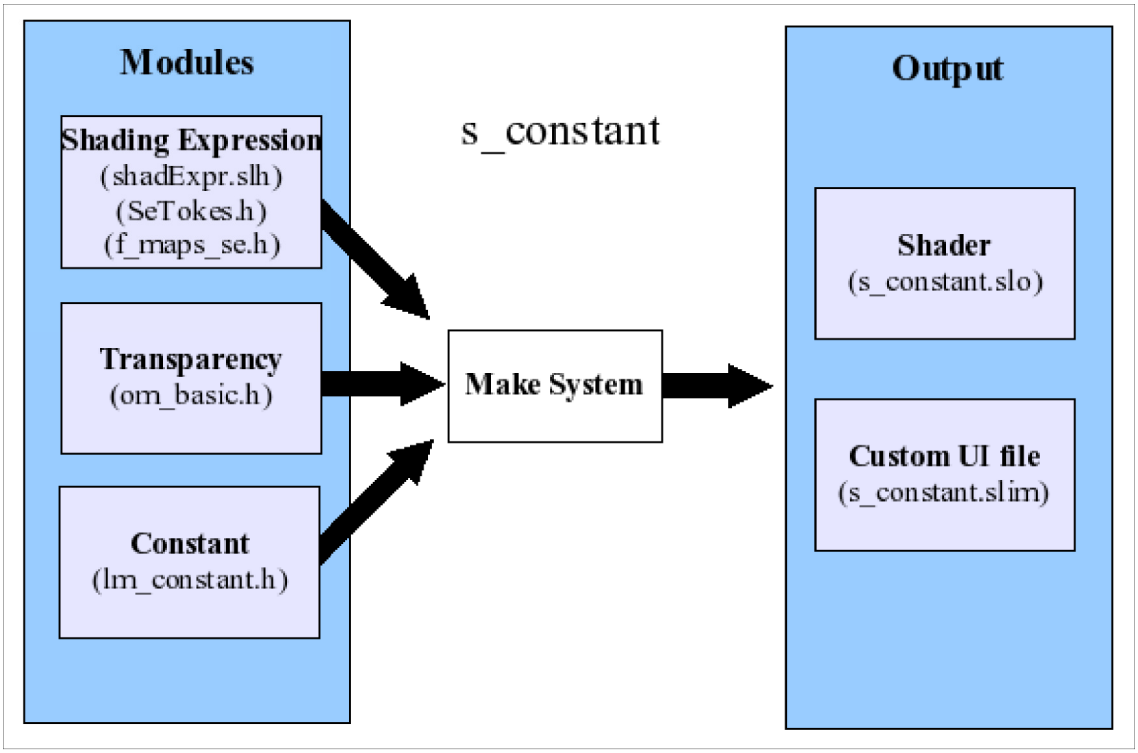


Image 22: s_constant file breakdown

This system has several immediate advantages. There is a common interface presented to all users. A basic reflection module will look the same no matter what shader they are using. In addition, production wide changes can be implemented quickly and globally with minimal effort. On "Meet the Robinsons" we have about 190 shaders. But essentially there are only 10 .sl files that we would edit. The other .slo files are variants of the different combinations of libraries that users have needed.

Look Development Tools and Methods

Tiling

The majority of our models are composed of subdivision surfaces. We use a method called tiling to define paintable regions on these surfaces. Each of these regions are a direct mappings to a separate parametric texture space.

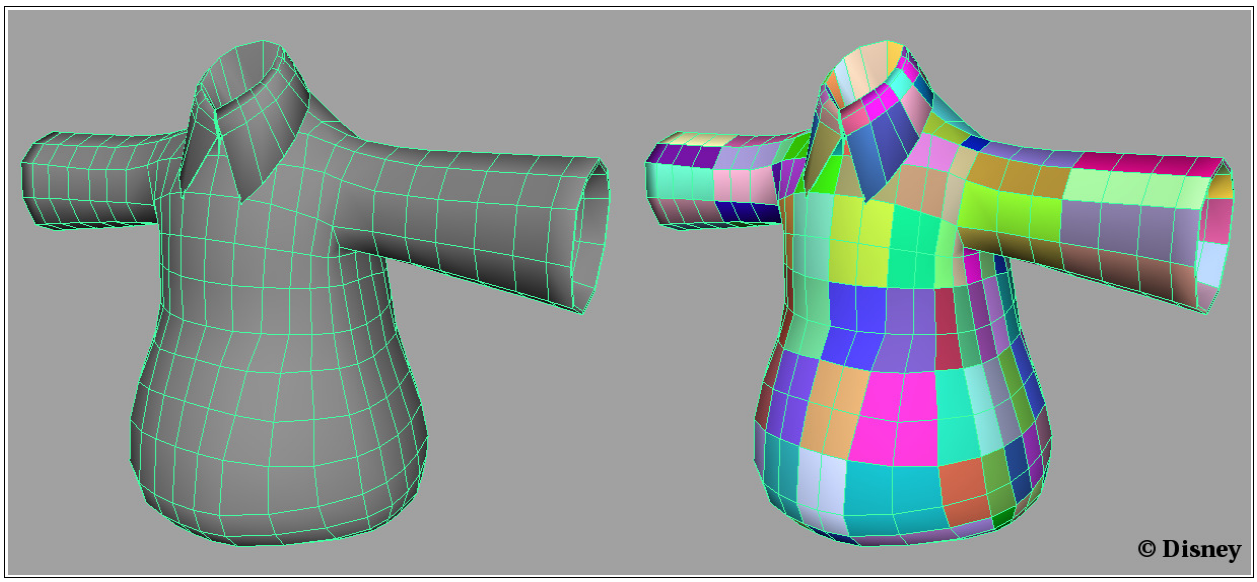


Image 23: The model with tile definitions.

Paint3d

Paint3d is our proprietary painting system. It allows users to directly paint on a 3d model or send snapshots of a particular view out to Photoshop for detailed paint work which can then be projected onto the 3d model. It can generate both point clouds and brick maps and it shares expression libraries with our shaders. Those same shader expressions can be tweaked and baked in paint3d if desired.

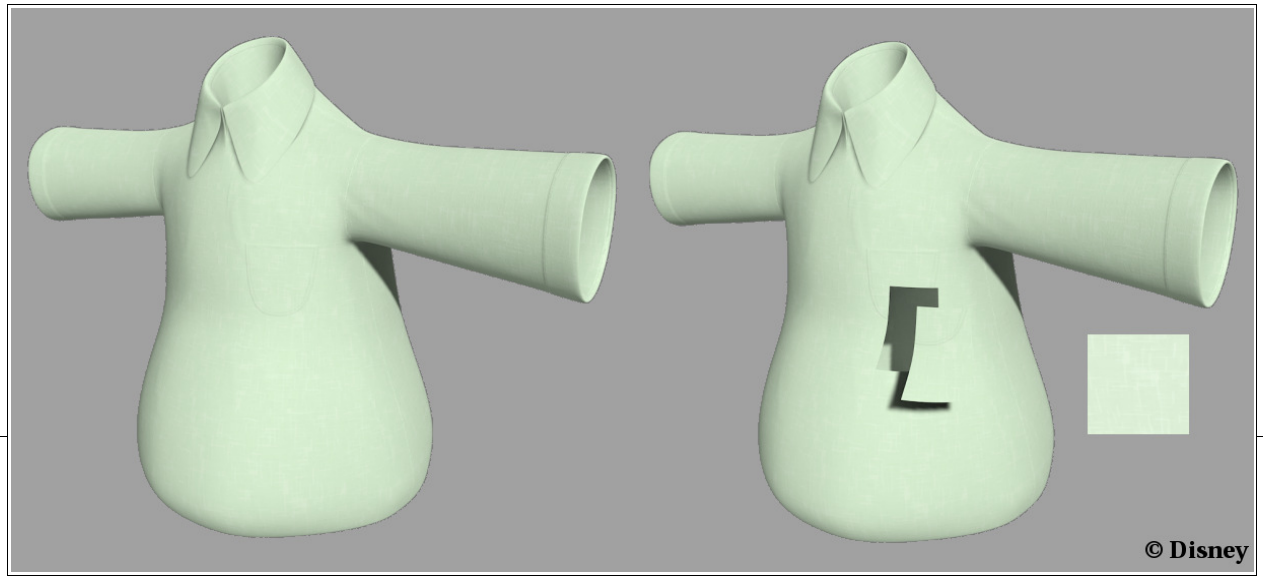


Image 24: Rendered image showing tile.

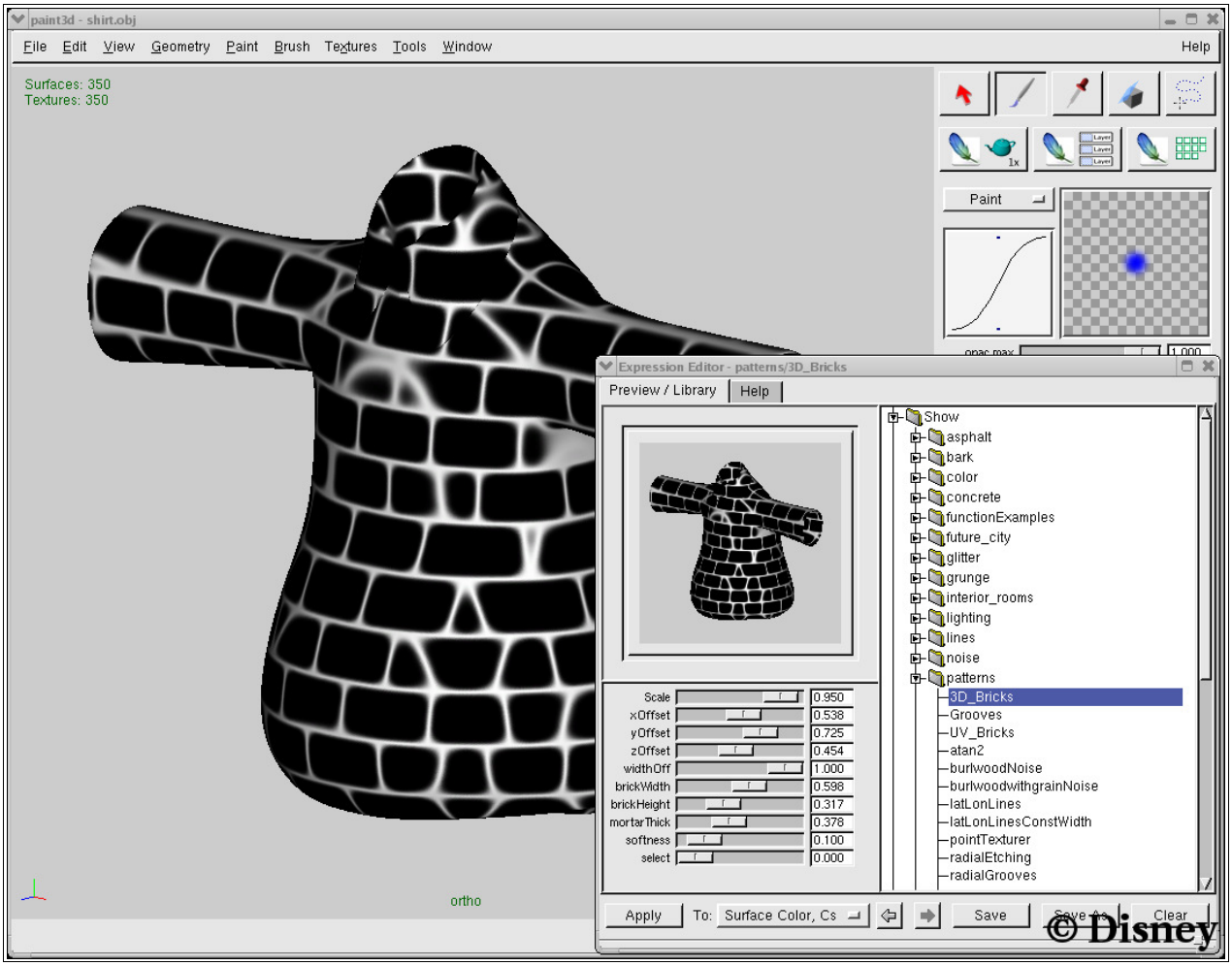


Image 25: Paint3d and a brick expression.

1

XGen

XGen is our arbitrary geometry generation tool. It gives us hair, feathers, scales, grass, leaves, litter and even entire trees, to name a few. Developed back in 2002, the basic premise behind XGen is to provide a system that allows for the population of primitives onto a model in much the same way that a hair system allows for hair. It is integrated into Maya and renders through MTOR with the assistance of a RIB generator. Much like Paint3d, XGen is also integrated into our workflow. It uses the same shader expression library that the shaders and paint3d use. There are also shader modules that allow access to XGen data, like object id, primitive position and even the point on the under lying surface that is generating the primitive. This lets us render hair with the hair color pulled from a matte painted on the head.

Image 26: Character with XGen hair, brows and mustache

Image 27: Set with XGen foliage

ppp.pl: A Perl Preprocessor for All Your Shader Build Needs

Heather Pritchett

The Challenge

Our modular shader system is managed through a series of Makefiles and the careful embedding of macros in our shader code. While very powerful, this system was a bit daunting to manipulate. One of the more frustrating issues with C pre-processor (cpp) macros was the difficulty working with multiple lines of code with the line continuation character '\'.

For example, in one of our simplest modules, constant, the reference in the main .sl file (s_basic.sl) would look something like this:

```
...
#include <lm_constant.h>
...

surface s_basic (
    ...
    LM_CONSTANT_INPUT_PARAMS
    ...
    )
{
    // Global Init section
    ...
    LM_CONSTANT_VARS
    ...
    // Function definitions
    ...
    LM_CONSTANT_FUNC_DEF
    ...
    // Perform lighting model calculations
    ...
    LM_CONSTANT_BODY
    ...
    // Apply lighting model calculations
    ...
    LM_CONSTANT_LIGHT;
    ...
}
```

This is highly simplified, but shows that the core .sl file breaks up all references to each modules as a series of macros that provide wrapper-like access to the module.

The include file for the constant module (lm_constant.h) would them contain a section of code that looked like this:

```
#define LM_CONSTANT_INPUT_PARAMS \
    float constant_bypass = 0; /* cat Constant type switch \
       desc {Turn this on to skip constant light.} */ \
\
    float Kconstant = 1.0;  /* cat Constant \
       desc {Scalar for constant contribution.}*/ \
\
    string constantMask = ""; \
    /* cat Constant type texture \
       desc { An attenuation mask for constant light. } */
/* LM_CONSTANT_INPUT PARAMS */

#define LM_CONSTANT_VARS color Ctconstant = 0;

#define LM_CONSTANT_BODY \
    color Ctconstant = 0; \
    if (constant_bypass != 1) \
        Ctconstant = fa_constant_calculate(oPref, Kconstant, \
                     ss, tt, constantMask);
/* LM_CONSTANT_BODY */

#define LM_CONSTANT_LIGHT \
    Ci += Ct*Ctconstant;
/* LM_CONSTANT_LIGHT */
```

These four sections, input parameters, global variables, body and light, were each macros that were called from the s_basic sl file. The rest of the shader contained the actual definition of the fa_constant_calculate function. Obviously, working with backslashes can be very difficult. With the addition of our preprocessor, the code can now look like this:

```
 %def LM_CONSTANT_INPUT_PARAMS {{
    float constant_bypass = 0; /* cat Constant type switch
       desc {Turn this on to skip constant light.} */

    float Kconstant = 1.0;  /* cat Constant
       desc {Scalar for constant contribution.}*/

    string constantMask = "";
    /* cat Constant type texture
```

```
       desc { An attenuation mask for constant light. } */
}} /* LM_CONSTANT_INPUT_PARAMS */

%def LM_CONSTANT_VARS color Ctconstant = 0;

%def LM_CONSTANT_BODY {{
    color Ctconstant = 0;
    if (constant_bypass != 1)
        Ctconstant = fa_constant_calculate(oPref, Kconstant,
                     ss, tt, constantMask);
}} /* LM_CONSTANT_BODY */

%def LM_CONSTANT_LIGHT Ci += Ct*Ctconstant;
```

It probably seems like a minor difference, but when you start working with longer shaders and involving #ifdef statements, then the complexity can easily become frustrating.

Implementation Details

The initial functionality was very simple. The "%def" marker was the equivalent of a #define. With the added {{ and }} markers, everything between was assumed to be part of the macro definition. The script handled all substitution itself and even tracked the original file numbers for error messages.

```
%def  X
%def  X {{
...
}}
```

Adding Parameters

This initial functionality was very useful and we began to expand upon it. We wanted to be able to pass parameters through to macros:

```
%def  X(A, B, C)
%def  X(A, B, C) {{
...
}}
```

In a very hypothetical example, lets say I wanted to create some different macros for my constant parameters in which the bypass and Kconstant settings had different default values. I could change my definition like this:

```
%def LM_CONSTANT_INPUT_PARAMS (defaultBypass, defaultK) {{
    float constant_bypass = defaultBypass; /* cat Constant type switch
       desc {Turn this on to skip constant light.} */

    float Kconstant = defaultK;  /* cat Constant
       desc {Scalar for constant contribution.}*/

    string constantMask = "";
    /* cat Constant type texture
       desc { An attenuation mask for constant light. } */
}} /* LM_CONSTANT_INPUT_PARAMS */
```

Now, assuming I'm making different macro references in different files, I can easily set different default values:

```
LM_CONSTANT_INPUT_PARAMS(0, 1)
```

or

```
LM_CONSTANT_INPUT_PARAMS(1, 0.5)
```

Adding Comments

We also decided to change how we dealt with comments. We initially decided to not expand comments dealing with a macro. However, as we starting looking toward new functionality, we felt we needed this option. We debated simply folding the new functionality back into the original %def, but decided to introduce a new convention to avoid breaking older code and because we felt default comment substitution wasn't very safe.

```
%commentDef  X(A,B,C)
%commentDef  X(A,B,C) {{
...
}}
```

This was primarily intended to support changes to our slim parameter definitions, which are embedded in comments. As an example, the LM_COMSTANT_INPUT_PARAMS (defined above) would produce this section of code in a .slim file:

```
collection Constant {Constant} {
parameter float constant_bypass {
        subtype switch
        description {Turn this on to skip constant light.}
        default {0}
}
parameter float Kconstant {
        description {Scalar for constant contribution.}
        default {1.0}
}
parameter string constantMask {
        subtype texture
        description { An attenuation mask for constant light. }
        default {}
}
}
```

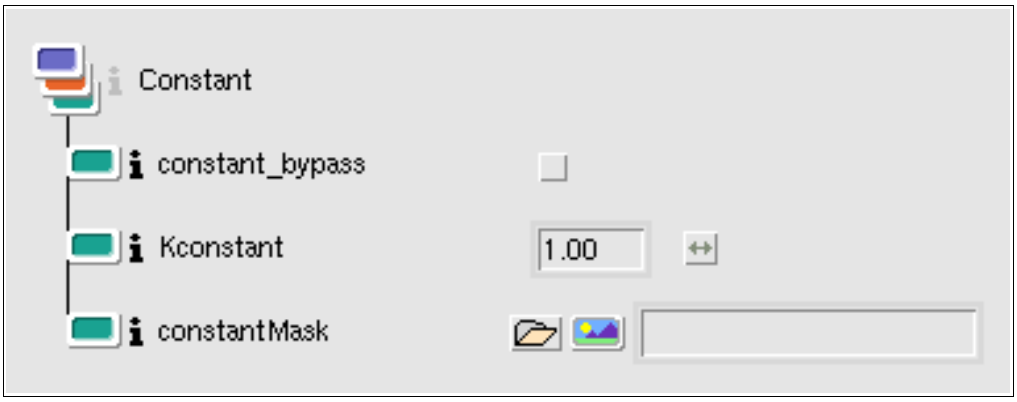


Image 28: Automatically generated .slim files

Lets assume (again, vastly hypothetical) that I want to change the name of the slim category based on whether or not the bypass is on or off.

```
%def LM_CONSTANT_INPUT_PARAMS (catLabel,defaultBypass,defaultK) {{
    float constant_bypass = defaultBypass; /* cat catLabel type switch
       desc {Turn this on to skip constant light.} */
```

```
    float Kconstant = defaultK;  /* cat catLabel
       desc {Scalar for constant contribution.}*/

    string constantMask = "";
    /* cat catLabel type texture
       desc { An attenuation mask for constant light. } */
}} /* LM_CONSTANT_INPUT_PARAMS */
```

Which can be referenced as:

```
LM_CONSTANT_INPUT_PARAMS(ConstantOn,1,1)
```

or

```
LM_CONSTANT_INPUT_PARAMS(ConstantOff,0,1)
```

Special Operators

Things get a little more complicated if I decide I want to change the actual parameter name. Now we have white space issues. To resolve this, we introduced some operators that told the preprocessor when it needed to concatenate objects.

%cat%

%Ucat%

These operators simply append the right symbol to the left symbol, similar to ##. It allows us to greatly simplify macro calls and assists in maintenance. So now we can do something like:

```
%def LM_CONSTANT_INPUT_PARAMS (catLabel,varLabel,defaultBypass,defaultK)
{{
    float varLabel%cat%bypass = defaultBypass; /* cat catLabel type switch
       desc {Turn this on to skip constant light.} */

    float K%cat%varLabel = defaultK;  /* cat catLabel
       desc {Scalar for constant contribution.}*/
```

```
    string varLabel%cat%Mask = "";
    /* cat catLabel type texture
       desc { An attenuation mask for constant light. } */
}} /* LM_CONSTANT_INPUT_PARAMS */
```

Which can be referenced as:

```
LM_CONSTANT_INPUT_PARAMS(ConstantOn,constantOn,1,1)
```

or

```
LM_CONSTANT_INPUT_PARAMS(ConstantOff,constantOn,0,1)
```

The %Ucat% operator is identical to %cat%, except it will capitalize the right symbol.

symbolA%cat%symbolB => symbolAsymbolB

symbolA%Ucat%symbolB => symbolASymbolB

This allows for finer control over parameter names since some people are particular about capitalization.

Tease

The examples presented using %commentDef and the parameter expansion are overly simplified and provide minimal functionality. These developments were generated for use on shows after Meet the Robinsons for a system of embedded alternate shading models. This system is still in development and wasn't appropriate for demonstration this year, but we hope to have the opportunity to present in at SIGGRAPH 2007.

Credits

Scott Mankey, initial implementation

Patrick Dalton, further development

Further Information

A detailed description of the slim file format can be found included in the RAT documentation under:

~yourRatDocsHomes/programmingRAT/customizing_slim/slimfile.html

An example script and description for the encoding can be found at:

http://www.renderman.org/RMR/Utils/sl2slim/index.html

Normal Mapping: All the tree, none of the weight.

Heather Pritchett

The Challenge

Several sets in our movie contained large and expansive outdoor vistas populated by numerous shrubberies and topiaries. We had an established look for topiaries that used XGen, an arbitrary geometry package we already used for hair. We used it to grow leaves on the bush shapes. Unfortunately, this method, which producing excellent foliage, was a little heavy and didn't scale down very well for distant topiaries. We need some method to produce a low resolution look that gave the feel of individual leaves, but without all the weight.

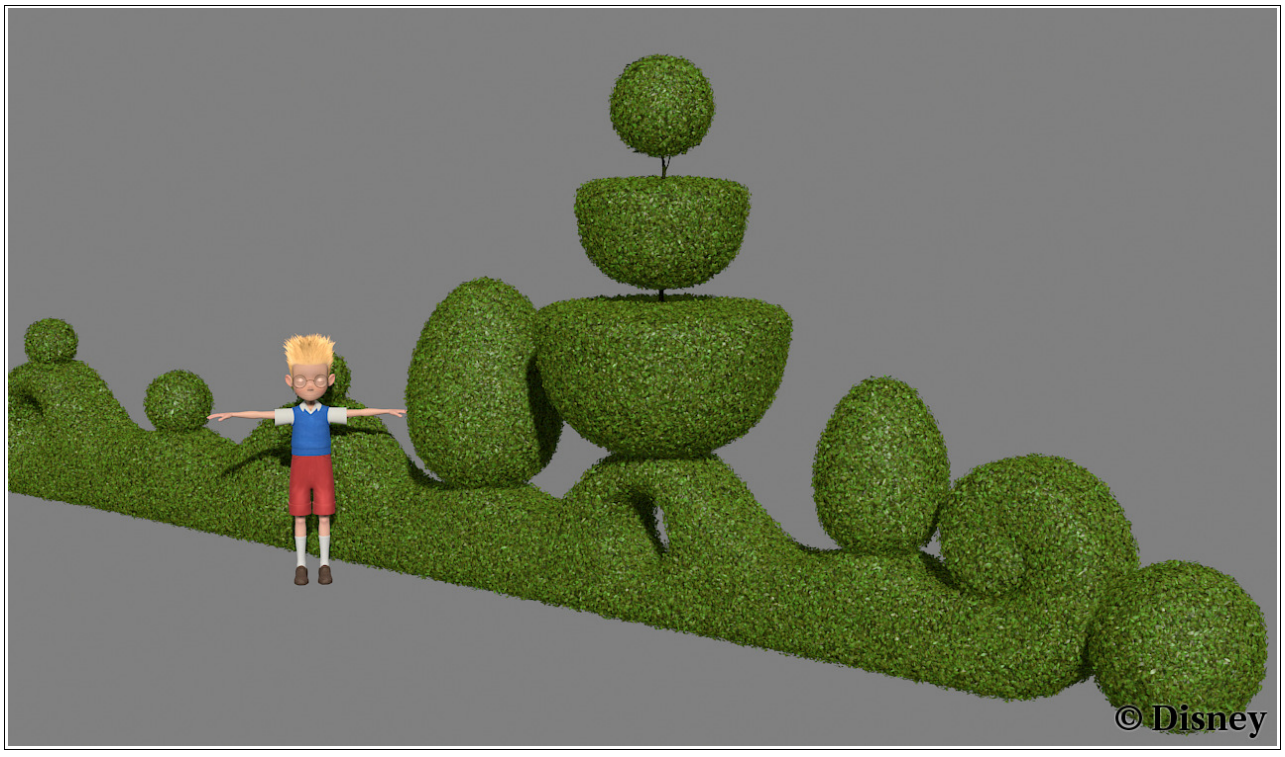


Image 29: One of our more complex topiaries

Solution: Capture the final "XGened" look and paste it on using shaders. In the final render, cross fade between the high resolution individual leaves and the low resolution shaders.

Step 1. Capture the XGen

After some trial and error, we determined we could reproduce the final look with three captured layers:

- A color layer, presenting the final color of the leaves
- A displacement layer, showing the final displacement of the leaves
- A normal map containing the actual surface normals of each leaf.

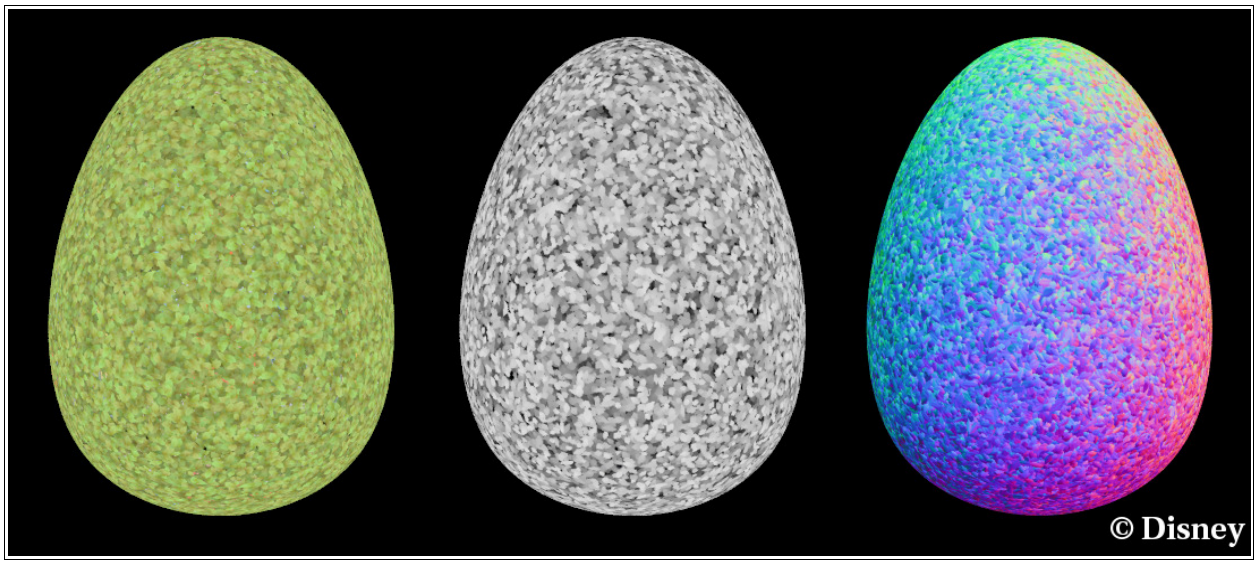


Image 30: Three captured layers.

To capture the data, we wrote a fairly simple shader that shoots a ray along the normal in toward the surface. The ray grabs the value we're interested in (color, displacement or normal) and saves it out in a point cloud data format. See Appendix I for the full code.

Several changes had to be made to RenderMan's default rendering environment to support this method of capturing the leaf values onto the underlying surface since, by default, RenderMan prefers to render only what the camera sees. We used ribboxes to set these attributes:

```
attribute "cull" "backface" [0]
attribute "cull" "hidden" [0]
attribute "dice" "rasterorient" [0]
```

In addition we turned off trace visibility for the underlying surface as well as camera visibility for the leaves.

```
attribute "visibility" "int trace" \[0\]  # apply to source surface
attribute "visibility" "int camera" \[0\] # apply to leaves
```

The shader itself has a bias involved that defines how far out the ray starts. This can be critical for concave surfaces. If the bias is too high, it will cause artifacts in the maps. If it's too low, it misses the leaves, which are positioned just above the surface. The bias was especially critical for the displacement layer, since it was used to normalize the values. If the value was too high, you would lose resolution. Concave surface in general were something of a challenge. The worst case image below was nearly impossible to create a clean capture for. However, in our final use of these images, the artifacts created were acceptable.

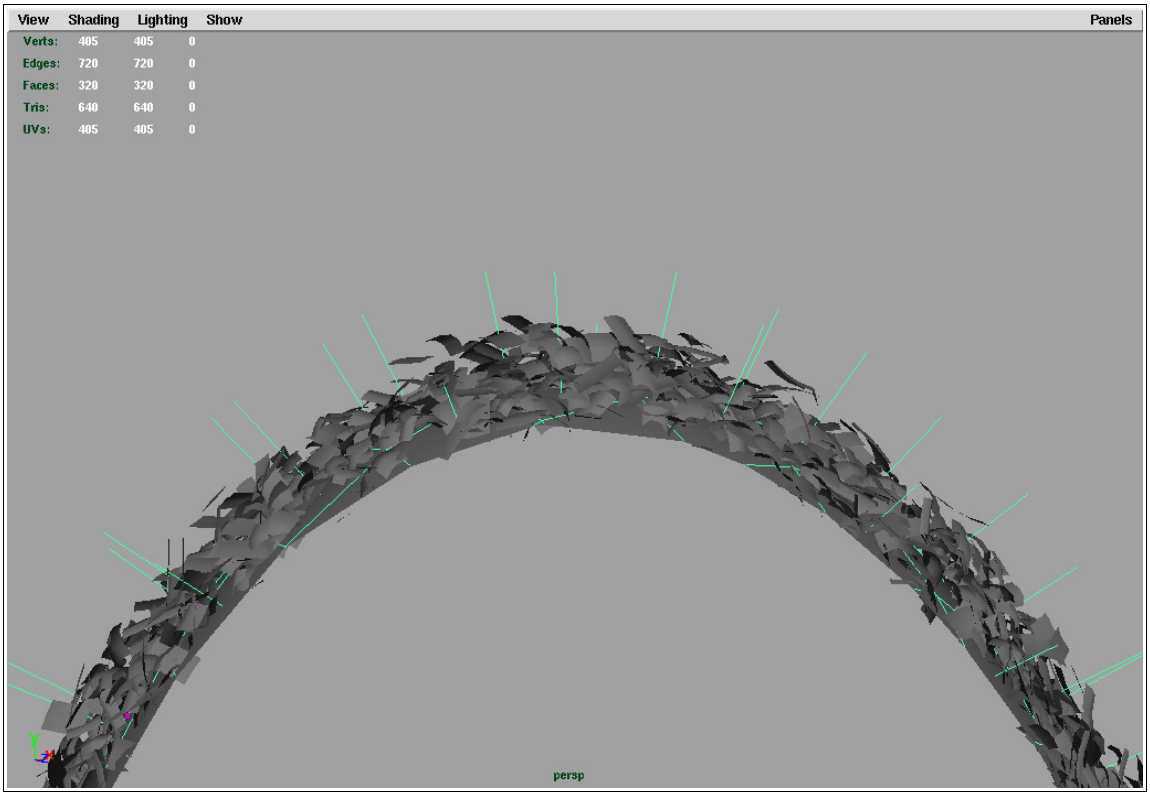


Image 31: Easy bias example.

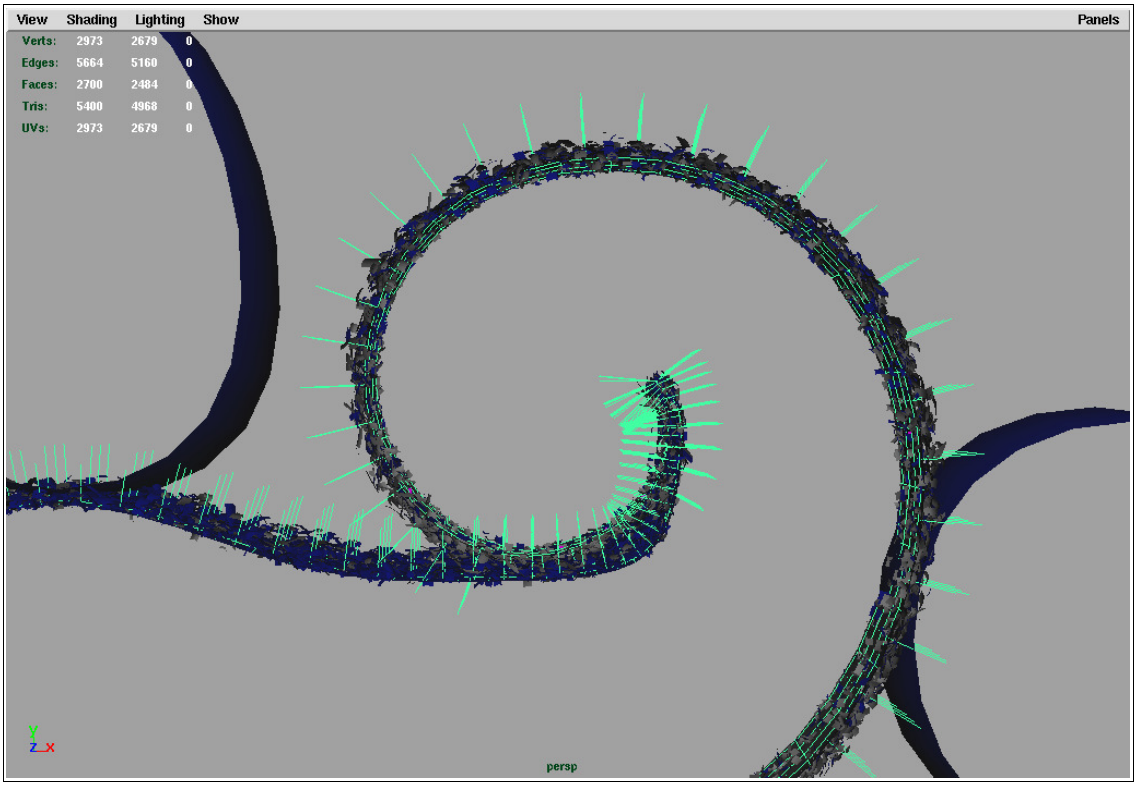


Image 32: A difficult bias example

The process itself was relatively straightforward. We would break each of the topiaries out into component shapes. In each component, we would apply the final XGen leaf look to the surface and attach the s_bakeproperties shader to the underlying surface. We would generate the three point cloud files for each layer.

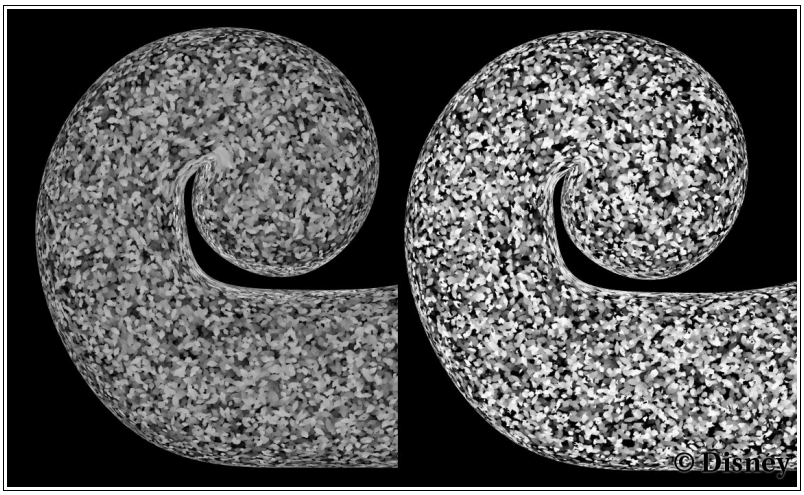


Image 33: Bias too high and too low.

Step 2. Parametric Maps

Because we like dealing with parametric maps in our shaders, we decided to bake the point cloud data back into parametric maps that could be used by our standard shaders. This was done with our proprietary paint3d package. It was able to load each map and apply it to the original component surface that generated it. The process proved to be a little more challenging than we had anticipated as we discovered that several of the large components (like the bottom part of this topiary) generated massive point clouds that overwhelmed paint3d. The solution was to break up some of the larger shapes and render their layers passes in sections.

Step 3. Shader Implementation

With our three maps in hand, the final shader implementation is relatively straight forward. The surface shader simply references the new color map. The specular is masked out by two combined masks. The first is generated from the displacement map and blocks out areas with no leaves (areas where the rays didn't hit anything). The second mask is a simple voronoi noise pattern that we're using to break things up.

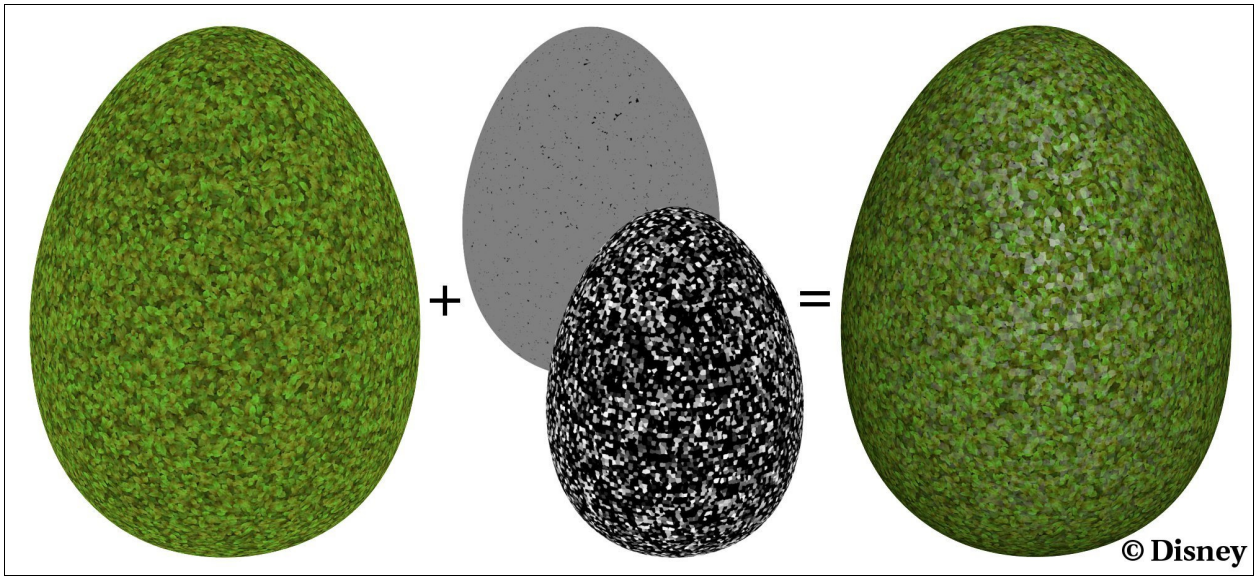


Image 34: Color shader

The displacement is a little trickier. We had to adjust the regular texture access when dealing with the normal maps:

Old displacement map call:

```
texture(mapFile[0], ss, tt, "swidth", 1, "twidth", 1, "filter", "radial-bspline")
```

New normal map call:

```
texture(mapFile, ss, tt, "swidth", 0, "twidth", 0, "filter", "gaussian")
```

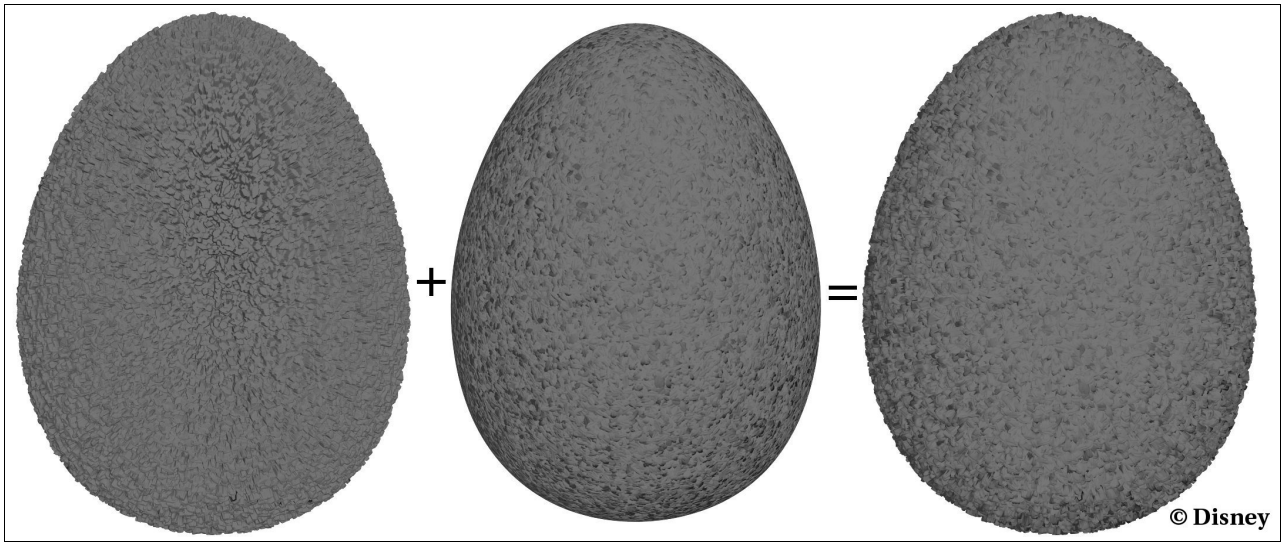


Image 35: Displacement shader.

Since we're accessing a map of normal values, any attempt to filter them will cause problems. Treat them like you would a shadow map. Beyond that, our displacement shader is essentially the same, except we're replacing the normal for the normal map call.

Image 36: Final normal mapped look compared with final XGen.

Step 4. Camera Fade Off

Now we have a very light and fast topiary look that will hold up well in all distant shots without any individual

leaves. We use a camera-based falloff to shift between the hero look (with leaves) and the normal map look (without leaves). The normal map look is actually applied to the surface beneath the leaves, but the displacement is very low for all close up shots.

The shift begins usually at the range where the high resolution renders lose individual leaf distinction. In order to prevent popping, the individual leaves actually start to shrink and are culled out when they get too small. At the same time, we begin to raise the displacement of the underlying normal mapped surface.

Image 37: The leaves as they would appear halfway through the camera falloff.

We chose the final displacement value based on what looks good from a distance and it is considerably higher than the samples we rendered here. In addition, we added an overall noise to put a slight “wonkiness” on the final shape. We found this helped sell the topiary look from afar.

Image 38: Example of "wonkiness"

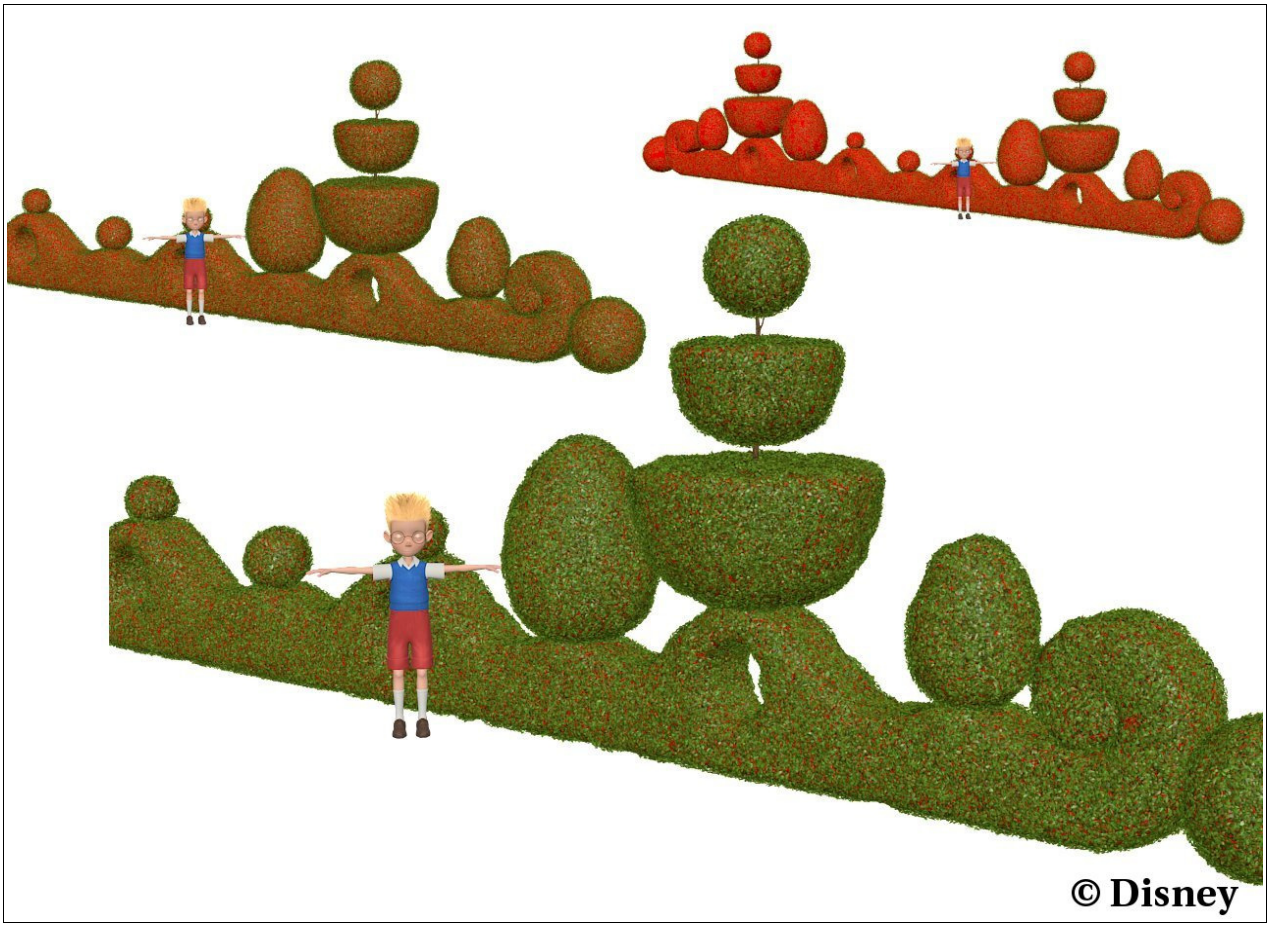


Image 39: Changing the underlying surface color to red to better demonstrate the camera falloff.

Credits

Adrienne Othon, design, research and development

Brent Burley, shader development

Daniel Teece, paint3D development

Appendix I

The s_bakeproperties shader for baking out the three pre-defined layers. This code has been pulled from our module system and is presented as is. It might need some cleanup to work as a stand alone shader.

```
surface s_bakeproperties (

  float bias = 10;

  float useCameraDirection = 0;

  uniform float bakeColor = 1;  /* type switch */
  string colorbakefilename="bake.color.ptc";

  uniform float bakeNormal = 1;  /* type switch */
  string normalbakefilename="bake.normal.ptc";

  uniform float bakeDisp = 1;  /* type switch */
  string dispbakefilename="bake.disp.ptc";

  uniform float bakeLeafMatte = 1;  /* type switch */
  string leafmattebakefilename="bake.leafmatte.ptc";

)

{
    normal Nn = normalize (N);

    color hitcolor = (0,0,0);
    color hitopacity = (0,0,0);

    vector camvec = (0,0,-1);
    vector tvector = useCameraDirection*camvec + (1-useCameraDirection)*Nn;
    tvector = normalize(tvector);

    if(bakeColor == 1)
        {

        hitcolor = (0,0,0);
        hitopacity = (0,0,0);

        /* Replace this with whatever your no-hit color should be */
        color basecolor = Cs;

        /* trace inward, from a point sufficiently "outside" xgen */
        gather("", P + tvector * bias, -tvector, 0, 1,
               "bias", 0.01,
               "surface:Ci", hitcolor,
               "surface:Oi", hitopacity,
               "maxdist", bias,
               "label", "color")
        { /* do this when getting color */
            Ci = hitcolor + (1-hitopacity)*basecolor;
        }
        else /* no ray intersection */
        {
            Ci = basecolor;
        }
```

```
        /* clamp the color range */
        Ci = (clamp(comp(Ci,0),0,1),
              clamp(comp(Ci,1),0,1),
              clamp(comp(Ci,2),0,1));

        /* generate the point cloud file */
        bake3d(colorbakefilename,"_dchannel",P,Nn,"_dchannel",Ci);
    }
    if(bakeDisp == 1)
    {
        hitcolor = (0,0,0);
        hitopacity = (0,0,0);

        /* trace inward */
        gather("",P + bias * tvector, -tvector, 0, 1,
               "bias", 0.01,
               "surface:Ci", hitcolor,
               "surface:Oi", hitopacity,
               "maxdist", bias,
               "label", "disp")
        { /* do this when getting displacement: */
            float dist = length(point(hitcolor) - P * comp(hitopacity, 0));
            Ci = clamp(dist/bias,0,1) * (1,1,1);
        }
        else /* no ray intersection */
        {
            Ci = (0,0,0);
        }
        /* generate the point cloud file */
        bake3d(dispbakefilename, "_dchannel", P, Nn, "_dchannel", Ci);
    }

    if(bakeNormal == 1)
    {
        hitcolor = (0,0,0);
        hitopacity = (0,0,0);

        normal Nw = ntransform("world",Nn);
        color Nc = color (xcomp(Nw)/2+.5,ycomp(Nw)/2+.5,zcomp(Nw)/2+.5);

        /* trace inward */
        gather("", P + bias * tvector, -tvector, 0, 1,
               "bias", 0.01,
               "surface:Ci", hitcolor,
               "surface:Oi", hitopacity,
               "maxdist", bias,
               "label", "normal")
        { /* do this when getting normal: */
            Ci = hitcolor + (1 - hitopacity) * Nc;
        }
        else /* no ray intersection */
        {
            Ci = Nc;
```

```
        }
        /* generate the point cloud file */
        bake3d(normalbakefilename, "_dchannel", P, Nn, "_dchannel", Ci);
    }

    Oi = 1;
}
```

Eye rendering on Meet The Robinsons

Heather Pritchett

The Challenge

The initial eye rig from Chicken Little did all the work in the animator rig. The rig handled all scaling, rotation and even allowed for scaling of the iris and pupil regions and preservation of area, all applied to a single surface courtesy of a complex set of deformers. We painted the final surface and everything worked just fine. What the animator saw was what the animator got.

Problem: The rig was painfully heavy times two, one for each eye.

The rigging department could produce much lighter rigs where the eye surface was a simple constructive solid geometry (CSG) composed of booleans between a sphere and two cylinders pointed out from the center of the sphere. Unfortunately, we couldn't easily paint or render those surfaces.

Solution: The shader would recreate what the animator was seeing. The two cylinders being used to create the iris and pupil booleans lent themselves well to a projected ellipse in the shader.

We determined we would need the following controls:

1. A coordinate system that defines the center of the eye as well as any rotation information.
2. Iris Size, height and width
3. Pupil Size, height and width

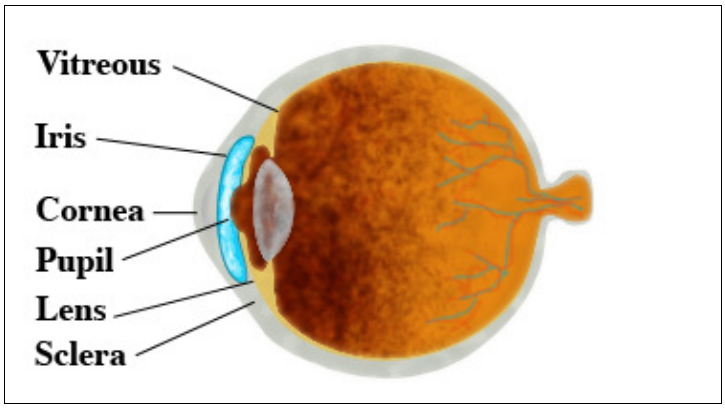


Image 40: Basic eye with labels.

We later added a rotation around the gaze.

The rig controlled the creation of the coordinate systems (one for each eye) and linked them to the animator's controls in addition to tracking the other settings above. They were treated as patch attributes and output directly to the RIB in shot finaling where they overrode the shaders. Using the standard RAT format for specifying primitive variables (see Application Note #22 in your Renderman docs), the maya attributes are:

rmanFirisX	-> irisX
rmanFirisY	-> irisY
rmanFpupilX	-> pupilX
rmanFpupilY	-> pupilY
rmanFrotation	-> rotation

We had default values for these in the look development department, as well as a default position for the coordinate system, but all of these were expected to be replaced by the animator's final settings once the scene arrived in shot finaling.

The Shader

In essence, our iris and pupil are defined as projected ellipses onto a unit sphere. The shader knows the size of each region and maps that back into a single texture map which has hard-coded the eye, pupil and sclera boundaries to s and t texture map calls. First we determine where a point on the surface is in relation to the center of the eye. This just happens to be the z-axis of our coordinate system.

```
// transform sample point into the center pivot coord system
varying point Pt = transform("current", coordSysName, Pbase);

// unit vector from origin pointing toward current sample
varying vector vecP = normalize(vector(Pt));

// angle between current sample and z axis
varying float  angleP = acos(comp(vecP,2));
```

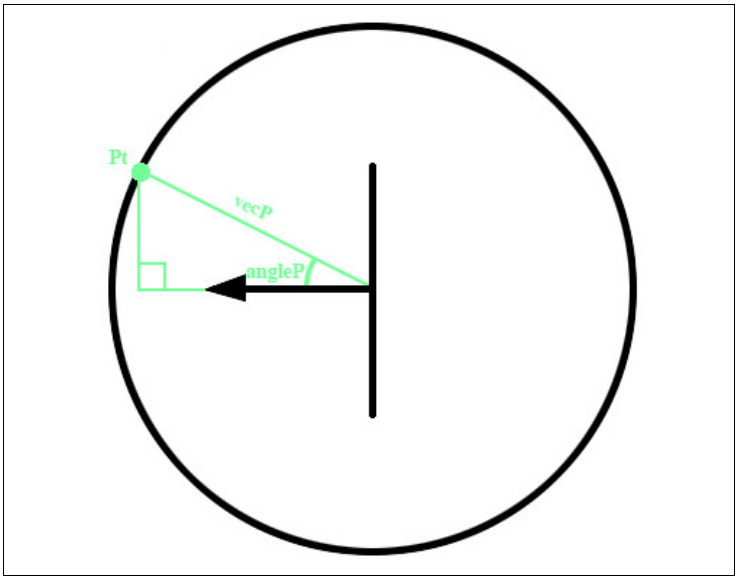


Image 41: Initial projection calculation.

Now we know how the point relates to the center of the eye, so we just need to factor that back into our texture map. Normally, we could just take our angle measurement and go with that, but our process is complicated by the fact that we need to be able to scale certain regions of the eye (the pupil and iris) by arbitrary values.

Given the size of both regions, and assuming that the pupil is always smaller than the iris, we can now predict what region the point is in and map that back to a texture map. First, we have to define the regions:

```
// get regions edge angles in the direction of the current sample
varying float pupilAngle  = ellipseAngle( Pt, pupilX, pupilY, rotation);
varying float irisAngle   = ellipseAngle( Pt, irisX,  irisY, rotation);
```

The function above (provided in the appendix) returns the angles that represent the boundaries between pupil and iris and iris and sclera. Now we simply map the point onto the surface into either the pupil, iris or sclera. Then that value is normalized and remapped into a single texture map. The texture map regions are hard-coded into the shader and all painted maps must conform to those settings.

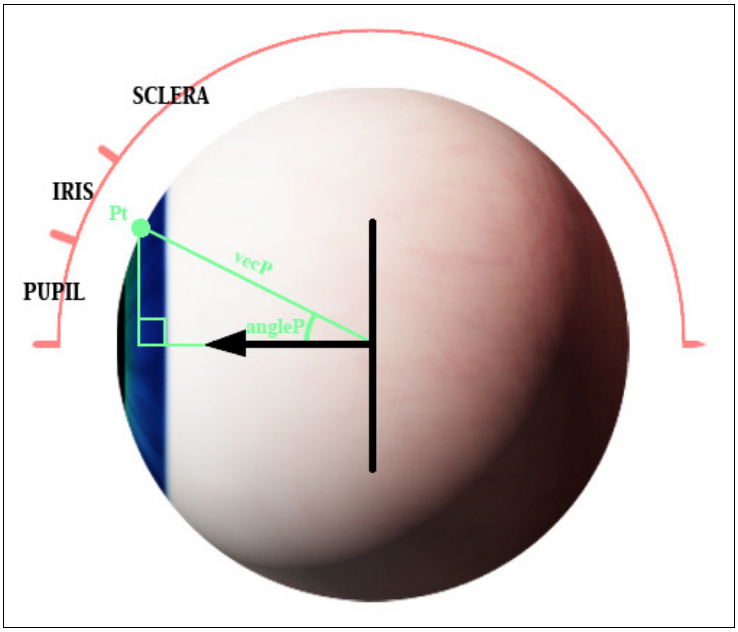


Image 42: Pupil, iris and sclera regions

```
// Compute u texture coordinate

    #define EYEPROJECTION_EPSILON 1e-5

    // color map boundaries
    uniform float COLOR_MAP_PUPIL = 0.1;
    uniform float COLOR_MAP_IRIS = 0.3;

    // initialize u texture coord and region flag
    float angleU =  0;

    if ( angleP > irisAngle ) // point is in sclera (outside projected iris)
      { // determine sclera coords
        // angle between current sample and edge of iris, normalized over the
        // angle between the edge of the iris and back pole of the unit sphere
        if ( abs(PI - irisAngle) > EYEPROJECTION_EPSILON )
          angleU = (angleP - irisAngle)/(PI - irisAngle);

        // remap U texture coord into sclera region of color map
        angleU = COLOR_MAP_IRIS + angleU*(1.0 - COLOR_MAP_IRIS);
      }
     else {
        if ( angleP > pupilAngle ) { // point is inside iris

          // angle between current sample and pupil edge normalized over the
          // angle between the iris overlap edge and the pupil edge
          if ( abs(irisAngle - pupilAngle) > EYEPROJECTION_EPSILON )
            angleU = (angleP - pupilAngle)/(irisAngle - pupilAngle);
```

```
      // remap U texture coord into iris region of color map
      angleU = COLOR_MAP_PUPIL + angleU*(COLOR_MAP_IRIS - COLOR_MAP_PUPIL);
    }
    else { // point is inside pupil
      // normalized angle between current sample and z-axis
      if ( abs(pupilAngle) > EYEPROJECTION_EPSILON )
        angleU = angleP/pupilAngle;

      // remap U texture coord into pupil region of color map
      angleU *= COLOR_MAP_PUPIL;
    }
  }
  map_ss = angleU;
```

Using the angle between the z-axis (the center gaze) and the point, we can now determine where the point is mapped into our eye texture map as far as “t”.:

```
// get v texture coord (same for all regions)
varying float angleV = atan(comp(Pt,0),comp(Pt,1));
map_tt = (angleV + PI)/(2*PI); // normalize angle
```

Now that we have the s and t values, the texture map reference is handled as normal.

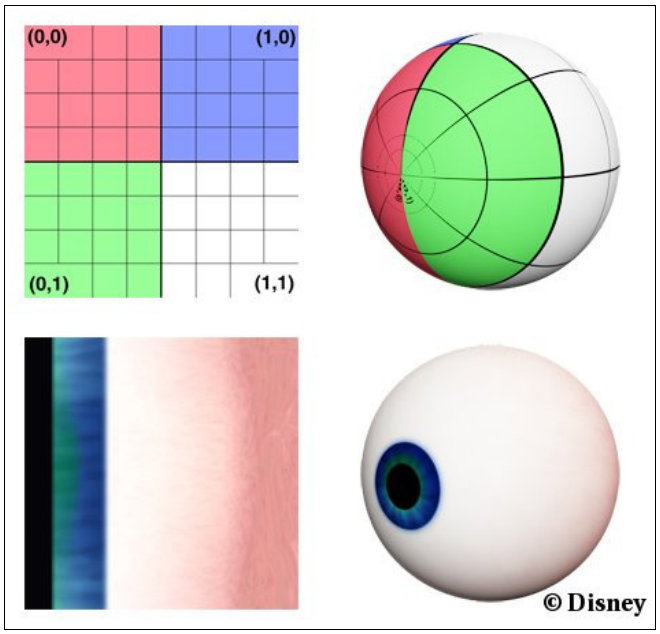


Image 43: Sample texture maps.

It's important to note that this projection technique is not limited to just perfectly round eyeballs. While the initial projections are done on a unit sphere, we can project the resulting eye regions on any final shape. Spheres work best and the closer a shape is to spherical, the less the projection will deform.

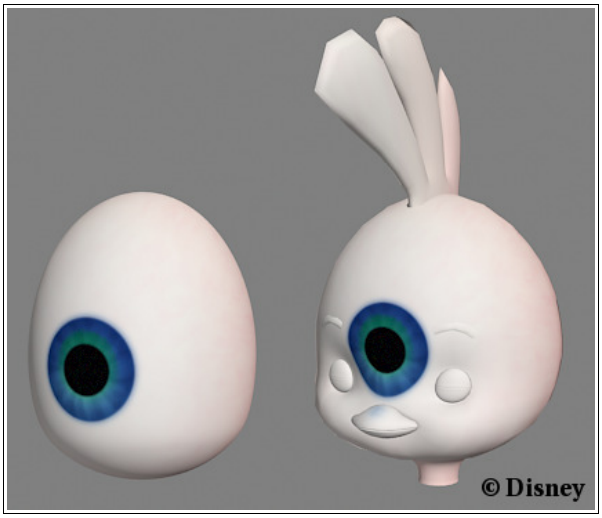


Image 44: Different eye shapes.

Implementation Details

Each eye is represented by two spheres. The main eyeball sphere contains the pupil, iris and sclera regions as defined above. A second, slightly larger, sphere is used to represent the cornea. It is rendered completely transparent except for reflection and specular highlights.

We use displacement on both spheres. On the eyeball itself, we displace the iris and pupil down to more accurately reflect what a real eye looks like. We found that skipping this step made the eyes look odd and threw the gaze off. Oddly enough, the eye rig used by the animators doesn't have this displacement. However, the rotations are set in place by a locator that defines what the character is actually looking at. They don't have to fine-tune the gaze themselves, they just have to define where it's going.

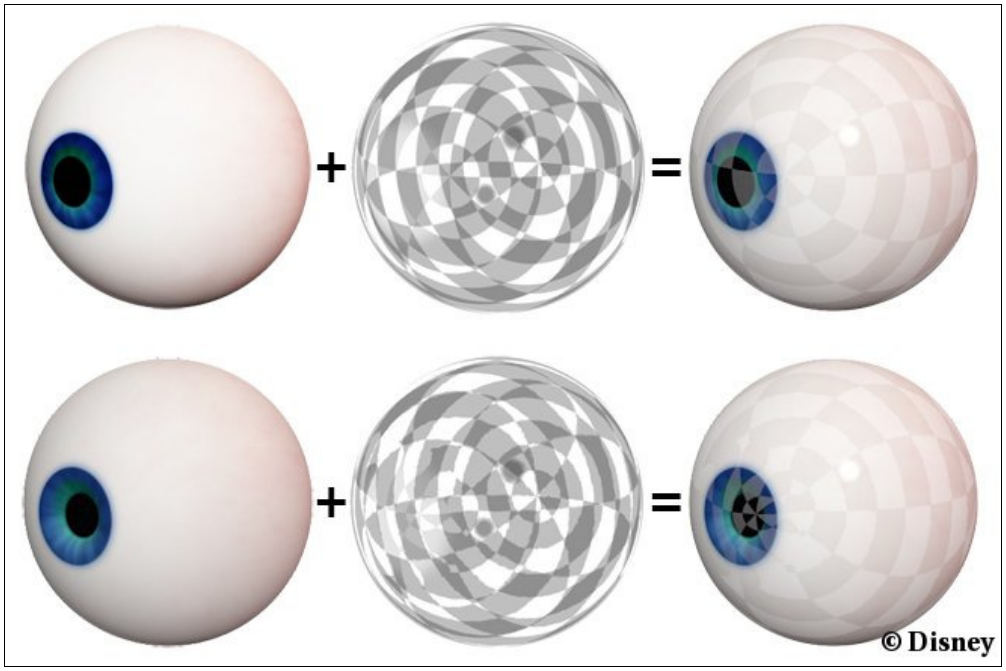


Image 45: Both spheres with final look, displaced and un-displaced.

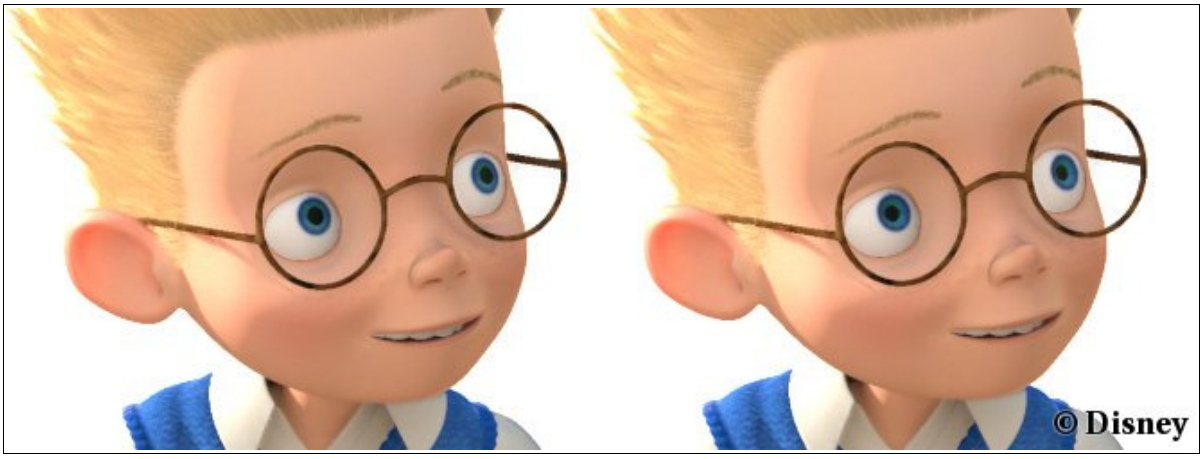


Image 46: Before and after of Lewis gaze.

In addition, we have a bump map on the cornea to define the corneal bump, as well as a little bumpy noise. We opted for bump over true displacement to prevent any inner-penetration issues with the eyelids. The bump map gives us a nice breakup of the specular directly over the eye, much like a true cornea.

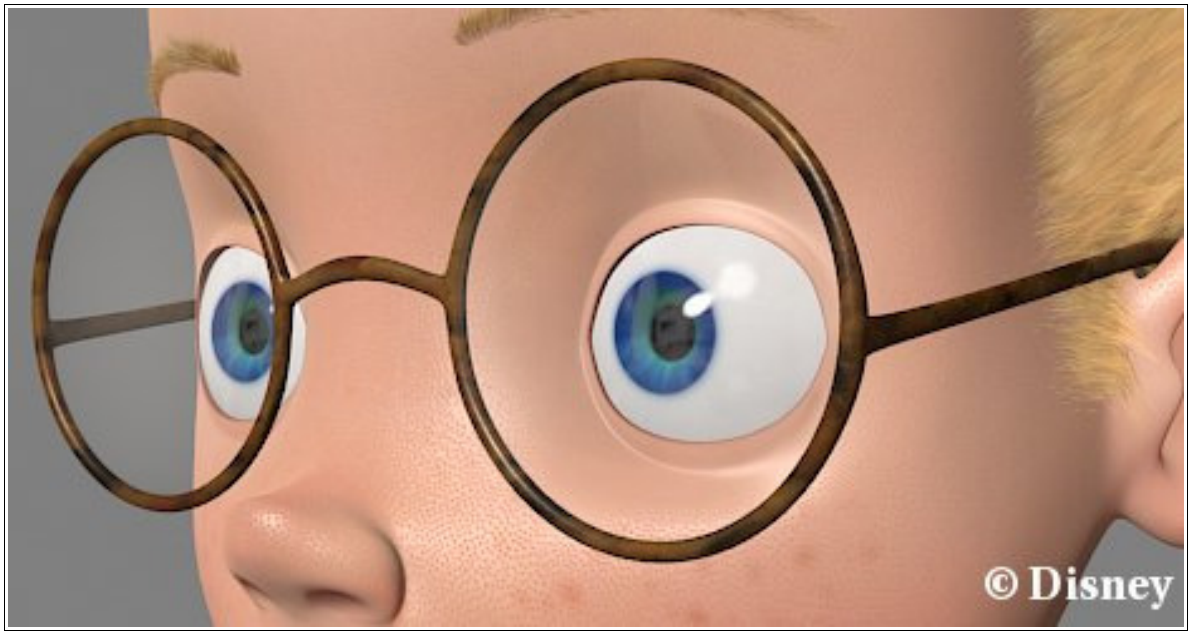


Image 47: Corneal bump.

Further Work

This presents the core functionality behind our eye projection technique. The basic concept of projection from the center of the eye enabled us to add many features beyond simple texture projection. We added several features to soften and blend the edges between the pupil, iris and sclera. In addition, future productions have expanded the shader module to include:

- Procedural displacement of the iris and sclera to avoid displacement artifacts from extraordinary vertices in our current maps.
- Adding a vector offset to the specular highlight so the lighting artist can nudge it to the left or right without having to fiddle with the light.
- Adding shape controls to the specular highlight so it can be as exactly round or oblong or whatever shape needed by the lighting artist.
- Adding specular highlight on the opposite side of the underlying iris to produce a rich caustic-like effect. This can also be nudged with a vector offset.
- Adding a scale factor to the shadow calculation so it is, in effect, cheated in producing a dropped shadow along the inside of the eyelid.

- Similar, since our coordinate systems have a concept of up, we can rotate the shadow map to produce a heavier drop shadow on just the upper lid.

Finally, the development came full circle when we altered the animator's rig to better match what we were rendering. Instead of using CSG geometry, we created a hardware shader plugin in Maya that mirrored the RenderMan shader projection technique precisely. The plugin actually incorporates Nvidia's Cg shader language into Maya and allows us to drive the shader controls with attributes. In this case, the same rmanFirisX that is driving the RenderMan shader. We're even referencing the same default texture file. This proved to be more accurate than the previous solution and faster as well.

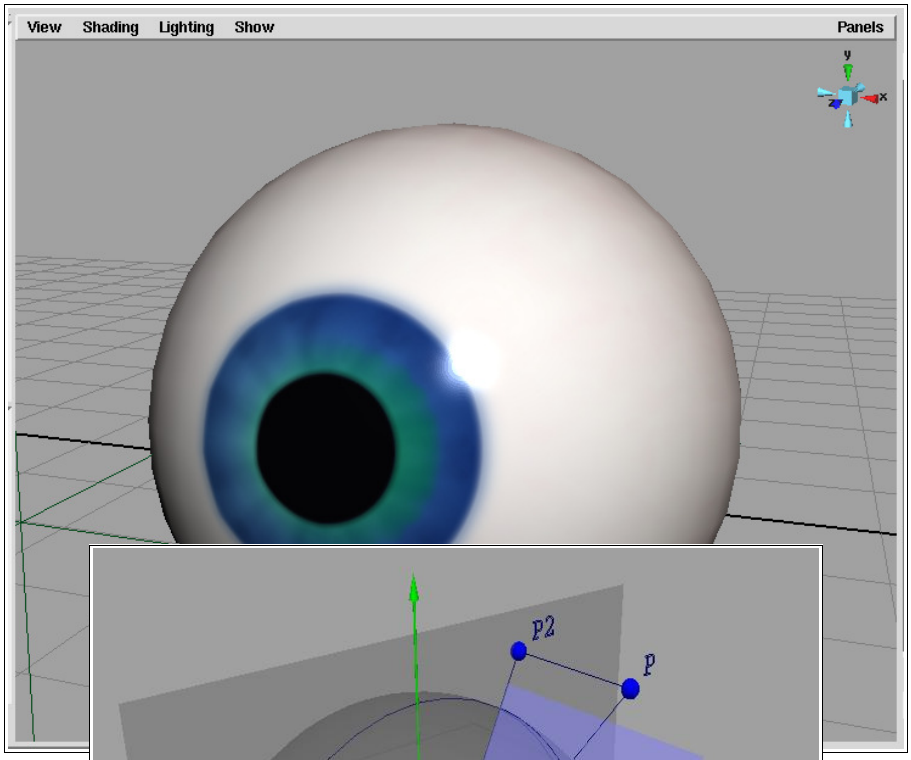


Image 4

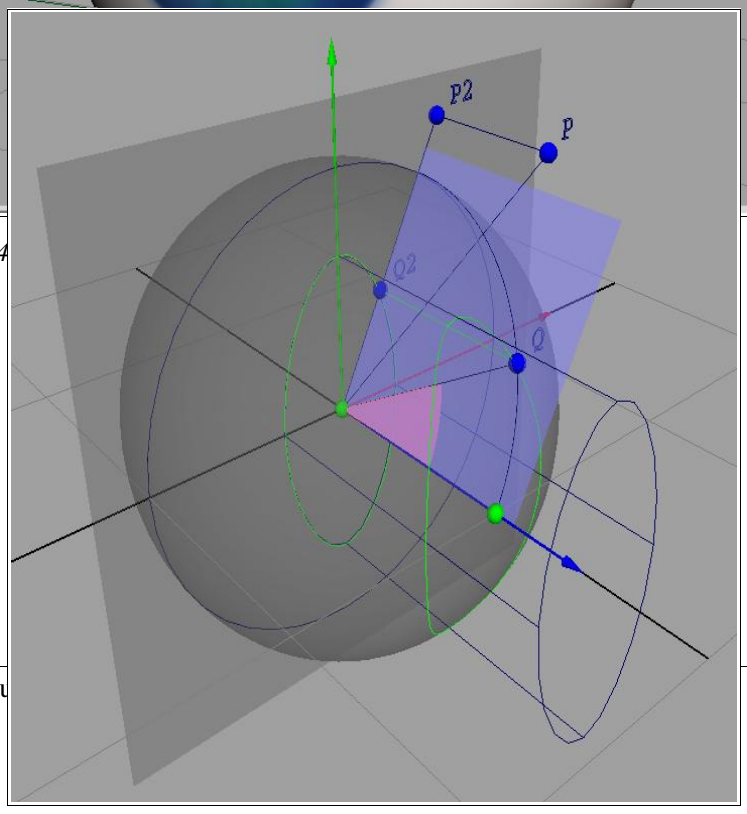


Image 49: A visual representation of the ellipseAngle() calculations.

Credits

Lewis Siegel, initial design and prototyping

Liza Keith, module implementation and development

Chris Springfield, further development

Arthur Shek, hardware shader

Appendix I

Function to determine size of iris and pupil regions. Everything else is sclera.

```
float ellipseAngle( varying point Pt;  // sample point in local space
                    varying float a;   // x radius of ellipse; assumed < 1.0
                    varying float b;   // y radius of ellipse; assumed < 1.0
                    uniform float rot; // rotation angle of ellipse about z axis
                  )
{
    // Given a point Pt in 3-space, and an ellipse in the xy
    // plane, determine the angle between vector V and the z-axis
    // where:
    //
    //   V is the normalized vector from the origin to point Q
    //
    //   Q is the projection along the z-axis of point Q2 in the
    //   xy-plane onto the unit sphere
    //
    //   Q2 is the intersection of the ellipse and the vector
    //   from the origin to the point P2
    //
    // and
    //
    //   P2 is point P projected into the xy plane along the z-axis.

    varying float  X  = comp(Pt,0);
    varying float  Y  = comp(Pt,1);

    if (mod(abs(rot),360) != 0 ) {
        float tmpX = X;
        float tmpY = Y;
        float rad = radians(rot);
        ROTATE2D(tmpX, tmpY, rad, 0.0, 0.0, X, Y);
    }

    varying float angle = 0;
```

```
    if (X==0 && Y==0) angle = 0;
    else if (X==0 && Y!=0) angle = acos(sqrt(1-b*b)); // Pt aligned with y-axis
    else
    { // the general case
        varying float m = Y/X;
        varying float x = (a*b)/sqrt(b*b + a*a*m*m);
        varying float y = m*x;
        varying float z = sqrt(1 - (x*x + y*y));
        angle = acos(z);
    }

    return angle;
}
```

Appendix II

Hardware shader code (three files).

```
// eyeVS.cg (vertex shader)

struct inputData
{
    float4 position : POSITION;
    float3 normal   : NORMAL;
};

struct outputData
{
    float4 HPosition : POSITION;
    float3 position  : TEXCOORD0;
    float3 normal    : TEXCOORD1;
    float3 objPos    : TEXCOORD2;
        float3 worldPos  : TEXCOORD3;
        float4 wPosition : TEXCOORD4;
    float3 wNormal   : TEXCOORD5;
    float3 wEyeDirection : TEXCOORD6;
};

outputData main( inputData IN,
                 uniform float4x4 ModelViewProj,
                 uniform float4x4 ModelView,
                 uniform float4x4 ModelViewIT,
                 uniform float4x4 ObjectToWorld,
                 uniform float4x4 ObjectToWorldIT,
                 uniform float4x4 ViewToWorldIT,
                 uniform float4x4 cgCoordSysMatrix )
{
    outputData OUT;
    OUT.position  = mul( ModelView, IN.position ).xyz;
    OUT.normal    = mul( ModelViewIT, float4( IN.normal, 0 ) ).xyz;
    OUT.objPos    = IN.position;
    float4 temp = mul( ObjectToWorld, IN.position );
    float3 worldPos = mul( cgCoordSysMatrix, temp ).xyz;
    OUT.worldPos   = worldPos;
    float displacement = 0;
    OUT.HPosition = mul( ModelViewProj, IN.position );
    OUT.wPosition = mul( ObjectToWorld, IN.position );
    OUT.wNormal = normalize(mul( ObjectToWorldIT, float4( IN.normal, 1 ) ).xyz );
    float3 eyeDir = normalize( -mul( ModelView, IN.position ).xyz );
    OUT.wEyeDirection = normalize(mul(ViewToWorldIT, float4( eyeDir, 1 ) ).xyz );

    return OUT;
}
```

```
// eyePS.cg (pixel shader)

#include "mayaLight.cg"

float EYEPROJECTION_EPSILON = 1e-30;
float PI = 3.14159265358979323846264338327795;

float ellipseAngle( float3 Pt, float a, float b ) {
        float X = Pt.x;
        float Y = Pt.y;

        float angle = 0;

        if ( X==0 && Y==0 )
                angle = 0;
        else if ( X==0 && Y!=0 )
                angle = acos(sqrt(1-b*b));
        else {
                float m = Y/X;
                float x = (a*b)/sqrt(b*b + a*a*m*m);
                float y = m*x;
                float z = sqrt(1 - (x*x + y*y));
                angle = acos(z);
        }

        return angle;
}

struct inputData
{
        float4 HPosition : POSITION;
        float4 position  : TEXCOORD0;
        float3 normal    : TEXCOORD1;
        float3 objPos    : TEXCOORD2;
        float3 worldPos  : TEXCOORD3;
        float4 wPosition : TEXCOORD4;
    float3 wNormal   : TEXCOORD5;
    float4 wEyeDirection : TEXCOORD6;
};

float4 main( inputData IN,
            uniform mayaLight light,
            uniform float cgIrisScaleX = 1,
            uniform float cgIrisScaleY = 1,
            uniform float cgPupilScaleX = 0.5,
            uniform float cgPupilScaleY = 0.5,
            uniform float cgShininess = 500,
            uniform float cgLightScale = 1,
            uniform float3 cgcPupilColor = float3(0.019607843, 0.019607843,
0.031372549),
            uniform sampler2D cgTexture ) : COLOR
{
```

```
        float3 N = normalize( IN.normal );
        float3 color = float3(0,1,0); // green == error

        // unit vector from origin pointing toward current sample
        float3 vecP = normalize( IN.worldPos );

        // angle between current sample and z axis
        float angleP = acos(vecP.z);

        // get regions edge angles in the direction of the current sample
        float pupilAngle = ellipseAngle( IN.worldPos, 0.3*cgPupilScaleX,
0.3*cgPupilScaleY );
        float irisAngle = ellipseAngle( IN.worldPos, 0.3*cgIrisScaleX,
0.3*cgIrisScaleY );

        // get v texture coord (same for all regions)
        // float angleV = atan2(IN.worldPos.x, IN.worldPos.y);
        float angleV = atan2(IN.worldPos.y, IN.worldPos.x);
        angleV = (angleV + PI)/(2*PI); // normalize angle
    // loop back around to texture to prevent boundary artifacts (should be
better way)
        if ( angleV > 0.5 )
                angleV = 1-angleV;

        // initialize u texture coord and region flag
        float angleU = 0;

        if ( angleP > irisAngle ) {
                if ( abs(PI - irisAngle) > EYEPROJECTION_EPSILON )
                        angleU = (angleP - irisAngle)/(PI - irisAngle);

                // remap U texture coord into sclera region of color map
                angleU = 0.3 + angleU*(1.0-0.3);
        } else {
                if ( angleP > pupilAngle ) // point is inside iris
                {
                        if ( abs(irisAngle - pupilAngle) >
EYEPROJECTION_EPSILON )
                                angleU = (angleP - pupilAngle)/(irisAngle -
pupilAngle);

                        // remap U texture coord into sclera region of color map
                        angleU = 0.1 + angleU*(0.3-0.1);
                }
                else
                { // point is inside pupil
                        if ( abs(pupilAngle) > EYEPROJECTION_EPSILON )
                                angleU = angleP/pupilAngle;

                        angleU *= 0.1;
                }
        }
```

```
    // prevents artifacts at center of pupil (texture boundary)
        if ( angleP < pupilAngle/2 )
                color = cgcPupilColor;
        else
                color = tex2D( cgTexture, float2(angleU, angleV) );

        float3 L;
        half3 lightColor = light.illuminate( IN.wPosition.xyz, L );

        N = normalize( IN.wNormal );
        half NdotL = dot( N, L );

        half3 E = normalize( IN.wEyeDirection );
        half3 H = normalize( E + L );
        half NdotH = dot( N, H );

        float4 litV = lit( NdotL, NdotH, cgShininess );

        color = ( cgLightScale * lightColor * color * litV.y ) + ( litV.y *
litV.z * lightColor * float3(1, 1, 1) );

        return float4( color.xyz, 1 );
}
```

```
// mayaLight.cg

#ifndef mayaLight_cg
#define mayaLight_cg


float4 getLightProjection( float4x4 matrix, float4 position )
{
    float4 lp = mul( matrix, position );
    lp.xyz /= lp.w;
    lp.w = 1.0;
    return lp;
}


struct mayaLight
{
    #ifdef AMBIENT_LIGHT
        float  _ambientShade;
        float3 _color;
    #endif
    #ifdef POINT_LIGHT
        float3 _position, _color;
        float4 _lightDecayCoeff;
    #endif
    #ifdef SPOT_LIGHT
```

```
        float3 _position, _direction, _color;
        float4 _lightDecayCoeff;
        float  _cosPenumbra;
        float  _cosUmbra;
        float  _radialDropOff;
    #endif
    #ifdef DIRECTIONAL_LIGHT
        float3 _position, _direction, _color;
        float4 _lightDecayCoeff;
    #endif
    #ifdef SHADOW_MAP
        sampler2D _shadowMap;
    #endif
    #ifdef PROJECTION_MAP
        sampler2D _projectionMap;
    #endif


    float3 illuminate( float3 wp, out float3 lightDir )
    {
        float3 lightColor = _color;

        #ifdef AMBIENT_LIGHT
            return lightColor * _ambientShade;
        #endif

        #ifdef POINT_LIGHT
            float3 lightVector = _position - wp;
            lightDir = normalize( lightVector );

            float lightDistance = length( lightVector );

            float attenuation =_lightDecayCoeff.x +
                               _lightDecayCoeff.y * lightDistance +
                               _lightDecayCoeff.z * lightDistance *
                               lightDistance + _lightDecayCoeff.w  *
                               lightDistance * lightDistance * lightDistance;

            attenuation = max( attenuation, 1 );

            return ( lightColor * ( 1.0 / attenuation ) );
        #endif

        #ifdef SPOT_LIGHT
            float3 lightVector = _position - wp;
            lightDir = normalize( lightVector );

            float lightDistance = length( lightVector );

            float attenuation =_lightDecayCoeff.x +
                               _lightDecayCoeff.y * lightDistance +
                               _lightDecayCoeff.z * lightDistance *
                               lightDistance +_lightDecayCoeff.w *
```

```
                            lightDistance * lightDistance * lightDistance;

        attenuation = max( attenuation, 1 );

        float SdotL = dot( lightDir, _direction );
        float spotRadialIntensity = 1;

        if ( SdotL < _cosPenumbra )
        {
            spotRadialIntensity = 0;
        }
        else
        {
            spotRadialIntensity = pow( SdotL, _radialDropOff );

            if ( SdotL < _cosUmbra )
            {
                    spotRadialIntensity = spotRadialIntensity *
                                          ( SdotL - _cosPenumbra ) /
                                          ( _cosUmbra - _cosPenumbra );
            }
        }


        return ( lightColor * spotRadialIntensity / attenuation );
    #endif

    #ifdef DIRECTIONAL_LIGHT
        float3 lightVector = _position - wp;
        float  lightDistance = length( lightVector );

        float attenuation =_lightDecayCoeff.x +
                           _lightDecayCoeff.y * lightDistance +
                           _lightDecayCoeff.z * lightDistance *
                           lightDistance + _lightDecayCoeff.w *
                           lightDistance * lightDistance * lightDistance;

         attenuation = max( attenuation, 1 );

        lightDir = normalize( _direction );

        return ( lightColor * ( 1.0 / attenuation ) );
    #endif
}


float3 illuminate( float3 wp, float4 lp, out float3 lightDir )
{
    #ifdef PROJECTION_MAP
      float3 lightColor = tex2D( _projectionMap, float2( lp.x / lp.w,
                                 lp.y / lp.w ) );
    #else
      float3 lightColor = _color;
```

```
#endif

#ifdef SHADOW_MAP
    float3 Dt = tex2D( _shadowMap, float2( lp.x / lp.w, lp.y / lp.w ) );
    float R  = lp.z / lp.w;
    float shadow = R <= Dt.x ? 1.0 : 0.0;
#else
    float shadow = 1.0;
#endif


#ifdef AMBIENT_LIGHT
    return lightColor * _ambientShade;
#endif

#ifdef POINT_LIGHT
    float3 lightVector = _position - wp;
    lightDir = normalize( lightVector );

    float lightDistance = length( lightVector );

    float attenuation = _lightDecayCoeff.x +
                        _lightDecayCoeff.y * lightDistance +
                       _lightDecayCoeff.z * lightDistance *
                       lightDistance + _lightDecayCoeff.w *
                       lightDistance * lightDistance * lightDistance;
    attenuation = max( attenuation, 1 );

    return shadow * ( lightColor * ( 1.0 / attenuation ) );
#endif

#ifdef SPOT_LIGHT
    float3 lightVector = _position - wp;
    lightDir = normalize( lightVector );

    float lightDistance = length( lightVector );

    float attenuation = _lightDecayCoeff.x +
                        _lightDecayCoeff.y * lightDistance +
                       _lightDecayCoeff.z * lightDistance *
                       lightDistance + _lightDecayCoeff.w *
                       lightDistance * lightDistance * lightDistance;
    attenuation = max( attenuation, 1 );

    float SdotL = dot( lightDir, _direction );
    float spotRadialIntensity = 1;

    if ( SdotL < _cosPenumbra )
    {
        spotRadialIntensity = 0;
    }
    else
    {
```

```
                spotRadialIntensity = pow( SdotL, _radialDropOff );

                if ( SdotL < _cosUmbra )
                {
                        spotRadialIntensity = spotRadialIntensity *
                                              ( SdotL - _cosPenumbra ) /
                                              ( _cosUmbra - _cosPenumbra );
                }
            }


            return shadow * ( lightColor * spotRadialIntensity / attenuation );
        #endif

        #ifdef DIRECTIONAL_LIGHT
            float3 lightVector = _position - wp;
            float  lightDistance = length( lightVector );

            float attenuation = _lightDecayCoeff.x +
                                 _lightDecayCoeff.y * lightDistance +
                                _lightDecayCoeff.z * lightDistance *
                                lightDistance + _lightDecayCoeff.w *
                                lightDistance * lightDistance * lightDistance;
            attenuation = max( attenuation, 1 );

            lightDir = normalize( _direction );

            return shadow * ( lightColor * ( 1.0 / attenuation ) );
        #endif
    }
};


#endif//mayaLight_cg
```

Ambient Occlusion as Shadowing

Tal Lancaster

The concept of ambient occlusion has been around in the graphics community for some time. Now most renderers have support for doing this in some form or another. The way that ambient occlusion is traditionally handled is through rendering NDC maps(screen space) in a separate pass. These maps are then used in at the compositing stage after the final renders are completed. Examples of these more traditional methods of ambient occlusion (along with reflection occlusion) can be found in the SIGGRAPH 2002 RenderMan course #15, "Production-Ready Global Illumination", Hayden Landis from ILM pp. 87-97.

An issue with doing ambient occlusion in compositing is that the lighter doesn't know what impact the ambient occlusion is going to have on their light set up until later in the comp phase. There is nothing really wrong with doing it this way, it is only a question of feedback and if this delay is acceptable.

On Meet The Robinsons production, the lighters wanted to have a more immediate/direct feedback. This meant that they wanted the ambient occlusion (and reflection occlusion) data used directly by the lights. This way they would see what effect the occlusion was having on their light setup while they were still working with their lights.

Some perks, with having the occlusion used directly in the lighting, are that different adjustments can be made on a per material (or rather per lights) much easier. For example through the use of light linking (assigning lights to be only be available to specific objects), each light could interpret the occlusion maps differently to create object specific looks. Even directed lighting (i.e. Spotlights) can use the occlusion data to simulate soft lights. The lights have controls to adjust the tinting in the highs and lows. Providing the control to make the occlusions warmer in the highlight areas and cooler tones in the darker areas. The effects can be as dramatic or subtle as the situation calls for it. Also with these controls for the ambient occlusions in the lights, we have found situations where only a small fraction of lights are needed compared to what we would have done traditionally. One last item some of these most likely could be done through the compositing method, but it would have required, breaking out the elements into more layers and doing more render passes and having more data to manage. While being able to control them through the lights seems to be more intuitive and easier to manage.

Below are example images from Walt Disney's "*Meet The Robinsons*" using this occlusion technique.

Image 50: Occlusion © Disney

Image 51: Ambient light (traditional) © Disney

Image 52: Ambient light using occlusion in render © Disney

Implementing this capability meant a large alteration to every lighting module (material property) and light model (light shaders) in our library. It wasn't enough to just have, say, the ambient properties (and lights) effected. The desire was essentially to treat the occlusions as special shadows. There was also the desire to not just have "ambient" occlusions, but to allow for directional occlusions by letting non-ambient lights work with this data, too. We needed to allow for diffuse, specular, and reflection categories to be affected by the ambient occlusion data. Any light needed the capability of reading both camera and non-camera based baked maps.

The non-camera based maps are brick maps or point cloud data that were converted into parametric UV maps. These sets of maps are called baked maps. While the camera based maps are referred to as NDC (short for Normalized Device Coordinates) maps. (Yes, technically NDC maps are baked maps, too. Don't ask.) The NDC allowed for frame dependent occlusion (ie. for moving objects). While the baked maps were used for static objects.

The implementation required that this occlusion data affect `Cl`, much like a shadow does. This required a very close relationship/dependency between the light shaders and the surface shaders. Part of the reason for this symbiosis was that the maps were potentially parametric. As light shaders don't naturally have parametric surface data, there were essentially two approaches. The first was to pass a bunch of parameters down from the light and have a ton of duplicate code replicated throughout every lighting call (illuminance). The other was to send up varying surface data up to the light so it can perform the lighting calculations with occlusion.

The chosen design was to have the baked maps read once from the surface shader and then to have this information passed up to the light shader. Then the light shader would be responsible for making use of this baked map data along with the NDC maps to modify the light's `Cl` output value. The light then goes back to the surface shader within each lighting module's illuminance loop.

Here is a snipped of code for the diffuse lighting module:

```
uniform string cacheState = "reuse";

/* The baked map was read earlier and stored in the the ambOccbm_val.
   __ambOccl is an user attribute to say if this surface shader is
   to wants ambient occlusion data or not.  If not then the lightcache
   is not refreshed for efficency reasons. */

extern uniform float __ambOccl;
extern float ambOccbm_val;

/* Now only refreshing cache if __ambOccl is > 0 */
if (__ambOccl > 0)
```

```
    cacheState = "refresh";

 illuminance (   "traditionalLight", P, Nn, PI,
                 "send:light:__ambOccl__", __ambOccl,
                 "send:light:occlBM_val", ambOccbm_val,
                 "lightcache", cacheState) {

/* your favorite diffuse code here */
}
```

ambOccbm_val contains the baked map data. It is a global variable and contains occlusion from either a brickmap or a UV parametric map. Static geometry typically will have this type of data. The map data is read once in the surface shader and the value is stored into this variable. Then all of the illuminance statements can pass this on to a light shader.

__ambOccl is a state variable. It is also a global variable, set via an user attribute statement. It tells the shaders if the current render should make use of ambient occlusion or not. Besides controlling which sections of the shader are to be active or not, it also triggers the cache state of the lights via *cacheState*.

cacheState tells an illuminance statement if the light is to be cached or not -- ie. “reuse” (cached), or “refresh” (not cached). One of PRMan's many efficiencies is to have cached lights, which means that if a surface shader has multiple illuminance loops (which includes the calls difffuse() and specular()), the light shaders cache their results, which in turn are reused by the other illuminance loops. Turning off the cache shouldn't be taken lightly as it will not make your renders as efficient as having them cached. However, if one needs to send up varying data, then you have no choice.It wasn't until very recently that Pixar provided this option to select between the cache states (circa. PRM-12.5). Before this option was provided, there was another way to disable the cache:

```
//within light shader
illuminate () {
  P = P;
}
```

changing P inside a light shader's illuminate loop will result in flushing this cache. The “lightcache” reuse/refresh is a much better alternative in that it provides one with the option of being selective from the surface shader's point of view to decide if a flushing of the cache is warranted.

Another thing to take note of is that sometime in the late '90s PRMan made sure that the illuminance statement (along with specular() and diffuse() calls) properly ignore ambient lights. Before this change

it was common for shader writers to have a test in their lighting blocks to compare the vector L against (0, 0, 0). If it was equal to (0, 0, 0) then this meant it was an ambient light and that normally much of the luminance block would be skipped.

So what was the purpose of that trip on the way-back machine for that piece of trivia? It means that light categories and message passing (MP) do not work with the traditional ambient light and the RSL ambient() call. We wanted the occlusion data to effect all materials and lights, this included ambient lights. Due to the issues associated with ambient that were just mentioned, we had to move away from the RSL ambient() call and the standard code of creating an ambient light.

Remember from the RenderMan spec that any light that doesn't have a solar or illuminate statement is considered ambient. So while the standard ambient light looks like:

```
light
ambientlight(
    float intensity=1;
    color lightcolor=1;
)
{
    L  = (0,0,0);
    Cl = intensity * lightcolor;
}
```

Our minimal ambient light would look like this:

```
light
l_ambient(
    float intensity=1;
    color lightcolor=1;
    string __category = "ambientClass";
)
{
    L  = (0,0,0);
    solar() {
        Cl = intensity * lightcolor;
    }
}
```

We rely heavily on the category feature of RenderMan. It allows us to have very specialized light behavior and to have very specific surface material behavior that is looking at only a particular set of lights. A disadvantage is that the lights and surface shaders tightly bound together. Making it harder to use light shaders that aren't our

own. Lights like the standard ambient or distantlight for instance.

Here is a snippet another ambient light:

```
light l_ambOccMP (
    float intensity = 1;
    color lightcolor = color 1;
    string __category = "ambientMP,ambientClass";
    varying float occlBM_val = 0;
              . . .
) {. . .}
```

In this case this light is both an `ambientClass` light, like the previous shader. But we are also saying it is an `ambientMP`. The `ambientClass` represents the general ambient light family. In the case of the l_ambient shader, it is a close to a traditional ambientlight as we can get. The l_ambOccMP light is something more sophisticated. The addition of the `ambientMP` to the `__category` allows us to signify that this class of ambient light is capable of message passing (in this case sending surface shader information up to the light shader).

Here is a snippet of our ambient material model:

```
if (__ambOccl > 0)
        cacheState = "refresh";

illuminance ("ambientMP", P,
    "send:light:__ambOccl__", __ambOccl,
    "send:light:occlBM_val", ambOccbm_val,
    "lightcache", cacheState) {

    C += Cl;
}

/* Ambient lights but without MP (Message Passing), so don't have to worry
       about refreshing the cache. */
illuminance ("-ambientMP&ambientClass", P)
{
    C += Cl;
}
```

The first illuminance block looks for our Message Passing ambient lights. The ones that have a `__category` set to "`ambientMP`".It is those lights that we send the varying data `ambOccbm_val` up to so they can make use of the ambient occlusion data and modify the `Cl` that the light sends back to us int that first illuminance block.

In the second block we are looking for any `ambientClass` light that is not an `ambientMP` light. Meaning any simple ambient lights (ones that don't have the MP capability and are not influenced by ambient occlusion) . Also these lights always get to reuse their cache.

Here is a more developed example of "ambient" light with message passing (the code only lists a portion of the NDC section and the baked section is completely missing):

```
light l_ambOcclMP (

    float intensity = 1;
    color lightcolor = color 1;

    /* Skip the NDC section when bypass == 1*/
    float occlNDC_bypass1 = 1;
    string occlNDC_Map1;

    /* Flip the NDC map value if invert == 1 */
    float occlNDC_MapInvert1 = 0;

       /* Note that the light has two ambOccl variables.
    1. __amOccl is set by the light and sends this value to the
    surface shader.
    2. __ambOccl__ is set by the surface shader, via it's __ambOccl
    and passed to the light shader.
    */
    output uniform float __ambOccl__ = 0;
    output uniform float __ambOccl = 1;

    string __category = "ambientMP,ambientClass";
)
{
    float Foccl_1 = 0;

    /* Go into NDC space */
    point Pndc = transform("NDC", Puse);
    float x = xcomp(Pndc);
    float y = ycomp(Pndc);
    uniform float foundAmbOccl = 0;

    /* Only do NDC occlusion if all of these are met */
    if (OcclNDC_Map1 != "" && OcclNDC_bypass1 == 0 &&
        __ambOccl__ > 0 && __ambOccl > 0) {
        Foccl_1 =
        float texture ((OcclMap)[0], x, y, "blur",
                    occl_blur, "filter", "gaussian");
        Foccl_1 = (OcclNDC_MapInvert1 == 0) ? Foccl_1 : 1-Foccl_1; /* invert */
        foundAmbOccl = 1;
    }
    /* NOTE: May want some code to tint, gamma, or remap the occlusion
       before using it */
```

```
    solar () {
    /* Using pointlight mode.  But shouldn't really matter */
    Cl = intensity * lightcolor;

    if (foundAmbOccl == 1)
        Cl *= Foccl_1;
    }
}
```

For about four months or so we were functioning with our ambient occlusions with light shaders similar to the one just listed. However, every so often we would get frames with strange artifacts when using NDC maps. There were two issues in particular that needed to be solved. One was an edge haloing that occurred near the screen edges. The second was a haloing along edges of character's faces as they moved in a way the revealed geometry that was hidden in the prior frame. Image 53 is an example of the second case. Note the band just to the left of the green sphere.

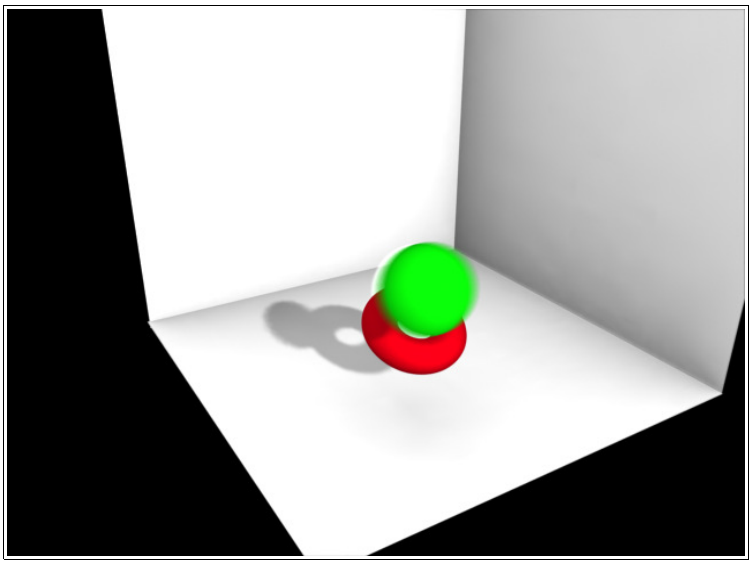


Image 53: halo left of green sphere (2nd issue)

We thought that the artifacts for the 1st problem were a result of not rendering the NDC maps with motion blur. We did regenerate some of the NDC maps with motion blur. This did help some frames, but for others the artifact got worse.

Another important issue thing is that as these were being rendered as NDC maps, they were generally rendered a 1280x700. When these renders were being converted into texture maps, this was being done with the command:

```
% txmake -mode clamp -resize up-
```

This is essentially the command line that we use when creating texture maps based off of our parametric maps.

Now is a good time to pause and have a review of building texture maps under PRMan as people sometimes get tripped up on what some of the txmake options do.

PRMan always wants to deal with texture maps in a power of two. With this command the 1280x700 image will be resized to the nearest power of two, ie. 2048x1024.The '-' after the “up” is very important. By giving the option “up” instead of “up-” to txmake the texture will be encoded with the pixel aspect ratio of original image. Displaying the resultant texture maps created from an “up” and “up-” will visually appear the same. But doing a txinfo between the two will show the one with “up” as having a 'Pixel aspect ratio correction factor of 1.333333, while one created with “up-” will have a Pixel aspect ratio correction factor of 1.000000. So while visually they might appear identical, they will produce very different images when you actually want to use the texture map.

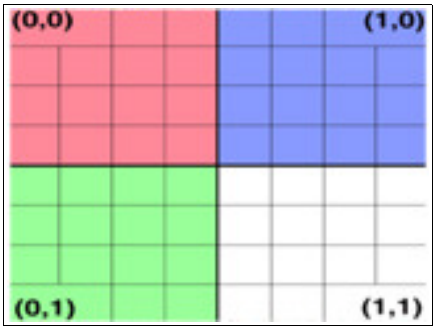


Image 54: Original NDC Render

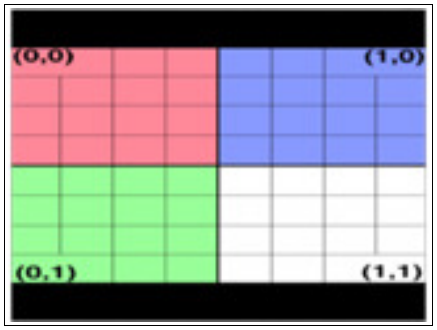


Image 55: NDC projection with txmake -resize up

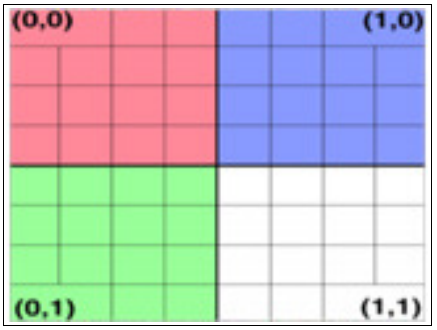


Image 56: Expected NDC using txmake -resize up-

Without the “-” after the resize option, txmake will track the aspect ratio of the texture. The intent is that this would allow for a resized texture or non-square texture to be mapped as a square image when mapped onto a square geometry patch (Image 55). With the “-” option we are saying don't do that. But to instead make the texture coordinates range from 0-1 in the texture map (Image 56).

It is hard to see without loading up the original images and flipping back and forth, but Images 54 and 56 do not line up exactly. Image 56 is slightly smaller. There are some reasons for this. The main thing to keep in mind is that one almost never gets pixel for pixel accuracy when using the PRMan texture map call as it is always trying to filter the map when reading.

Back to our haloing problems...

It seem to us that the culprit to our screen edge hallowing was due to this texturing filtering going on and that this was most likely being compounded by the original image size of the NDC map,1280x700, was being upscaled to 2048x1024. So what we wanted was some way of reading the texture maps with point by point accuracy (ie. point sampling). One way that people typically try to emulate this with the texture call is to use the four coordinates variant and to use the same coordinate for each corner:

```
texture ("mymap.tx", s, t, s, t, s, t, s, t);
```

Or this way for an improved result:

```
texture ("mymap.tx", s, t, s, t, s, t, s, t, "width", 0);
```

This did help some of the screen edge haloing. But it did not handle them all. We still needed a better way of getting point sampling accuracy, especially along the screen edge.

In our quest for point sampling we had tried out the "-resize none " option of txmake:

```
txmake -resize none
```

Looking at this texture map:

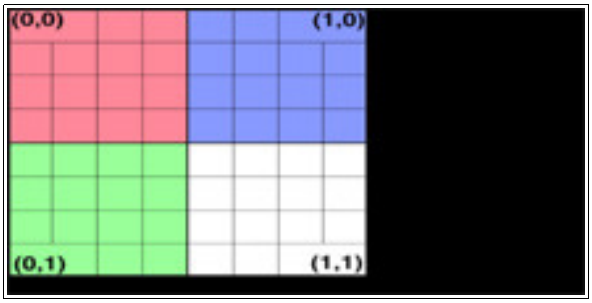


Image 57: txmake -resize none --> 640x480 to 1024x512

Applying this option on a 640x480 image produces a texture map that is 1024x512. Remember PRMan texture maps must be a power of two. But this time instead of upscaling the image features, they are left intact at their original image size, while all of the new image area is filled in with empty data. This looked promising as we could just remap the map access, in the shader, giving us something pretty close to the pixel for pixel accuracy we were looking for.

Unfortunately, this texture didn't render properly. The culprit is an issue with the "`-resize none`" on Pixar TIFF texture maps. Doing a txinfo on this map, would show that the "Pixel aspect ratio correction factor" to be 0.0.This seems to be a bug with txmake when creating TIFF texture maps.

Fortunately, we still have access to the older Pixar texture file format:

```
txmake -format pixar -resize none
```

This will give the proper "Pixel aspect ratio correction factor" of 1.0 and will produce the expected image when rendered. Going with this was what was needed to fix the screen edge haloing issue.

Quick review: To fix the screen edge hallowing issue, we built our NDC maps with:

```
txmake -format pixar -resize none
```

Then in the light shader remap the texture coordinates to access only the original texture area. Here is the light shader with these changes incorporated into it:

```
light l_ambOcclMP (
    float intensity = 1;
    color lightcolor = color 1;

    //Original NDC render size.  ie. source texture
    uniform float occlNDC_width = 1280;
    uniform float occlNDC_height = 700;

    // skip NDC if bypass == 1
    float occlNDC_bypass1 = 1;
    string occlNDC_Map1 = "";
    float occlNDC_MapInvert1 = 0;


       /* Note, that the light has two ambOccl variables.
    1. __amOccl is set by the light and sends this value to the
    surface shader.
    2. __ambOccl__ is set by the surface shader, via it's __ambOccl
    and passed to the light shader.
    */
    output uniform float __ambOccl__ = 0;
    output uniform float __ambOccl = 1;

    string __category = "ambientMP,ambientClass";
)
{
    uniform float foundAmbOccl = 0;    // true if have ambOccl
    float Foccl_1 = 0;                 // ambOccl value

    //NDC space
    point Pndc = transform("NDC", Puse);
    float x = xcomp(Pndc);
    float y = ycomp(Pndc);


    //assumed size of map assuming 1280x700 with -resize none
    uniform float txSize[2] = {2048, 1024};

    //Get the texture map size
    if (textureinfo (NDCmap, "resolution", txSize) == 0) {
    printf ("NDC map: %s texureinfo failed.  Results suspect!!!\n",
          NDCmap);
    }

    / Remap x,y for the original map data ie. the non blank region
    uniform float widthRatio = txSize[0] / occlNDC_width;
    uniform float heightRatio = txSize[1] / occlNDC_height;

    if (heightRatio != 1.0)
    y /= heightRatio;
    if (widthRatio != 1.0)
    x /= widthRatio;
```

```
    //If all conditions are met read NDC map.
    if (OcclNDC_Map1 != "" && OcclNDC_bypass1 == 0 &&
    __ambOccl__ > 0 && __ambOccl > 0) {
        Foccl_1 =
        float texture (OcclNDC_Map1[0], x, y, "blur",
                    occl_blur, "filter", "gaussian");
        Foccl_1 = (OcclNDC_MapInvert1 == 0) ? Foccl_1 : 1-Foccl_1; /* invert */
        foundAmbOccl = 1;
    }
    /* NOTE: May want some code to tint, gamma, or remap the occlusion
       before using it */

    solar () {
    /* Using pointlight mode.  But shouldn't really matter */
    Cl = intensity * lightcolor;

    if (foundAmbOccl == 1)
        Cl *= Foccl_1;
    }
}
```

This next image (Image 58) is represents the second main issue we had with the NDC maps being used in the lights.

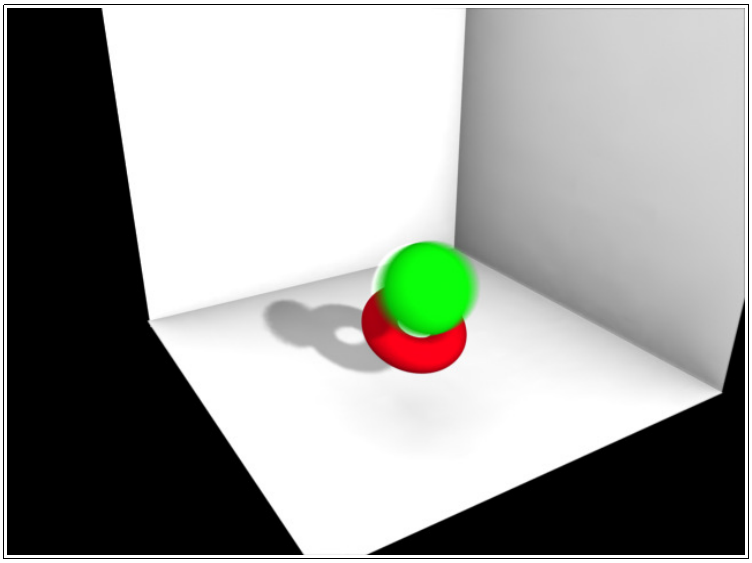


Image 58: inter-object halo issue (left of green sphere)

The inter-object halo, which showed up when the renders were motion blurred, was caused by a moving object revealing an object that was hidden. The NDC map no longer accurately captured the scene as the object was being dragged across during the motion blur. (As mentioned earlier NDC maps were not rendered with motion blur and when this was attempted as a solution. It, at best, only improved the artifact slightly or made it much more pronounced.)

In the end, the solution was to generate Zdepth maps at the time of creating the NDC occlusion maps. Then the light shader would compare the depth maps between the current frame and the motion blur direction frame (ie. shutter open and shutter close) and choose the NDC map based on what this depth test returned.

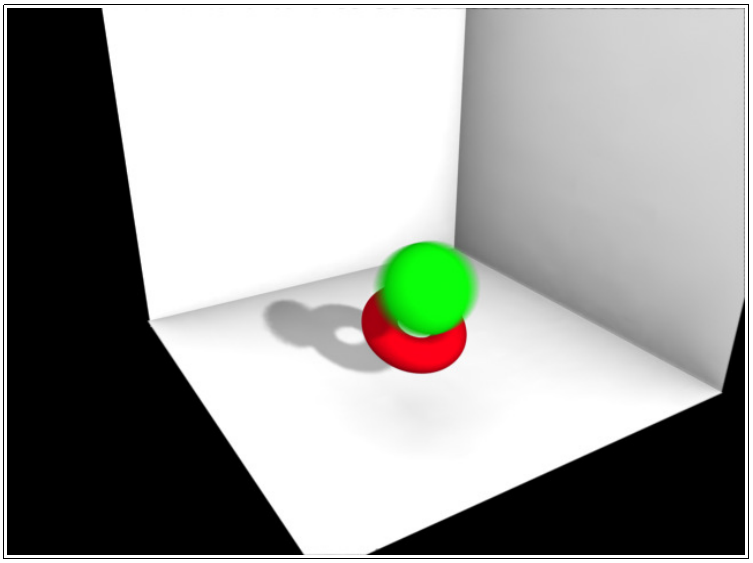


Image 59: Fix for inter-object halo (depth maps)

Building on from the previous incarnation of our example light shader we now add the code to handle the Zdepth information to fix the inter-object halo.

One more thing should be mentioned before going on to the source. In addition to adding the Zdepth map, we have the NDC maps be written as RGBA (4th channel alpha). So when generating the NDC maps we have the backplane rendered with a constant shader which has `Ci = 1` and `Oi = 0` with no premult. The shader reads the alpha channel of the NDC map to undo the blending of the background object onto the edges of the foreground object.

The NDC maps are black where there is no occlusion and white where there is. So if there was any extraneous black in the NDC map, this would show up as brighter areas by the light. Normally the renderer tries to blend the background in with the forground objects as part of its geometric anti-aliasing.

By setting up the backplane and rendering the NDC with an alpha in this way we are minimizing effect. This is the purpose of the alpha code in the light source forthcoming.

Special thanks to Brian Gardner who came up with some of the workarounds for the issues that came up through the many generations of this technique.

Example light shader using zdepth information to fix artifacts.

```
/*
 * Example of ambient light with message passing
 *
 * Tal Lancaster
 * Walt Disney Feature Animation
 *
 */
light l_ambOcclMP (

    float intensity = 1;
    color lightcolor = color 1;
    //Original NDC render size.  ie. source texture
    float occlNDC_width = 1280;
    float occlNDC_height = 700;

    // skip NDC if bypass == 1
    float occlNDC_bypass1 = 1;
    string occlNDC_Map1 = "";
    float occlNDC_MapInvert1 = 0;

    // Current frame number being rendered
    float occlNDC_frameNum = 1;

    // depth map
    string occlNDC_Z_Map = "";
    float occl_blur = 0.01;

    /* -1 - reverse MB; using previous frame as frame close
       0    centered MB
       1    forward MB; next frame is frame close. */
    float occlNDC_motionBlurDirection = -1;


    /* Note, that the light has two ambOccl variables.
    1. __amOccl is set by the light and sends this value to the
    surface shader.
    2. __ambOccl__ is set by the surface shader, via it's __ambOccl
    and passed to the light shader.
    */
    output uniform float __ambOccl__ = 0;
    output uniform float __ambOccl = 1;

    /* an ambient Message Passing light and also a member of the ambientClass
       family */
    string __category = "ambientMP,ambientClass";
)
{
    uniform float foundAmbOccl = 0;    // true if have ambOccl
    float Foccl_1 = 0;                 // ambOccl value

    // NDC space
    point Pndc = transform("NDC", Ps);
    float x = xcomp(Pndc);
```

```
float y = ycomp(Pndc);

//assumed size of map assuming 1280x700 with -resize none
uniform float txSize[2] = {2048, 1024};

//Get the texture map size
if (textureinfo (occlNDC_Map1, "resolution", txSize) == 0) {
      printf ("NDC map: %s texureinfo failed.  Results suspect!!!\n",
      occlNDC_map1);
}

//Get the texture map size
uniform float widthRatio = txSize[0] / occlNDC_width;
uniform float heightRatio = txSize[1] / occlNDC_height;

if (heightRatio != 1.0)
y /= heightRatio;
if (widthRatio != 1.0)
x /= widthRatio;

float Fshutter = 0;

/* Stores the frame open frame number (current frame)
   and the close frame number - the start and end motion blur frames */
uniform float IOpenFrameNum = round(occlNDC_frameNum);
uniform float ICloseFrameNum =  round(occlNDC_frameNum +
                                occlNDC_motionBlurDirection);

//If all conditions are met read NDC map.
if (occlNDC_Map1 != "" && occlNDC_bypass1 == 0 &&
     __ambOccl__ > 0 && __ambOccl > 0) {

    // Only continue if have the zdepth map.
    if (occlNDC_Z_Map != "") {

        // the depth of the current point being shaded
        float Fdepth = length(vtransform("world", vector(Ps - point(0))));

        // the depth value provided by the depth map.
        float Foccl_depth = float texture (occlNDC_Z_Map[0], x, y,
                            "blur", 0, "filter", "gaussian", "lerp", 1);

        /* str_reframeMap is a shadeop DSO that looks for the frame
           number in the map string name and creates a new map name
           with the close frame number in it.

           ie. zdepth.0003.tx -> zdepth.0004.tx (or whatever
           ICloseFrame is set to).

           An alternative to this if one wanted to just stay in the RSL is
           to  be set to something like:
           'zdepth.%04d.tx'.  Then you could use the format call to
          set a map name for OcclZCloseMap
```

```
        OcclZCloseMap = format (occlNDC_Z_Map, ICloseFrameNum);
        (and you would need something like a OcclZOpenMap for
        the open frame, too)
    */
    // The name of the close motion blur depth map.
    uniform string OcclZCloseMap = str_reframeMap (occlNDC_Z_Map,
                                    ICloseFrameNum);

    float FocclClose_depth = float texture (OcclZCloseMap[0], x, y,
                             "blur", 0,"filter", "gaussian", "lerp", 1);
    if (FocclClose_depth <= Foccl_depth)
      // Using the shutter-open NDC values
      Fshutter = 0;
    else {
      // Mixing between the shutter-open and shutter-close
      // NDC maps.
      Fshutter = clamp(smoothstep(0.01, 0.05, smoothstep(Foccl_depth,
                     FocclClose_depth, Fdepth)), 0, 1);
        }

    Foccl_1 = float texture (occlNDC_Map1[0], x, y, "blur",
                             occl_blur, "filter", "gaussian");

  /*Here is the use of the alpha to attempt to undo any blending
    That will occur with the background and foreground's edges. */
float alpha = float texture ((occlNDC_Map1)[3], x, y, "blur", occl_blur,
                             "filter", "gaussian", "lerp", 1, "fill", 0);
if (alpha != 0)
  Foccl_1 = clamp((Foccl_1-(1-alpha))/alpha,0,1);

  Foccl_1 = (occlNDC_MapInvert1 == 0) ? Foccl_1 : 1-Foccl_1; /* invert */
  foundAmbOccl = 1;

/* NOTE: May want some code to tint, gamma, or remap the occlusion
   before using it */

if (OcclZCloseMap != "" && occlNDC_Z_Map != "") {
    FocclClose_1 = float texture ((OcclCloseMap)[0], x, y,
                   "blur", occl_blur,
                   "filter", "gaussian", "lerp", 1);

    float alpha = float texture ((OcclCloseMap)[3], x, y,
                   "blur", occl_blur,
                   "filter", "gaussian", "lerp", 1, "fill", 0);
    if (alpha != 0)
      FocclClose_1 = clamp((FocclClose_1-(1-alpha))/alpha,0,1);

 FocclClose_1 = (OcclNDC_MapInvert1 == 0) ? FocclClose_1 :
                1-FocclClose_1;
}
if (FocclClose_1 < Foccl_1)
    Foccl_1 = mix(Foccl_1, FocclClose_1, Fshutter);
```

```
    }
        }
    solar () {
    /* Using pointlight mode.  But shouldn't really matter */
    Cl = intensity * lightcolor;

    if (foundAmbOccl == 1)
        Cl *= Foccl_1;
    }
}
```

Disney Shader Profiling

Tal Lancaster

(*The following covers work done prior to RPS-13, which now has native profiling capabilities. The source code for the timer can be found in the SIMD DSO section. Brent Burley came up with the concept and coded up the DSO. Also make sure to check out the PRMan-13 shader profiling section.*)

An earlier section provides a little background on on Disney's shader build system. You might want to read that before continuing so it doesn't need to be repeated again here. But here are a couple key bits:

- library of functions/modules amassed over the years
- shader templating – final shader built by picking which pieces are active.

Frequently the shaders produced tend to be large. This is due to the users' appetites to have more features available in the shaders. But at a certain point, they started to complain that the shaders are slow and that the rendering times are becoming prohibitive.

If the slowdown just started soon after you pushed out your changes, then most likely these two events are related. So looking at this most recent code seems like a logical place to start. On the other hand, what if there were some older features that they haven't been made use of until around the time you released the new ones? If you have a library amassed over 10 years, it might not be obvious what the problem might be. Or maybe there are several shader writers putting changes into the system, how do you know what is causing the slowdown? Profiling can be used preemptively to tell how much a desired feature costs in rendering time before it is made public public to the users. This way they are aware of the consequences of utilizing this feature.

Another way to think of the usefulness of shader profiling is to consider how useful profiling is in traditional programming. You have a code base that you would like to speed up. One might be tempted to think that if you go line by line through your program and really tweak it out with every heuristic you can come up with you should be able to make your program as efficient as it possible can. In a perfect world where your code base is pretty small and you are up on how your compiler works, and you have infinite time to burn tweaking it, then yes, most likely you could. However, in practice this isn't really an efficient use of your time. Profiling helps to steer you away from bad assumptions by providing clear indications of where the bottlenecks are in code.

Without profiling you are tempted to make guesses and think “hmm... I bet is section X really wasteful. I bet I can really improve on it”. Let's say you spend a couple of weeks and you improve this section of code so that it is twice as fast. You pat yourself on the back, thinking “job well done”. But was this really a good use of your time if you find that section X accounts for less then 1 percent of the overall rendering time? The user isn't going to notice this 2X speed up one bit.

This is why profiling is important. It takes out any guess work on where you need to spend your time to improve your renders. It also tells you where it's just not worth taking the time. Profiling has helped us to zero into problem areas very quickly. It has also helped to detect a number of Pixar PRMan bugs, too.

For all of these reasons, we have found it extremely useful to add shader profiling into our shaders. This is accomplished via a timer DSO shadeop which is called within all of our major shader modules. It is also used within all of the DSO calls (including itself). This way we know where our rendering times are being spent. If there is a need for even more refined information, we can add more timing information in the area in question.

These timers use a CPU instruction counter for nano-second accuracy with negligible overhead. They are accurate and lightweight. In fact, we build all of our shaders to use them all of the time. But a user will only see the results if they ask for it before starting the render.

In our code, the use of the timers look something like:

```
TIMER_START("timer name");
... shader code
TIMER_STOP("timer name");
```

The output is of the form:

```
timer_name              Wall Time (percentage of time)     Number of times called
```

Here is an example output:

```
% prman -p test.rib

       Real time:     05:18
       User time:     04:34
```

```
s_diffuse_sheen_sss12       260.53 (86.1%)          2107
lm_subSurfaceScatter        200.69 (66.4%)          2097
shadExpr                    46.59 (15.4%)           51832
shadExpr map read           41.76 (13.8%)           71028
cm_layers                   38.70 (12.8%)           3739
lm_mmpSheen                 17.90 ( 5.9%)           2971
s_basic_sheen8              9.69 ( 3.2%)            541
s_basic_sheen12             7.39 ( 2.4%)            333
l_uberList4                 6.85 ( 2.3%)            32532
lm_prmanLambertDiffuse      3.41 ( 1.1%)            3863
```

Note: the percentages do not add up to 100%!! The reason for this is the timer just measures the start and end times for the block of code that that a given timer is run. In the code there are timers within timers. That is to say as we add refinement timers deeper into the code. Therefore, these timer results will have some overlap with their higher level parent functions. For these numbers to make sense, it is helpful to be familiar with the shader's function call structure.

Briefly here is a overview to help the numbers start to make some sense:

- `s_diffuse_sheen_sss12`, `s_basic_sheen8`, `s_basic_sheen12` are surface shaders.
- `l_uberList4` is a light shader.
- `lm_prmanLambertDiffuse`, `lm_mmpSheen`, and `cm_layers` are all modules used by all of the surface shaders, while `lm_subSurfaceScatter` is only called by `s_diffuse_sheen_sss12`.

`shadExpr` is a common function called by the surface shaders and many of the modules. Also, `shadExpr map` is a function called within the `shadExpr` function. Shader Expressions (SE) were first presented by Brent Burley at the Stupid RAT Tricks 2004:

http://www.renderman.org/RMR/Examples/srt2004/index.html#shaderExpressions

SE allow us here at Disney to extend static .slo shaders by entering arbitrary mathematical expressions {and texture loop ups} through string parameters long after the compiled shader was delivered to the end user.)

Oh, and don't forget that surface shaders essentially call light shaders.

So hopefully now you can start to make a little sense of the times and percentages column. In the above example `s_diffuse_sheen_sss12` (`sss12`) took up 86.1% of the wall time for this render. `lm_subSurfaceScatter` which is only used by the `sss12` surface shader accounted for 66.4% of the total render time. However since it is a subset timer of `sss12` (and that no other shaders use it) we know that it

accounts for 77% of sss12 -- 200.69 / 260.53.

`cm_layers` is used by all of the surface shaders, which means that at some point(s) possibly all three of the surface shaders made calls to this module, which in turn activated this module's timer. All that we can safely conclude is that it accounts for 12.8% of the total render time.

Note the `shadExpr` and "`shadExpr map read`" lines. As mentioned, pretty much every shader and module calls the `shadExpr` function (it is called 51832 times in this render). Someone might conclude that `shadExpr` is accounting for 15.4 % of the total render time. However, this is misleading if you don't account for any texture accesses that were triggered through the evaluation of the shader expression. Which is why the "`shadExpr map read`" line is there. 13.8% of the total render time was spent doing texture lookups, meaning this section is I/O bound. As it is a subset of `shadExpr`, we can conclude that the non-texture reading portion of `shadExpr` only accounts for 1.6% of the total render time (15.4%-13.8%). We separate the I/O part for evaluating the Shader Expression costs as this I/O would still be there in a non-Shader Expression shader if the shader writer anticipated the need for this texture lookup and performed it traditionally.

The third column of the timers represents the number of times this timer was called. These counters are called once per shading grid. So they help to see how much griding one is getting. If you know a shader is only called so many times, but a particular function call (for which no other shader uses) is being called many more times. It might be worth looking into (if that call is contributing to a decent percentage of the bottleneck. The counters are also helpful it alert you have lights that are refreshing the cache. It might be worth seeing if this is absolutely necessary all of the time.

Here are the output from two different runs on essentially the same RIB file. But pointing to different shader sets.

```
% prman -p test.rib

      Real time:     05:18
      User time:     04:34

  s_diffuse_sheen_sss12                          260.53 (86.1%)         2107
  lm_subSurfaceScatter        200.69 (66.4%)     2097
  shadExpr                    46.59 (15.4%)      51832
  shadExpr map read           41.76 (13.8%)      71028
  cm_layers                   38.70 (12.8%)      3739
  lm_mmpSheen                 17.90 ( 5.9%)      2971
  s_basic_sheen8              9.69 ( 3.2%)       541
  s_basic_sheen12             7.39 ( 2.4%)       333
  l_uberList4                 6.85 ( 2.3%)       32532
  lm_prmanLambertDiffuse      3.41 ( 1.1%)       3863
```

Now compare these number with the numbers from the same RIB file rendered a few months later.

```
% prman -p test.rib

       Real time:      12:47
       User time:      11:54

  s_diffuse_sheen_sss12        702.92 (94.2%)          2107
  lm_subSurfaceScatter         641.49 (86.0%)          2097
  shadExpr                     47.72 ( 6.4%)           51832
  shadExpr map read            41.90 ( 5.6%)           71028
  cm_layers                    39.27 ( 5.3%)           3739
  lm_mmpSheen                  17.82 ( 2.4%)           2971
  s_basic_sheen8               9.17 ( 1.2%)            541
  s_basic_sheen12              7.38 ( 1.0%)            333
  l_uberList4                  6.87 ( 0.9%)            32532
```

Some users saw a 2X slowdown in their renders when they started using a newer shader base. Looking at the profiles we see the top two sections are `s_diffuse_sheen_sss12` (94.2%) and `lm_subSurfaceScatter` (86.0%)! As sss12 calls the subsurface scattering module, this seemed like the most likely area to check out to see what happened to cause this massive slowdown. Without this information, I would have had to go through all kinds of modules to track each of the hundreds of changes that every shader writer made over those past months and verify that they are ok. Before the profiling data, this is something I have had to do on more occasions than I care to think about.

Here is a code snippet of how a section of the subsurface code looked a few months prior:

```
gather (   "illuminance", Pl, Vl, 0, 1,
           "ray:length", r_depth,
           "attribute:user:sssdensity", sssdensity,
           "attribute:user:ssscolorr",ssscolorr,
           "attribute:user:ssscolorg",ssscolorg,
           "attribute:user:ssscolorb",ssscolorb,
           "subset", traceSet) {

    if ( r_depth < Dl + 2 * depth ) {
           float w,w2;

           // flux contribution with respect to light direction.
           // front scatter preferred to backscatter.

           if(mode == 1)
           {
                  w = (Ln.dVn)+1;
```

```
                w2 = 1;
        }
        else
        {
                w = 1;
                w2 = 1;
        }

    . . .
}
```

Then here was how it was looking when the users were seeing the slow down (new code in bold):

```
gather (   "illuminance", Pl, Vl, 0, 1,
           "ray:length", r_depth,
           "primitive:N",norm,
           "attribute:user:sssdensity", sssdensity,
           "attribute:user:ssscolorr",ssscolorr,
           "attribute:user:ssscolorg",ssscolorg,
           "attribute:user:ssscolorb",ssscolorb,
           "subset", traceSet) {

        if ( r_depth < Dl + 2 * depth ) {
           float w,w2;

           norm = normalize(norm);

           // flux contribution with respect to light direction.
           // front scatter preferred to backscatter.
           if(mode == 3)
           {
               w = (Ln.dVn)+1;
               w2 = clamp(-norm.Vl,0,1);
           }
           else if(mode == 2)
           {
               w2 = clamp(-norm.Vl,0,1);
               w = 1;
           }
           else if(mode == 1)
           {
               w = (Ln.dVn)+1;
               w2 = 1;
           }
           else
           {
               w = 1;
               w2 = 1;
           }
```

```
        . . .
    }
```

Looking at these two code sets there really wasn't anything obvious at first glance as to what would be causing a 2X slow down. As there were several months of code changes over many of the modules and there wasn't anything obvious here, most likely I would have moved on looking at other the other changes, hoping to find some obvious connection.

But with the profile data, I knew it had to be this module. Something here was not right. The culprit turned out to be a Pixar PRMan bug in the gather call. In theory a "primitive" query call is not suppose to trigger shader execution, yet it was. (This bug is fixed is PRMan-13 and so it is no longer an issue.)

The workaround was to redo the gather call so that if the user didn't want the two newer modes, the gather isn't called with the "primitive:N" query, but gather would be called with this query if the user did ask for the mode options that needed it. With this the render times we back to normal when they selected either of the two original modes – which was most of the time. Also, we were able to let the users be aware of the cost if they decide to use either of the other two choices.

Be sure to check out the PRMan-13 section for information on the renderer's built-in in profiling feature. So now that PRMan-13 provides profiling does this method, presented here, still have any use for PRMan users? For us, for the immediate future anyways, we intend to use both.

The Pixar one gives a nice stack state of your calls and allows you to jump to your the locations of your source code where the potential bottlenecks are. But the Pixar method has the potential of increasing the cost rendering with it on (both in memory and CPU time). So you might not want to have this on for every render.

The method presented here is lightweight enough (assuming you don't go overboard), that we have them on all of the time. Also with them, one can control the level of granularity that is being tracked. Some might consider this a good thing or others that it is a bad thing, in that one must sprinkle the timers through out the code, while with the Pixar method its automatic as long as the shaders are compiled properly.

The source code for the timer DSO can be found in the SIMD DSO section. Beware that the code is setup to only work with Linux.

Color Opacity -- what's it good for OR who really uses this?

Tal Lancaster

Before RenderMan renderers could do ray tracing people have struggled to mimic glass-like objects from the typical lighting models. Now with access to ray tracing (RT), we can get better looking glassy objects. However, with ray tracing new issues arise. One of these issues is what if the glass object is to have varied opacity across the surface (compare the spheres in Images 60 and 61, the centers are fairly transparent in the centers but go more opaque along towards the edges)? Another issue is that one tends to prototype the look with out ray tracing (for faster iterations), but as soon as you turn on RT the object doesn't behave like it did before RT, especially if you were depending on some use of diffuse to create the look.

Part of these issues are due to PRMan's auto ray continuation. With RT off one typically would use a small amount of opacity. But as soon as you wanted to have RT refractions you had to set the opacity to one. Changing from almost no opacity to fully opaque will throw off your traditional lighting models like diffuse.

To get the nice look of ray-traced refraction, in the end it isn't uncommon for people to just not worry about varying opacity of the semi-transparent surface, letting all of their glass objects appear to have the same opacity.

When PRMan first introduced ray tracing (PRMan-11), it didn't support semi-transparent objects very well. With it, if a ray hit a non-opaque object, the shader writer would have to send off other rays from the hit surface. Also if the shader writer was really gung-ho they go could try to track the opacity levels for each hit. Eventually PRMan (12) offered auto-ray continuation, which made the shader writer's life much easier.

In the “Ray-Traced Shading in PRMan” Application Note, there are these two sentences:

“If an effect such as refraction is desired then the surface shader on the refracting object should set `Oi=1.0;` (meaning opaque, no further automatic continuation) and cast a new refraction ray itself in the desired direction. The background colors found by the refraction rays are then used to set the output color, *Ci,* of the refracting surface.”

For a long time, I took this literally. So when ray-traced refractions were needed, I would resign the shader to immediately turning off any transparency that was going on in an object and just bring in the background (refracted) colors to the front of the surface.

These images,60-62, illustrate some of the common glass like materials that people use. The images contain two spheres. The left sphere is always the same material. In images, where there is ray traced (RT) refractions, the right sphere is the one having this applied to it. While the left sphere is still the original material. In images without RT refraction, both spheres have the same material.

Image 60 is a typical result without the use of ray tracing. It is done with combinations of opacity, specular, and a little bit of diffuse.

In image 61, the right sphere, now has ray tracing. Note the diffuse had to be turned off. Also any sense of varying opacity is gone due to setting the opacity equal to 1. One is left with the use of fresnel to have any sense of thickness of the glass. (One might try having a varying fresnel value to have a sense of varying thickness.) Another way to may get a sense of thickness is one could have a surface tint color to tint the refraction by. Changing the saturation and luminance of the color could help give a different sense of opacity of the surface.

Image 62, shows why we have to turn off diffuse once RT refraction is being used.

Image 63, shows the much more desirable result where both left and right spheres behave very similarly. The left sphere is without RT refraction. While the right one does. The reason for this behavior is due to through the use of color opacity.

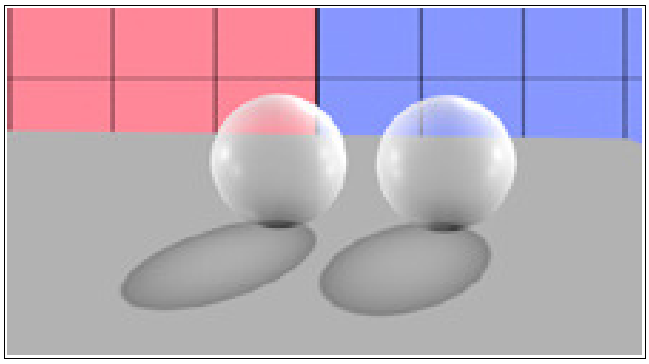


Image 60: traditional non-RT semi transparent glass (mainly transparency, specular, and diffuse)

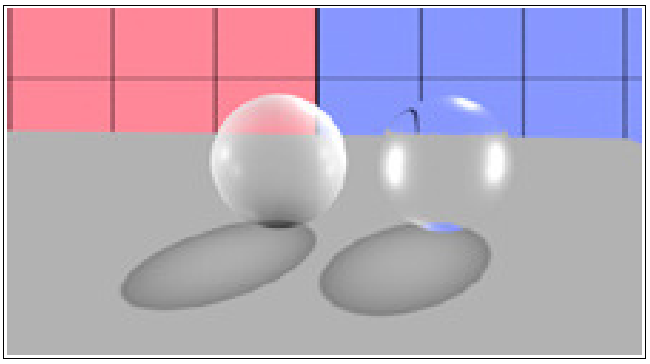


Image 61: Right sphere now ray-traced refraction. Had to turn off diffuse and varying opacity

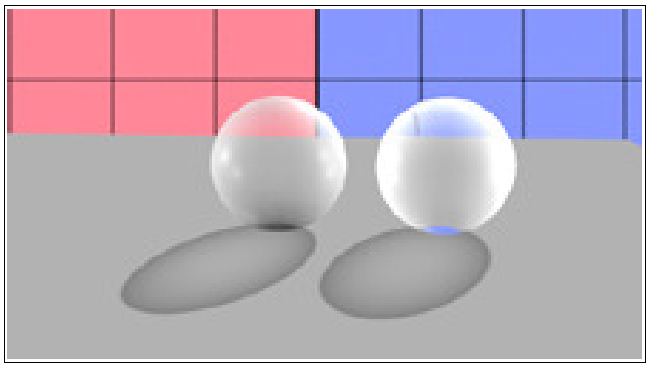


Image 62: Right sphere has RT refraction and diffuse on (causing blowout)

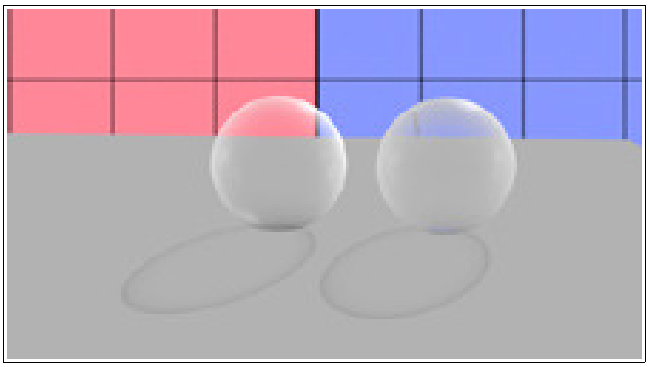


Image 63: Much better effect and balance through colored opacity

Images 64-67 are very much like images 60-63. But instead of white glass material, we have green glass that is fairly transparent in the center but gets pretty opaque toward the edges.

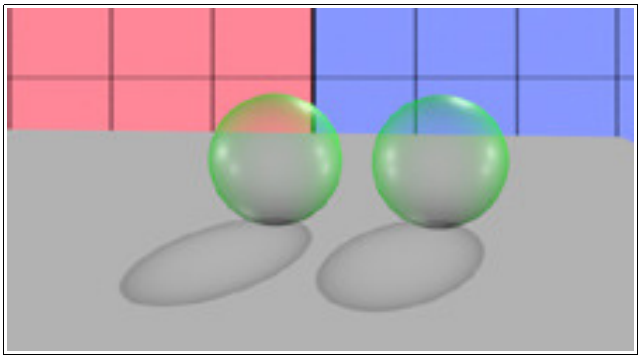


Image 64: semi-transparent green: varying transparency, diffuse, and specular

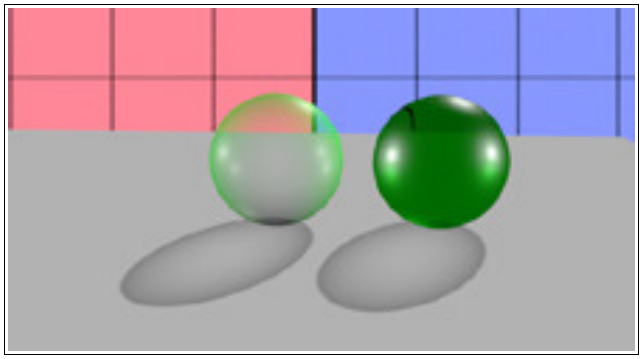


Image 65: Right sphere has RT refraction, but diffuse is now off

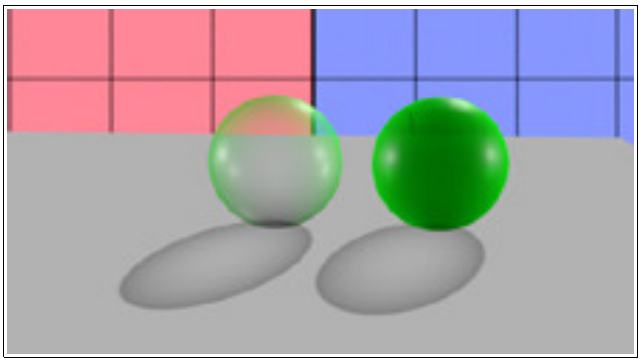


Image 66: Right sphere has RT refraction and diffuse (and specular)

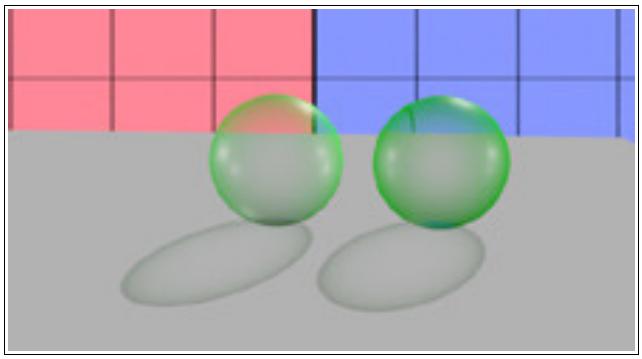


Image 67: Much better effect and more balance, through colored opacity

To sum up images 60-67 attempt to illustrate issues that come up when setting up a look without the use of ray-traced (RT) refractions and then turning on RT refraction. Any sense of semi-transparency is lost (images 61, 62, 65, 66). Using diffuse illuminance along with refraction also has its problems (images 13 and 17). The end result is, under common shader implementations, the end user must try to go through an effort to balance their material when RT is on and when RT is off.

Images 63 and 67 show a more pleasing solution. There is a much closer balance between the RT (right sphere) and the non-RT (left) sphere. There are two more points worth mentioning (however, it is a little hard to pick up the subtly in this lo-res images):

1. The spheres in these two images feel much more glass like than with traditional approaches.
2. The shadows automatically take on the color and opacity of the casting surface. So where the image is more opaque the shadow is darker. In image 18 those shadows are really green. (Of course these

shadows are deep shadows; non-deep shadows couldn't do this.

So what did we have to do to get Images 63 and 67 working the way they did?

An issue that came up on the production was the need for color shadows. We are not talking about just scaling the shadow map by a solid color (ie. tinting) but the need to have an arbitrary color surface and have the shadow take on those characteristics. Take for example, a stained glass window:

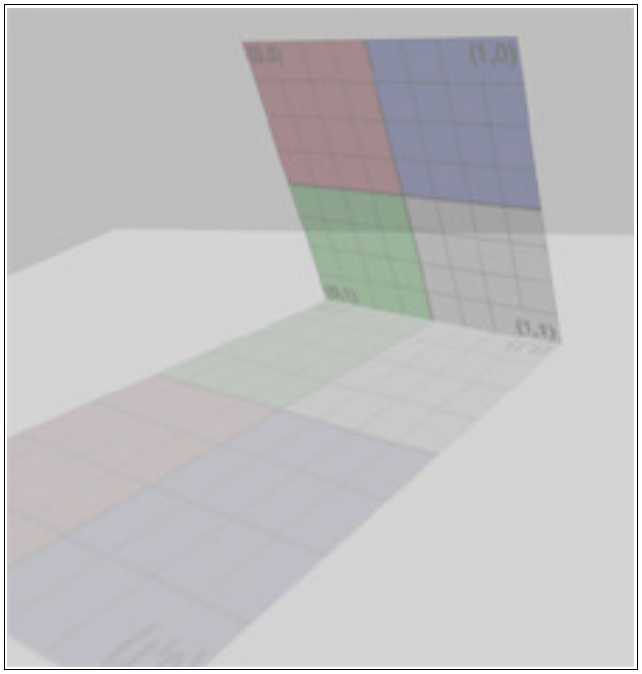


Image 68: color plane with 60% opacity; casting matching color shadow

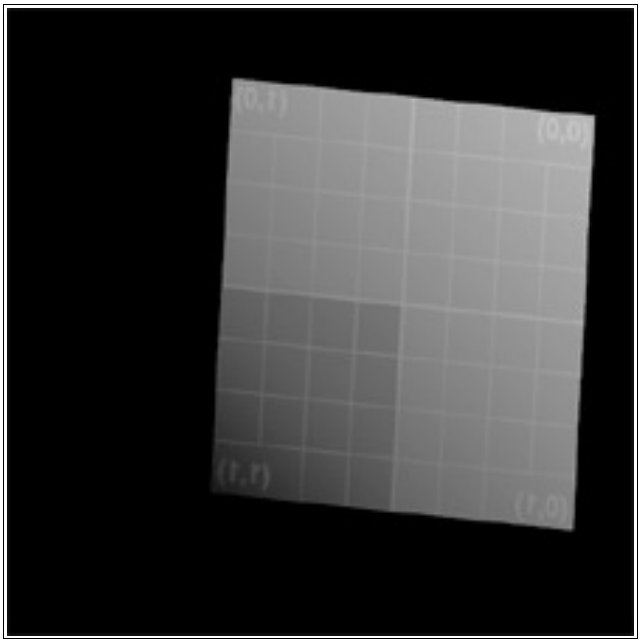


Image 69: Color deep shadow map. No really. Just can't display the color part.

So obviously we will need to somehow get the color into the shadow maps. Which according to the PRMan docs the deeps shadows are suppose to do just fine. However, don't waste your time trying to get sho or IT to display the color shadow maps for you. They only give shadow information as floats. (Image 69)

Speaking of shadows as floats, every light shader I had ever written assumed that the shadow map was just a pure depth value, ie. float. Even when deep shadows were introduced (which could have color data in them), still our shaders only assumed a float value. But now there was a need to think of shadows as color values that would have the source object's color value in it.

Ever since the RenderMan spec came out, `Ci` and `Oi` were defined as colors. I never really gave it much

thought. In every example I knew of `Oi` was essentially a gray scale image. So I just got lazy and thought of it as a float. Just like the usage of the shadow maps, in every shader I had written the transparency section was just a set of float values. There wasn't any way a user could give a color opacity map.

Deep Shadows came along and with this the shadow call got updated so it could return colors and not just pure float values. They track the opacity of a sample as a color value. Deep Shadows can give color shadows by saying how transparent this sample is in terms of RGB.

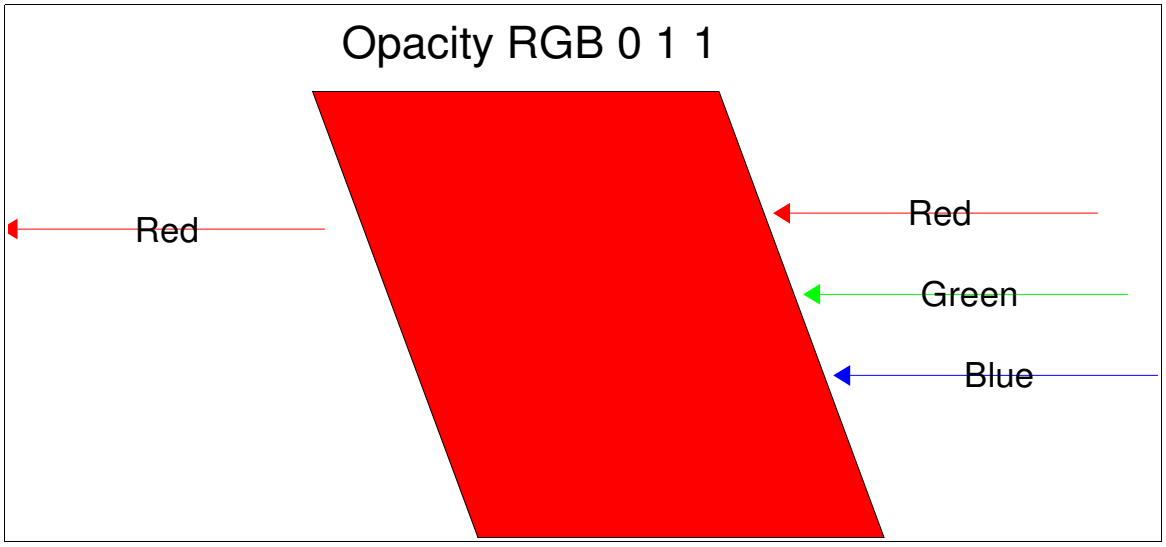


Image 70: Red filter absorbs all but red. Meaning red channel transparent and green and blue are opaque

To get the proper Deep Shadow color, think about opacity as light absorption. Take a red filter as in image 70.

What makes it a red filter is that the red channel is fully transparent in the red, but the green and blue channels are opaque. This is saying to absorb all of the light but red. The result is a red shadow. If instead you were thinking in terms of making the opacity red (1, 0, 0) (like a normal color value), you would be saying to the deep shadow to absorb all of the red light and let through the rest of the light color resulting in a cyan shadow. So to get the proper shadow color, the opacity should be the inverse of the intended color.

To get color deep map shadows, this would mean that some kind of the color section of the shaders would need to be run during the shadow generation call. This assumes that the color shadow wanted is based off the color (or rather the inverse color) portion of the surface shader. To have this affect the surface's opacity when the Deep Shadow map is being built.

So if one needs to produce color opacity for color Deep Shadows, why not have this feature available for the color rendering pass, too?

With some extra effort, I was able to expand on the color opacity to not be just used when making deep shadow color maps, but also during the normal surface color calculations, too. For one this makes for consistency between the shadow and color pass. This also turned out to be the missing link needed to make glass look more natural and for RT refractions to have a better sense of semi-transparent objects and to have the diffuse module be have correctly when RT refractions were being used.

Here is a typical plastic shader:

```
surface plastic( float Ks=.5, Kd=.5, Ka=1, roughness=.1; color specularcolor=1 )
{
    normal Nf;
    vector V;

    Nf = faceforward( normalize(N), I );
    V = -normalize(I);

    Oi = Os;
    Ci = Os * ( Cs * (Ka*ambient() + Kd*diffuse(Nf)) +
                specularcolor * Ks * specular(Nf,V,roughness) );
}
```

Here is what a color opacity plastic shader looks like:

```
surface plastic( float Ks=.5, Kd=.5, Ka=1, roughness=.1;
           color specularcolor=1;
```

```
            // Toggle to enable/disable having the color module affect opacity
            float colorInOpacity = 0;

            // Toggles to increase opacity based on lighting result
            float increaseAmbintOpacity = 0;
            float increaseDiffuseOpacity = 0;
            float increaseSpecularOpacity = 0;

 )
{
    normal Nf;
    vector V;

    Nf = faceforward( normalize(N), I );
    V = -normalize(I);

    Oi = Os;

    Ci = 0;

    // Run color module...Or just to keep example simple:
    color Ct = Cs;

    /* If flag is true, go a head and put the inverse of the colorInOpacity
       into the opacity. */
    if (useColorInOpacity == 1) {
    color Copacity = Ct;
    Oi *= mix (1 - Copacity, color 1, Oi);
    }

    color Cambient = Ka * ambient();

    // increase opacity based on lighting result
    if (increaseAmbientOpacity == 1)
        Oi = max (Oi, color average_rgb (Cambient);
    Ci += Ct * Cambeint;

    color Cdiffuse = Kd * diffuse(Nf);
    if (increaseDiffuseOpacity == 1)
        Oi = max (Oi, color average_rgb (Cdiffuse);
    Ci += Ct * Cdiffuse;

    color Cspec = specularcolor Ks * specular(Nf,V,roughness);
    if (increaseSpecularOpacity == 1)
        Oi = max (Oi, color average_rgb (Cspec);
    Ci += Cspec;

    Ci *= Os;
}
```

The key section of the code is this piece:

```
    if (useColorInOpacity == 1) {
     color Copacity = Ct;
     Oi *= mix (1 - Copacity, color 1, Oi);
     }
```

The opacity is being scaled by the inverse of the base color. It happens as a mix towards white the more opaque the object is. Without this mixture a color object could never be fully opaque, which would make using the shader a bit problematic. Take for example, if instead of the mix line we had:

```
    Oi *= (1-Copacity);
```

and `Oi` was color 1 (fully opaque) and the base surface color was (1, 0, 0), we would end up with an `Oi = color (0, 1, 1)`. Which means when the user wanted a fully opaque object they would be getting a red shadow, because we are saying that the object is transparent in the red channel. So having the mix gives a more expected behavior.

Something else to keep in mind: if your surface shaders typically only compute the opacity during shadow computation (ie. skip the color work), you will need to make sure they now fully compute the base color as this is needed to give a proper color shadow map.

It is probably worth mentioning in case you didn't catch this in the source example. This is the fact that after the base color section the opacity is scaled by the inverse of the color. Then at the end, post lighting, `Ci` is then scaled by `Oi` to perform the pre-mult. So the color is sort of being scaled by itself. It hasn't caused any problems that we have noticed. Still it does seems like a bad thing to be doing.

Here are some code snippets to get your light shader to work with color Deep Shadows.

```
light l_mylight (
   ...
   float useDeepShadows = 0;
)
{
    float isDeepShadow = 0;
    float unoccluded = 1;
    float shadval = 0;
    color Cshadval = 1;

    Cshadval = shadow (shadowname, Ps, "samples", samps, "bias" shadowbias);
```

```
    string stype;
    textureinfo (shdname, "type", stype);

    if (stype == "deep shadow") {
           isDeepShadow = 1;
           shadval = average_rgb (Cshadval);
           Cshadval = 1 - Cshadval;
    }
    else {
           shadval = comp (Cshadval, 0);
           Cshadval = shadval;
    }

    . . .

}
```

The points here are that we still track a float value of the shadow, ie. the percentage of occlusion of the shaded point. It seems to make sense to leave the occlusion as a float. At some point Cshadval is scaled with `Cl` to get the final light color (color shadow times light color).

Now to the issue of getting the RT refraction to behave closer to the non refracting renders. This includes both having the brightness and a sense of varying opacity if it existed.

Remember the quote earlier where refraction needs to have the surface opacity set to 1 for the automatic ray continuation to function? It doesn't say when in the shader the opacity needed to be set to 1. In the past, I always just set it near the beginning of the shader where transparency would normally be figured out. Now I let the lighting models do all of their color pre-multing with what would normally be the surface shaders opacity (without RT). Then at the very end of the shader after any pre-multing has occurred, I set `Oi = 1`.

Unfortunately, there is a little more to it than just that. There is a need to track our old friend the non-color opacity which is used by the RT refraction code to perform its own alpha compositing. Recall the following snippet:

```
    if (useColorInOpacity == 1) {
     color Copacity = Ct;
     Oi *= mix (1 - Copacity, color 1, Oi);
     }
```

We now add the statement `OnotColor = Oi` which allows the refraction code to track the original

opacity,

```
    if (useColorInOpacity == 1) {
     color Copacity = Ct;
     OnotColor = Oi;
     Oi *= mix (1 - Copacity, color 1, Oi);
     }
```

The `OnotColor` is used to know how much of the refracted light color needs to be mixed in with the surface color. It is also used as the control value in mixing the `darkenWithOpacity` effect.

Keep in mind that anywhere in the shader you are altering `Oi`, you will most likely want to perform a similar operation to the `OnotColor` variable. Such as in any `increaseOpacity` sections of the shader. Otherwise there will be a mismatch between the two opacity values.

At the very bottom of the shader after we are done with setting `Oi`, `Ci`, and any pre-multing of anything (including AOVs), with the exception of RT refraction (and refraction AOVs), we add something like the following:

```
if (raytraceOn == 1&& rt_refractBypass == 0) {
    if (useColorInOpacity == 1) {

    color Ctemp = (remoteCiOi == color -1)? Crt_refract:
        remoteCiOi;

    color traditionalOi = Oi/(1-Ct);
    color workOi = clamp (Oi+remoteOi, color 0, color 1);

      /* refraction amount is mixed in with the surface base colorInOpacity
         and the refraction value.  Based on the non-color version
         for the preceeding and current surface: OnotColor */
    Crt_refract =mix ( Crt_refract,
                   Ct, OnotColor);

      /* Tends to give pleasing result by darkening the refractions
         based on the opacity */
    if (rt_refract_darkenWithOpacity == 1) {
        Crt_refract = mix (Crt_refract, color 0, OnotColor);
    }

    }
    else {
    Crt_refract *= Ct;
```

```
    }
}
Ci += rt_Krefract * rt_fKt_refr * Crt_refract;

if (raytraceOn == 1 && rt_refractBypass == 0) {
    Oi = 1;
}
```

Remember this is the end of the shader. Before this refraction code is called `Ci` has already been premulted with `Oi`. Also, Ci doesn't contain the refraction values until this code is run.

In the example code above there is the control flag: `rt_refract_darkenWithOpacity`. When this is on the refraction color will tend towards black the more opaque the object becomes. This tends to make the refraction effect look more natural.

In the above code snippet there is a variable, `remoteCiOi`. The shader is keeping track of the traditional premulted value of the surface color (`RT_Refract_Ct`). The gather call that is performing the RT refraction call queries this value from the incoming surface and stores it in the variable `remoteCiOi`.

In the shader parameter list we have the following two output variables:

```
output varying color RT_Refract_Ct = color -1;
output varying color RT_Opacity = color 0;
```

Also somewhere near the top of the shader we declare the following global variables:

```
    color Crt_refract = 0;
    color remoteCiOi = -1; /* from RT refraction call */
    color remoteOi = 0; /* from RT refraction call */
```

Then the code snippet for the refraction code:

```
{
    {
    gather ("illuminance", P, dir, blur * radians(5), samples,
           "label", label,
           "maxdist", maxDist,
           "surface:Ci", icl,
```

```
            "surface:Oi", iol,
            "surface:RT_Refract_Ct", iCiOi,
            "surface:RT_Opacity", iOi
        ) {
        if (envMap != "" && iol == color 0)
            C += Cenv;
        else
            C += icl;

        remoteOi += iOi;

        if (iCiOi != color -1)
            remoteCiOi += iCiOi;
        iCiOi = -1;

    }
    else {
        ... //gather didn't run

    }
    }


    C /= samples;
    remoteOi /= samples;
    refract_Oi = remoteOi;

    if (remoteCiOi != color 0) {
        remoteCiOi /= samples;
        refract_CiOi = remoteCiOi;
    }

    Crt_reflect = C;
. . .
```

Closing with some examples of a fish tank.

Image 71 is a traditional material without the use of ray tracing.

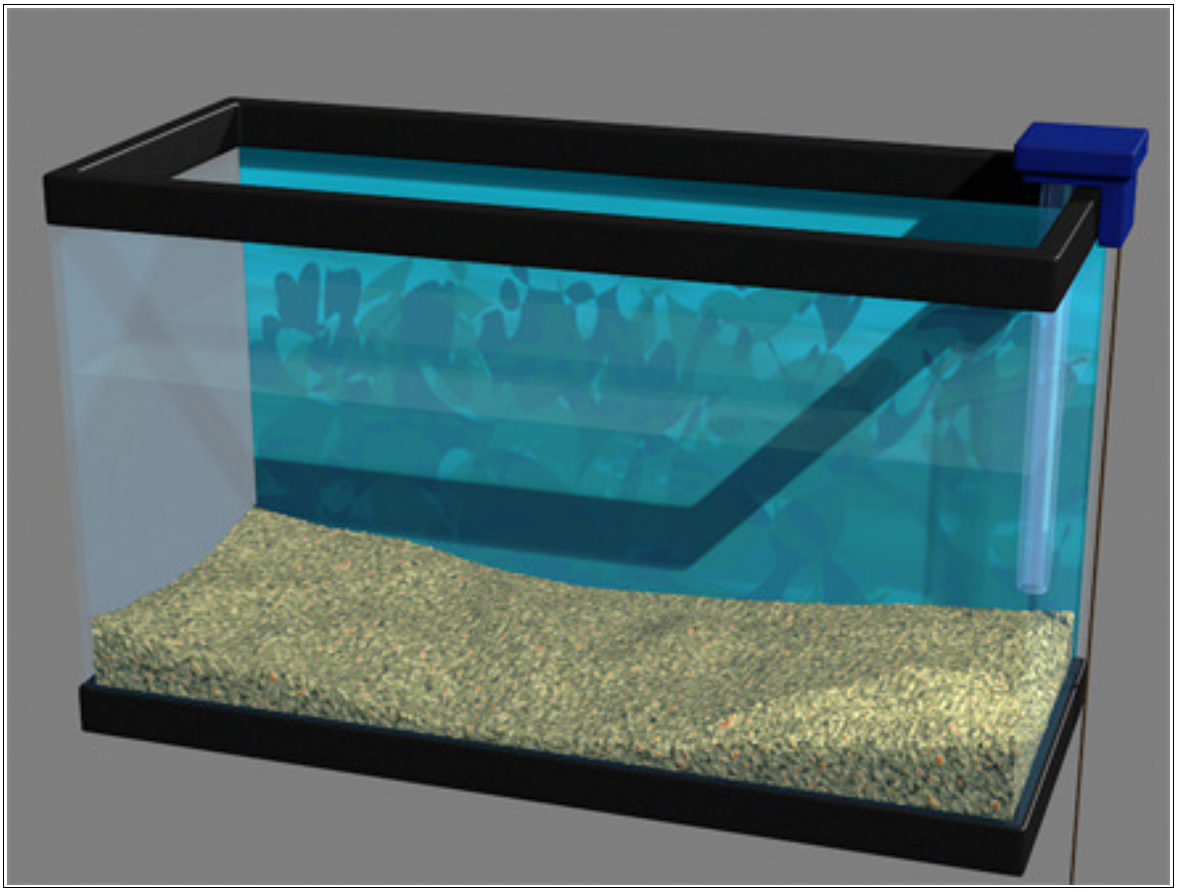


Image 71: traditional glass material (with out ray tracing) © Disney

Image 72 is using color opacity along with ray tracing refraction and reflections. Notice how glowey the corners are.

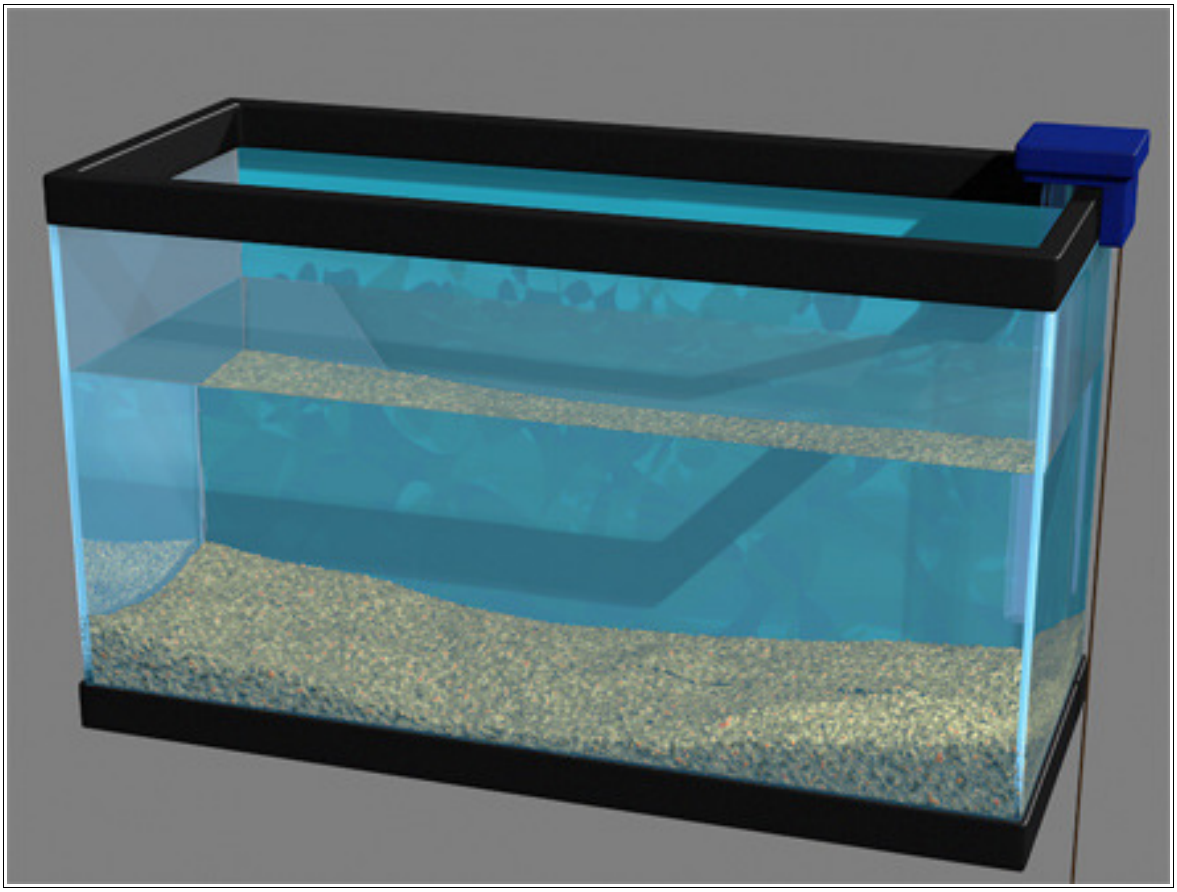


Image 72: glass with ray tracing (reflection + refraction) and color opacity © Disney

Image 73 added a ground plane; bumped up the opacity a bit. It has been rendered with color opacity on so now the shadows are colored.

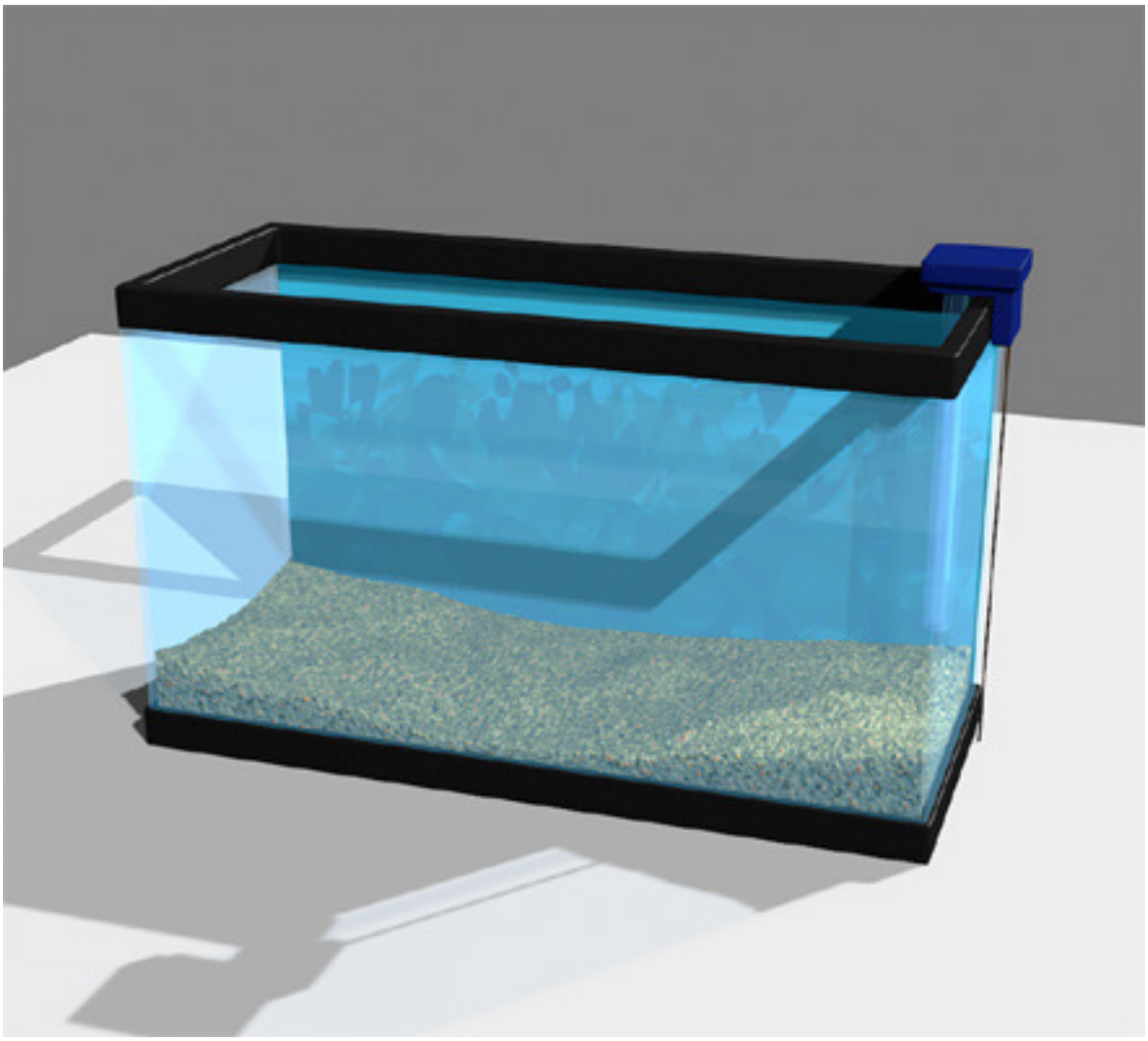


Image 73: Glass with color opacity

Image 74 also has color opacity, but this time with RT reflections and refractions. Notice how much the opacity has change. It now has the sense of being filled with water.

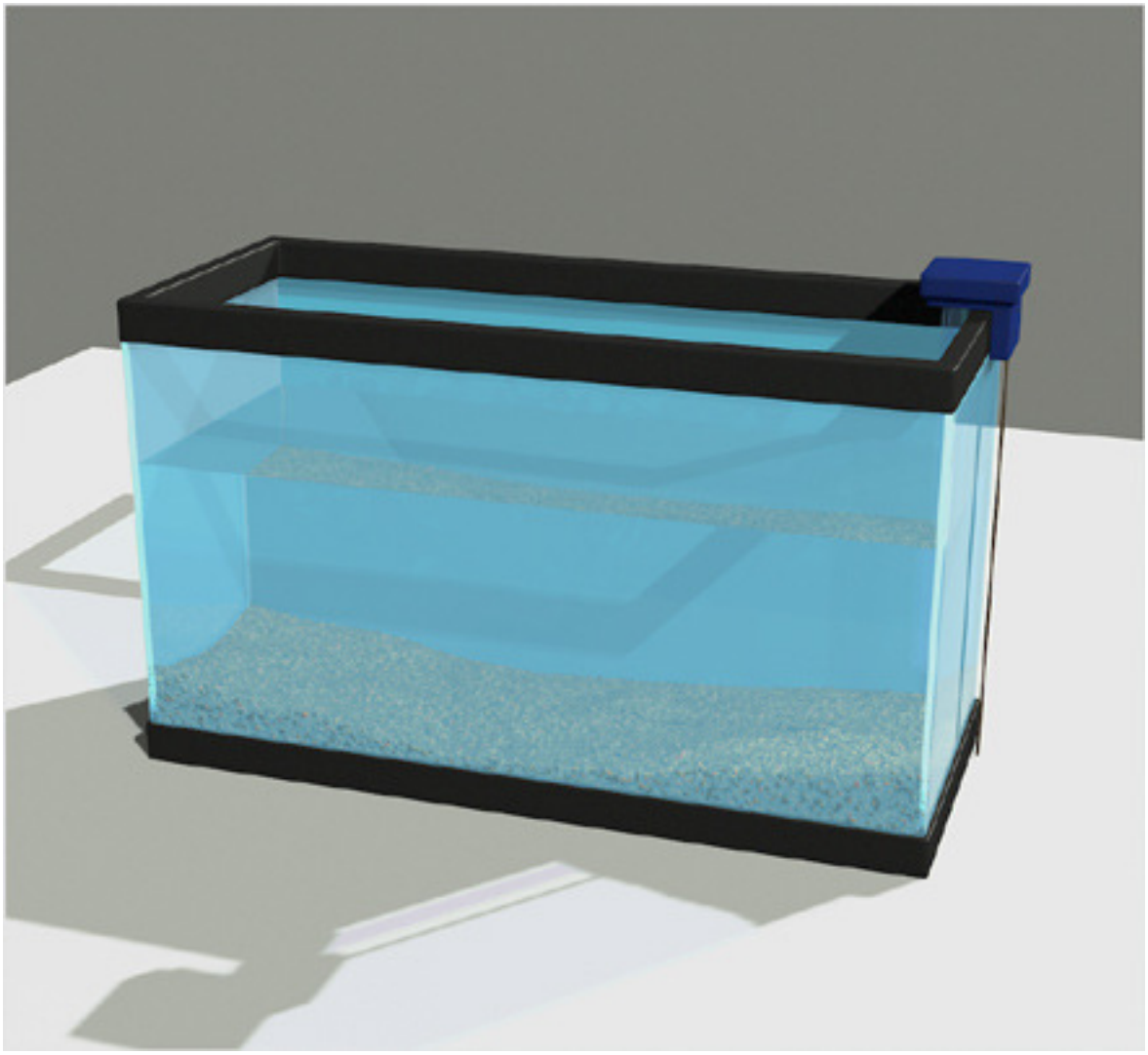


Image 74: color opacity and RT refractions © Disney

Pixar's PhotoRealistic RenderMan version 13

Tal Lancaster - Walti Disney Feature Animation

This section covers some enhancements to Pixar's PhotoRealistic RenderMan (PRMan). Specifically, changes that version PRMan-13 brings. Running through the release notes there are many areas that could be touched on. I will focus on bringing three of these to your attention.

1. A Sample of RenderMan Shading Language (RSL) changes.
2. SIMD shadeop DSOs and implications of a multi-threaded renderer.
3. XML Profiling.

RSL enhancements

For about 10 years or so one had to do something like the following to access the components of the tuple and matrix types in RSL:

```
//For points, vectors, normals
point p = point (1, 2, 3);
```

```
float x, y, z;

//getting the components
x = xcomp(p);
y = ycomp(p);
z = zcomp(p);

//assigning into the components
point p2 = p;
setzcomp(p2, 3.5);

//colors
color c = (0, .5, 1);
float green = comp(c, 1);

//matrix
//stetting a component of a matrix
matrix m;
setcomp (m, i, j, val);
```

Now with RPS-13 components accessed as arrays:

```
point p = point (1, 2, 3);
x = p[0];     // instead of  xcomp(p)

color c = (0, .5, 1);
float red = c[0];  // instead of comp(c, 0)
c[2] = 0;             // instead of setcomp(c, 2, 0)

matrix m;
float elem = m[0, 3];   // instead of comp(m, 0, 3)
```

Transform generalized

When the RenderMan specification was initially released the RSL only had only two tuple types, color and point. This was still this case until the mid-90s until the release of PRMan-3.5 (or maybe it was PRMan-3.6).

So prior this change in the RSL to get true normal and vector types, things like normal N was referred in the RSL as point N; and the vector I as point I. Feel free to dust off a copy of *The RenderMan Companion* to check out examples of this in the shader code. Back then one had to be very careful when using the RSL transform()

call on points that were really normals, and vectors. This is because these objects behave differently than true points.

Case in point, visualize a sphere and the normals for it. Now think about what happens when it gets scaled. The transform needed to keep the normals, normals to the scaled surface is different than that for moving the scaled points themselves.

So in these very early versions of the RSL, to do a proper space transformation for say a vector, one had to think about the vector as two points and transform these two points. A shortcut was to use the origin as one of the points giving:

```
VnewSpace = transform (newSpace, v) - transform(space, point (0, 0, 0));
```

or if newSpace is one of the built in spaces like “object”

```
VnewSpace = transform (“object”, v) + transform(space, point “object” (0, 0, 0));
```

Then in 1995-1996, Pixar (with PRMan-3.5 or PRMan-3.6) introduced vector and normal as valid types RSL types. With these types came the the ntransform (normal transform), vtransform (vector transform). To perform the correct transformations for these non-point types with different behaviors. Then once the matrix type was introduced, there came the mtransform (1997 PRMan-3.7) for dealing with matrix transformations.

With these items as true types we could transform a vector to a different space with something like:

```
vector VnewSpace = vtransform (newSpace, v);
```

Now with RPS-13, the transform has been made polymorphic. The call looks to see what the type is in is parameter and then applies the proper transform for that type. Which means you don't have to specify the type prefix to transform. Instead

```
vector v, VnewSpace;
VnewSpace = transform (newSpace, v);
```

SIMD RSL plugins

As feature rich as the RSL is, there is always somebody wanting/needing to do some wacky thing that just can't be done from within the language itself. With PRMan-3.8 (1998) Pixar introduced the capability to extend the RSL through the use of C (or C++) DSO shadeops. (More trivia, this feature came into being originally to allow BMRT to be called through PRMan (aka franken render) for a couple of ray traced refraction shots in "A Bugs Life".)

Now a days, many of us have taken this feature for granted and just naturally turn to writing a C/C++ DSO when there isn't a obvious or simple way to do it in the RSL.

There was always one nagging issue with these DSO extensions. The shaders in PRMan execute in SIMD (Single Instruction Multiple Data). Which means that the render operates the same instruction across the entire shading grid in parallel (one instruction, multiple data). To see an example of this put some print statements in different locations of your shader. Instead of seeing the print statements serially as you see them in your source, you will get a set of print statements related to your first one (across the shading grid), next another set for the next print statement, etc.

```
surface show_simd() {

    uniform float i = 1;

    printf ("%.0f: s %.02f t %.02f\n", i, s, t);
    . . .
    i =   2;
    printf ("%.0f: s %.02f t %.02f\n", i, s, t);
    . . .
}
```

If the render used your shader in SISD (Single Instruction Single Data – not in parallel but serially) as many people are use to thinking about their programs, you would expect to see something like:

1.0: s 0.00 t 0.00
2.0: s 0.00 t 0.00

1.0: s 0.00 t 0.03
2.0: s 0.00 t 0.03
. . .

Instead what you will see is something like:

1.0: s 0.00 t 0.00
1.0: s 0.00 t 0.03
. . .
2.0: s 0.00 t 0.00
2.0: s 0.00 t 0.03
. . .

This is due to the PRMan shaders executing in a SIMD parallel fashion. While shaders, in PRMan, are executed as SIMD, any DSO shadeop that is called occur in a serial manner. Which is to say the entire DSO performs its entire call and then returns one shading sample at at time.

Now, in PRMan-13, the DSO shadeop calls can be run as SIMD, too! This means the DSOs are given their arguments as arrays of data (to cover the shading grid) to operate on. Instead of before where just the current shading sample was being passed in, worked on, and returned. So as you can imagine, with the introduction, of SIMD C/C++ base DSOs there is now a whole new interface for creating these functions.

First let's review what things looked like before with the older serial interface.

(from the PRMan docs “Adding C Functions to Shading Language with DSOs” September 1999)

```
#include "shadeop.h"

SHADEOP_TABLE(sqr) =
{
        { "float sqr_f (float)", "", "" },
        { "point sqr_triple (point)", "", "" },
        { "vector sqr_triple (vector)", "", "" },
        { "normal sqr_triple (normal)", "", "" },
        { "color sqr_triple (color)", "", "" },
        { "" }
};

SHADEOP (sqr_f)
{
        float *result = (float *)argv[0];
        float *x = (float *)argv[1];
        *result = (*x) * (*x);
        return 0;
}
```

```
SHADEOP (sqr_triple)
{
        int i;
        float *result = (float *)argv[0];
        float *x = (float *)argv[1];
        for (i = 0; i < 3; ++i, ++result, ++x)
        {
                *result = (*x) * (*x);
        }
        return 0;
}
```

Before PRMan-13, first one needed to include the RMAN shadeop.h include file so the macros are setup.

The next item is the SHADEOP_TABLE. This defines the interface between the RSL language and the DSO. The macro argument of SHADEOP_TABLE, gives the RSL call name. This is the name that you would use in the RSL code. Each entry in the table gives the name of C/C++ call associated by this RSL call. An entry provides the function arguments and return type along with the C/C++ function call name. RSL is polymorphic, meaning that functions with different arguments and return types can all share the same name. The language determines which version to call based on the types of the values being passed in and what type they are suppose to return. While on the C/C++ side the DSOs are not allowed to be polymorphic. This is the reason for the different entries in the table, to account for the different types that might be called from within the RSL. The table entries end with an empty entry {“”}.

Also note each entry has three components. The first is the C/C++ function interface that we have been describing. The second one is an optional “init” function that is called by the renderer when it starts up. This is provided in case your application needs to do any prep work before it is to actually begin. Then the third component is an optional “cleanup” function. Which would be called once the renderer is done and shutting down.

Each SHADEOP section is the C/C++ function definition listed in the SHADEOP_TABLE. In each function first we pull off the arguments from the RSL. Note the first argument, arg[0] is reserved for the return argument (assuming you have a return argument) and we aways have the function return 0 for a successful completion of a shadeop call.

The source DSO file could have multiple SHADEOP_TABLES (and associated SHADEOPs). Also one could have multiple files (with possibly multiple SHADEOP_TABLES) can be compiled into 1 DSO file.

Here is another example. This one involves string manipulation. As the RSL doesn't handle string manipulation very well, a DSO is usually needed to alter strings. This example makes use of the pcre library (Perl Compatible Regular Expressions). To find out more about this library go to www.pcre.org.

The values from the RSL are passed to directly the DSO functions through by reference. However, string types are wrapped within a struct call STRING_DESC.

```
typedef struct {
    char *s;
    int   buflen;
    char  padding[4*sizeof(float)-sizeof(char *)-sizeof(int)];
} STRING_DESC;
```

Because the render allocates the string passed, you want to be careful of just be overwriting the values passed in passed structure. Typically instead one allocates new space, writes to this new space and give the shader back data using this newly allocated space. An issue with this is that you can't / shouldn't deallocate any of this string memory space that you have passed back to the renderer until after the render is completely finished. This means if you have lots of string manipulations, your memory consumption for the render will increase.

The big question is are your strings long enough and are the enough of them relative to your geometry to cause a big impact in your memory consumption? If you are concerned about this then you might want to consider making use of a caching scheme so you only allocate a string once and reuse it if you come across it again. This should help minimize the memory consumption and reduce some extra computation by being able to reuse the data.

A different method for handling the returning of strings from the DSO back to the RSL is to allocate a static char* variable in the DSO with more than enough memory to handle string length the function would ever need. The intent is that one would never have to worry about allocating new string data, but just reusing the same memory. The advantage with this is that the memory consumption would be minimal and very little computation overhead (compared to either the caching or string duplication methods) since the memory is being shared for every call. The catch with this is that it only works if you can guarantee that the shader will not overwrite this variable again before it would call the DSO again. As the shaders executed in SIMD and the DSOs serially, you knew you were pretty safe as long as use used the returned value before the shader would get to another instance of the call (This is assuming you might use the call multiple times in a shader(s)).

For many years one could get away with the static char* method for years. However, once ray tracing came along, we no longer can count on having returned value used before something overwrites it again. With a ray

tracing call, it has the potential of starting up another shader(s), before you might be done with this value. Which means the value, instead of being correct might contain the data from a different call. So unless you know for sure your DSO call will never by used by any shaders that can ray trace, this option is no longer safe. We are back to either a caching scheme or the more wasteful blind string duplication every time the DSO executes.

The following source DSO file has two shadeops (two TABLES) in it:

1. strreplace, search for a pattern in a source string and replace it with a new string.
2. str_reframeMap looks for a string with a frame number of the form %04 and replaces the frame with a new number passed in to the DSO.

This file has an interface to a string caching mechanism, but the actual code for this feature is not supplied. Either you will need to implement something like it yourself or just make use of the string duplication method.

```
/*
 * Walt Disney Feature Animation
 * Tal Lancaster
 *
 * strreplace.cpp
 *
 *  RenderMan DSO to look for a pattern in a string and to build up a
 *  new string -- ie. string replacement.
 *
 *   strreplace (src, pattern, subst)
 *   strreplace ("rmantex/@ELEM.color.tx", "@ELEM", "myElem")
 *      == "rmantex/myElem.color.tx"
 */

#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pcrecpp.h>
#include <stdio.h>

#include <iostream>
#include <string>

#include <shadeop.h>

SHADEOP_TABLE(strreplace) =
{
  {"string strreplace (string, string, string)", "" /* init*/, "" /* cleanup*/},
  {""}
};
```

```
extern "C"
SHADEOP (strreplace) {
    char del = ':';

    if (argc != 4) return 1;

    STRING_DESC* result = (STRING_DESC*)(argv[0]);
    STRING_DESC* rm_src = (STRING_DESC*)(argv[1]);
    STRING_DESC* rm_pattern = (STRING_DESC*)(argv[2]);
    STRING_DESC* rm_replacement = (STRING_DESC*)(argv[3]);

    bool isfound;
    std::string newstr;

#ifdef STRING_CACHE
    std::string key =
    std::string(rm_src->s) + del + rm_pattern->s + del +
    rm_replacement->s;

    char*& val = LookupStrCache((char*)(key.c_str()), isfound);
    if (!isfound) {
    newstr = rm_src->s;
    pcrecpp::RE(rm_pattern->s).GlobalReplace(rm_replacement->s, &newstr);

    val = strdup (newstr.c_str());
    }
    result->s = val;
#else
    newstr = rm_src->s;
    pcrecpp::RE(rm_pattern->s).GlobalReplace(rm_replacement->s, &newstr);

    result->s = strdup (newstr.c_str());
#endif // STRING_CACHE
    return 0;
}


/* str_reframeMap ()
 *
 * Assuming pattern baseName.%04d.tx replacing the 4 digit padded
 * frame number with the number that is passed in.  If the string
 * doesn't contain a 4 padded number the original string is passed back.
 *
 */

SHADEOP_TABLE(str_reframeMap) =
{
  {"string str_reframeMap (string, float)", "" /* init*/, "" /* cleanup*/},
  {""}
};
```

```
extern "C"
SHADEOP (str_reframeMap) {
    if (argc != 3) return 1;

    STRING_DESC* result = (STRING_DESC*)(argv[0]);
    STRING_DESC* rm_src = (STRING_DESC*)(argv[1]);
    float rm_frame = * ( (float*) argv[2]);

    bool isfound;
    char del = ':';
    std::string newstr;
    std::string pattern = "\\.\\d\\d\\d\\d\\.";

    int iframe = (int) rm_frame;
    char cframe[7];

    sprintf (cframe, ".%04d.", iframe);

#ifdef STRING_CACHE
    std::string key =
    std::string (rm_src->s) + del + pattern + del + cframe;

    char*& val = LookupStrCache((char*)(key.c_str()), isfound);
    if (!isfound) {
    newstr = rm_src->s;

    pcrecpp::RE(pattern).Replace(cframe, &newstr);

    val = strdup (newstr.c_str());
    }
    result->s = val;
#else
    newstr = rm_src->s;
    pcrecpp::RE(pattern).Replace(cframe, &newstr);
    result->s = strdup (newstr.c_str());
#endif // STRING_CACHE
    return 0;
}
```

Enough review, time to go over PRMan-13 SIMD RSL Plugins. So that whole serial methodology that was mentioned before, gone. The interface has been changed, right down to a new include file.

If this wasn't enough, PRMan-13 is now multi-threaded. The good news is only one copy of the RIB and RIB processing is used. Compared to the individual copies that the “-p” -ala netrenderman method provides. The bad news is that all global data is shared. This includes global data in DSOs!

If you want to get a better understanding of multi-threading in general go to your favorite web search engine and look for words like: atomic action, synchronization, mutual exclusion.

So without really getting into an explanation of these, here are the implications, that one should be concerned with when multi-threading is involved:

- Be careful with static qualifications and any global data.
- Either get rid of these if possible; localize the data; or pursue a lock/unlock thread method for reading and write global data.

More on this later. First let's go over just getting a handle on the new DSO interface by converting our original pre-13 call to use the new interface.

```
#include "stdio.h"
#include "RslPlugin.h"

extern "C" {

static RslFunction shadeops[] =
{
      { "float sqr (float)", "sqr_f"},
      { "point sqr (point)", "sqr_triple"},
    { "vector sqr (vector)", "sqr_triple"},
    { "normal sqr (normal)", "sqr_triple"},
    { "color sqr (color)", "sqr_triple"},
      NULL
};

RSLEXPORT RslFunctionTable RslPublicFunctions = shadeops;

RSLEXPORT int sqr_f (RslContext* rslContext,
int argc, const RslArg* argv[])
{
        RslFloatIter result (argv[0]);
        RslFloatIter x (argv[1]);

    int numVals = argv[0]->NumValues();

    // run through all of the active shading points
    for (int i = 0; i < numVals; ++i) {
        *result = *x * *x;
        ++result; ++x;
    }
        return 0;
}

RSLEXPORT int sqr_triple (RslContext* rslContext,
```

```
                    int argc, const RslArg* argv[])
{
  RslPointIter result (argv[0]);  // points, normals, colors float[3]
  RslPointIter x (argv[1]);

  int numVals = argv[0]->NumValues();
  for (int i = 0; i < numVals; ++i) {

      (*result)[0] = (*x)[0] * (*x)[0];
      (*result)[1] = (*x)[1] * (*x)[1];
      (*result)[2] = (*x)[2] * (*x)[2];

      ++result; ++x;
  }
  return 0;
}

} // extern "C"
```

The include file now RslPlugin.h. The SHADEOP_TABLE has been broken into two sections:

```
static RslFunction shadeops[] =
{
        { "float sqr (float)", "sqr_f"},
        { "point sqr (point)", "sqr_triple"},
  { "color sqr (color)", "sqr_triple"},
        NULL
};

RSLEXPORT RslFunctionTable RslPublicFunctions = shadeops;
```

The first is an array of RslFunction. The second is assigning this into a variable RslPublicFunctions of type RslFunctionTable.

Looking at the array of RslFunction above we see each array element has within it two sub-elements. The first is the interface to the RSL itself (its arguments, RSL name, and return type). The second sub-elment is the name of the C++ call itself. There are two more optional substrings that aren't given above. The third argument would represent the initialization function (much like the old interface provided). The fourth sub-element would represent a cleanup function to be used when the renderer completes (again like the old interface).

NOTE unlike the old interface, a DSO can only have one RslFunctionTable. Which in turn means only one RslFunction array. So if you want a DSO to be made up of a set of files, only one static RslFunction and one RslFunctionTable can exist in the compiled DSO. So if these other files themselves have DSO functions these

will have to be listed in one RslFunction array!

The next section in the example is the DSO function itself. In this case sqr_f, which is the float version of the RSL call. Every DSO function interface will look exactly like this one.

```
RSLEXPORT int sqr_f (RslContext* rslContext,
int argc, const RslArg* argv[])
{
 . . .
}
```

The only difference is the name of the call itself. One could almost just have a macro like:

```
#define RSL_DSO(funcname) RSLEXPORT int funcname (RslContext* reslContext, int
argc, const RslArg* argv[])
RSL_DSO (sqrf)
{ ...}
```

Looking within the function body:

```
RslFloatIter result (argv[0]);
RslFloatIter x (argv[1]);

int numVals = argv[0]->NumValues();

// run through all of the active shading points
for (int i = 0; i < numVals; ++i) {
    *result = *x * *x;
    ++result; ++x;
    }
return 0;
```

Under the old interface we just accessed the arguments as simple pass by reference. That was when things were simple and we just worked on the current point serially. However the as the DSO is executed in a SIMD fashion, we have to be prepared to handle all of the grid's shading points.

In *PRMan* a plugin function will operate over a collection of points at the same time. However, not all of the

points in the collection will be active at any given time (due to the nature of loops or conditional statements in the shader). In order to perform operations on only those points that are active the plugin interface provides iterators. The data for each active point of a given argument is accessed through an `RslIter` object of a given type. (From SIMD RenderMan Shading Language Plugins document November, 2005)

The first two lines, of the DSO call we get the arguments from the RSL and assign them to iterators of the proper type:

```
RslFloatIter result (argv[0]);
RslFloatIter x (argv[1]);
```

Next we find out how many active points were are working with the iterator method, numVals().

```
int numVals = argv[0]->NumVals();
```

This method returns the active shading point list. If you intend to call your function, in the RSL, as val = myDSOfunc(), or any context that is expecting an explicit returned value when the DSO has returned, then the DSO must place the result into argv[0] which is reserved for this usage. This is exactly the same requirement as the under the old interface.

In the section of code, the DSO runs through the active points performing the desired action on each point within the for loop. Then active points are accessed by incrementing the iterators to the next active point.

There are several things to note. In the PRMan SIMD plugin docs, they state that the prefix operator ++val is more efficient than the postfix operator val++ for the iterators. Also the docs state that one doesn't need to be concerned with using the iterators on uniform arguments. There isn't suppose to be any cost for doing a needless prefix ++ operations on them. So it should be just fine to just use the a **Rsl**type**Iterator** on any of the input arguments regardless of them being uniform or varying.

Another thing, if your DSO is a void function or returns any output values, then the NumVals method can't be used. Instead one would need to do something like:

```
int numVals = RslArg::NumValues(argc, argv);
```

This forces the system to examine all of the functions arguments to determine the mix of varying and uniform argument iterations.

There is one more new restriction (read protection). Technically in the RSL one can not assign a varying variable to a uniform variable:

```
extern varying float t;
uniform float myuniform = t;
```

The shader compile will catch this and not let it go through. Under the old interface the shader compiler would have let the following slide:

```
extern varying float t;
uniform float myuniform = myDSO(t);
```

The shader compiler is now on the lookout for this case and will not allow a DSO with varying arguments to be assigned to a uniform value. So if there is such a need this will need to be done by passing in the uniform variable as an output argument to the DSO.

Let's look at the second example converted to the SIMD interface. There isn't much here that hasn't already been covered already. However, there are two things worth mentioning:

1. The STRING_DESC structure has been replaced by a RslStringIterator. As with the other iterators, dereferencing this type gives us a char* (or whatever type the iterator is).
2. The two shadeop calls are in one RslFunctionTable. Instead of the old version where there were two sets of SHADEOP_TABLEs in the same file.

```
/*
 *  Walt Disney Feature Animation
 *  Tal Lancaster
 *
 *  RenderMan DSO to look for a pattern in a string and to build up a
 *  new string -- ie. string replacement.
 *
 *   strreplace (src, pattern, subst)
 *   strreplace ("rmantex/@ELEM.color.tx", "@ELEM", "myElem")
 *      == "rmantex/myElem.color.tx"
 */
```

```
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <pcrecpp.h>
#include <stdio.h>
#include <rx.h>

#include <iostream>
#include <string>

#include "RslPlugin.h"

extern "C" {
static RslFunction shadeops[] =
{
    {"string strreplace (string, string, string)", "strreplace"},
    {"string str_reframeMap (string, float)", "str_reframeMap"},
    NULL
};

RSLEXPORT RslFunctionTable RslPublicFunctions = shadeops;

RSLEXPORT int strreplace (RslContext* rslContext,
                      int argc, const RslArg* argv[])
{
    char del = ':';

    if (argc != 4) return 1;

    RslStringIter result (argv[0]);
    RslStringIter rm_src (argv[1]);
    RslStringIter rm_pattern (argv[2]);
    RslStringIter rm_replacement (argv[3]);

    std::string newstr;

    int numVals = argv[0]->NumValues();

    for (int i = 0; i < numVals; ++i) {
    bool isfound = 0;

#ifdef STRING_CACHE
    std::string key =
        std::string(*rm_src) + del + *rm_pattern + del +
        *rm_replacement;

    char*& val = LookupStrCache((char*)(key.c_str()), isfound);
#endif // STRING_CACHE

    if (!isfound) {
        newstr = *rm_src;
        pcrecpp::RE(*rm_pattern).GlobalReplace(*rm_replacement, &newstr);
```

```
#ifdef STRING_CACHE
        val = strdup (newstr.c_str());
#else
        *result = strdup (newstr.c_str());
#endif
    }

#ifdef STRING_CACHE
    *result = val;
#endif

    ++result; ++rm_src; ++rm_pattern; ++rm_replacement;
    }
    return 0;
}

/* str_reframeMap ()
 *
 * Assuming pattern baseName.%04d.tx replacing the 4 digit padded
 * frame number with the number that is passed in.  If the string
 * doesn't contain a 4 padded number the original string is passed in.
 *
 */
RSLEXPORT int str_reframeMap(RslContext* rslContext,
                      int argc, const RslArg* argv[])
{
    if (argc != 3) return 1;

    RslStringIter result (argv[0]);
    RslStringIter rm_src (argv[1]);
    RslFloatIter rm_frame (argv[2]);

    char del = ':';
    std::string newstr;
    std::string pattern = "\\.\\d\\d\\d\\d\\.";

    int numVals = argv[0]->NumValues();
    for (int i = 0; i < numVals; i++) {

    bool isfound = 0;

    char cframe[7];

    sprintf (cframe, ".%04d.", *rm_frame);

    //printf ("cframe %s\n", cframe);

#ifdef STRING_CACHE
    std::string key =
        std::string (*rm_src) + del + pattern + del + cframe;

    char*& val = LookupStrCache((char*)(key.c_str()), isfound);
```

```
#endif

    if (!isfound) {
        newstr = *rm_src;

        pcrecpp::RE(pattern).Replace(cframe, &newstr);

#ifdef STRING_CACHE
        val = strdup (newstr.c_str());
#else
        *result = strdup (newstr.c_str());
#endif
    }
#ifdef STRING_CACHE
    *result = val;
#endif
    ++result; ++rm_src; ++rm_frame;
    }
    return 0;
}

} // extern "C"
```

So that covers some of the basic changes as they relate to the SIMD interface. However, there are a many more methods available in the new interface. Such as the iterators only provide access to the active points. There may be cases where you will want to access all points (block copies) not just the active ones. In this situation, you will want one of the other methods for getting at the SIMD data and not use the iterators. But they aren't as efficient as accessing sparse grids via the iterators. To find out more check out the PRMan docs on the "RSL Plugin API Reference".

The next thing that might cause people some grief when porting their older shadeops over is to account for the fact that PRMan is now multi-threaded. Without getting into the definition of what multi-threading is, here are some quickey things to keep in mind. When putting the PRMan into multi-threaded mode, (which depends on the settings in your rendermn.ini file, the number of processors in your machine, and how you supply the -t option {-t, -t:4, etc.}), the renderer will commit N threads to render the current shading grid. Threads are executed in parallel and share the same global data.

If the renderer is executing in multi-threaded mode, this means your DSOs are being executed by multiple threads, too. So you need to be careful about making use of any global data your DSO needs to write to. Ideally you would want to write the code so that it doesn't depend on any global that that needs to be written to. Sometimes this just can't be avoided. When this arises the most expedient way of handling this is to force the threading system to execute certain code sections (namely when reading and writing to global variables) serially,

allowing only a single thread to access the data at a time. If you don't follow this then you are essentially, not making your code thread safe. Which means (assuming you actually can get a render), your renders most likely will not produce consistent results.

The process of forcing the threads to execute portions of your code in serial is platform dependent. You will need to find the specifics of your platform to access the necessary threading interface. Under Linux, the default threading environment is pthreads.

Let's look at the timer DSO that we use to profile our shaders. In this example there are several pieces of global data being used.

The use of this DSO is talked about a previous section of the course notes, called "Disney Shader Profiling".

(NOTE: this example is linux centric. Both in its use of pthreads and the timing interface.)

```
/*
 *
 * PRMan-13 timer shadeop
 *
 * Walt Disney Feature Animation
 * Brent Burley
 */

#include <rx.h>
#include <sys/time.h>
#include <string>
#include <map>
#include <vector>
#include <algorithm>
#include <stdio.h>
#include <pthread.h>
#include "RslPlugin.h"

namespace {
    inline void rdtsc(unsigned long long& ts) {
    __asm__ __volatile__ ("rdtsc" : "=A" (ts));
    }

    class timer {
    unsigned long long ts1, ts2, total, overhead, count;
    public:
    inline void start() { rdtsc(ts1); count++; }
    inline void stop() { rdtsc(ts2); total += ts2-ts1 - overhead; }

    timer() : total(0), overhead(0), count(0)
    {
        static bool calibrated = 0;
```

```
        static unsigned long long _overhead = 0;
        if (!calibrated) {
           calibrated = 1;
           for (int i = 0; i < 10; i++) { start(); stop(); }
           _overhead = total/10; total=0;
        }
        overhead = _overhead;
    }

    unsigned long long gettotal() { return total; }
    unsigned long long getcount() { return count; }
    };

    typedef std::map<std::string, timer> TimerMap;

    struct TimerSort {
    bool operator()(const TimerMap::iterator& a, const TimerMap::iterator& b)
    { return a->second.gettotal() > b->second.gettotal(); }
    };

    TimerMap* timers = new TimerMap;

    timer& GetTimer(const char* name)
    {
    return (*timers)[name];
    }

    struct PrintStats
    {
    unsigned long long starttics;
    double startsecs;

    PrintStats() {
        rdtsc(starttics);
        struct timeval tv; gettimeofday(&tv, 0);
        startsecs = tv.tv_sec + 1e-6 * tv.tv_usec;
    }

    ~PrintStats()
    {
        unsigned long long stoptics; rdtsc(stoptics);
        struct timeval tv; gettimeofday(&tv, 0);
        double stopsecs = tv.tv_sec + 1e-6 * tv.tv_usec;
        double totalSecs = (stopsecs - startsecs);
        double secsPerTick = totalSecs / (stoptics - starttics);

        std::vector<TimerMap::iterator> sorted;
        for (TimerMap::iterator i = timers->begin(); i != timers->end(); i++)
        {
           sorted.push_back(i);
        }
        if (sorted.empty()) return;
        std::sort(sorted.begin(), sorted.end(), TimerSort());
```

```
        // print entries more than 1% (or at least 10 entries)
        double threshhold = std::min(totalSecs / 100,
                                sorted[std::min(9, int(sorted.size()-1))]
                                ->second.gettotal() * secsPerTick);
        printf("%-37s %-16s %s\n", "Timer Name", " Seconds ( pct )",
               "Count");

        std::vector<TimerMap::iterator>::iterator iter;
        for (iter = sorted.begin(); iter != sorted.end(); iter++) {
          const char* name = (*iter)->first.c_str();
          double secs = (*iter)->second.gettotal() * secsPerTick;
          unsigned long long count = (*iter)->second.getcount();
          double percent = secs / totalSecs * 100;
          if (secs >= threshhold)
              printf("  %-35s %8.2f (%4.1f%%) %10.10g\n",
                    name, secs, percent, double(count));
        }
        fflush (stdout);
    }
    } printstats;
}

static bool useTimer = 0;

pthread_mutex_t mutex_timerShadeop = PTHREAD_MUTEX_INITIALIZER;

extern "C" {
static RslFunction shadeops[] =
{
    {"float timerStart(string)", "timerStart", },
    {"float timerStop(string)", "timerStop", },
    NULL
};

RSLEXPORT RslFunctionTable RslPublicFunctions = shadeops;

RSLEXPORT int timerStart (RslContext* rslContext,
                  int argc, const RslArg* argv[])
{
    static bool initialized = 0;

    // locking everything for now. Maybe can get better granularity
    pthread_mutex_lock (&mutex_timerShadeop);

    if (!initialized) {
    initialized = 1;
    int option=0, count=0;
    RxInfoType_t type;
    if (RxOption("user:usetimer", &option, sizeof(option), &type, &count) == 0 &&
        type == RxInfoInteger && count == 1 && option > 0) useTimer = 1;
    const char* env = getenv("USE_SHADER_TIMER");
    if (env) { useTimer = (strcmp(env, "0") != 0); }
    if (useTimer) {
```

```
        printf("Profiling timer enabled. "
               "Statistics will be printed at end of render.\n");
        //fflush (NULL);
    }
    }
    if (!useTimer) {
    pthread_mutex_unlock (&mutex_timerShadeop);
    return 0;
    }
    if (argc != 2) {
    pthread_mutex_unlock (&mutex_timerShadeop);
    return 1;
    }

    RslFloatIter  result (argv[0]);
    RslStringIter timerName (argv[1]);

    int numVals = argv[0]->NumValues();

    for (int i = 0; i < numVals; ++i) {
    GetTimer(*timerName).start();

    // not really returning anything
    ++result;
    ++timerName;
    }
    pthread_mutex_unlock (&mutex_timerShadeop);

    return 0;
}

RSLEXPORT int timerStop (RslContext* rslContext,
                  int argc, const RslArg* argv[])
{
    if (!useTimer)
    return 0;

    if (argc != 2)
    return 1;

    RslFloatIter result (argv[0]);
    RslStringIter timerName (argv[1]);

    int numVals = argv[0]->NumValues();

    pthread_mutex_lock (&mutex_timerShadeop);

    for (int i = 0; i < numVals; ++i) {
    GetTimer(*timerName).stop();
    ++result; ++timerName;
    }

    pthread_mutex_unlock (&mutex_timerShadeop);
```

```
    return 0;
}

} // extern "C"
```

static bool useTimer is the first example is the global. This is used to determine if timers are to be used for this render execution. It is set in the timerStart function through checking the status of a user Option or an shell environment variable. Another example, of global usage, is **static bool initialized** (In the timerStart call). Which is used to see if this is the first call to the timers or not.

Under pthreads the way to set aside a section of code so that only one thread can operate it at at time (ie. executed serielly) is through the pthread_mutex_lock and pthread_mutex_unlock functions (mutually exclusive lock and unlock). Making the code in between the lock and unlock is only allowed to be used by one thread at a time. These functions take an argument of type pthread_mutex_t. Typically one defines a variable of this type at the same scope level as the data being shared by the threads.

Right after the declaration of the useTimer, there is the following line:

```
pthread_mutex_t mutex_timerShadeop = PTHREAD_MUTEX_INITIALIZER;
```

Which is the variable we will use to control which sections of the timer are to be only run by one thread at a time. Near the beginning of the timerStart call there is:

```
pthread_mutex_lock (&mutex_timerShadeop);
```

This says from here on, until a unlock for the same variable, is seen only one thread at a time can operate. As stated before, typically one wants to lock and unlock just before something is being done with a global/shared variable. But in this example pretty much the entire function is being told to only allow one thread at at time to work with it. The assumption, for this example, is that there isn't that much code so the overhead of locking and locking multiple in times to protect the shared data, to allow some parallelism, would be higher than locking and unlocking once essentially making this call serial.

In this function there are multiple unlocks but for the mutex_timerShadeop:

```
pthread_mutex_unlock (&mutex_timerShadeop);
```

They are just making sure the lock is released by the time the function completes. So the first one unlocks before the return if timers aren't being used. The second if the argument count is wrong. The last at the end of the function.

In the other timer shadeop call, timerStop, the lock and unlock surrounds the SIMD section which is where the global data is being effected.

More details can be found in the Pixar "SIMD RenderMan Shading Language Plugins" document. These include interfaces for handling certain global data in other ways.

XML Profiling

A earlier section touched on the Disney shader/DSO profiling system and the benefits of profiling in general. PRMan-13 provides profiling support natively.

While the Disney method adds less then 1% to the render time,, according to the Pixar docs they expect an increase of less than 5% and a few 100k per shader. But once you see what it gives you, will understand why and hopefully be convinced that it is well worth it.

To of the Pixar PRMan-13 documents that should be helpful are: "RenderMan Shader Profiling" and "Using XML Frame Statistics". While we won't be getting into the new XML statistics output here, this document does give more background with XML in general and ways of processing the data.

To have get XML profiles out of the renderer there is a minimum of two things that you will need to do:

1. Add

   Option "statistics" "shaderprofile" ["profile.xml"]

   to the RIB stream. This is where you specify the profile output filename.

2. Compile all of your shaders with the shader compiler from PRMan-13. Also this assumes you are using the default -g option and not -g0. If you compiled your shaders with -g0 then this would have the same effect as trying to use shaders that haven't been compiled with 13. Which is to say, you wouldn't get any profiling information.

Once the render completes, it you will see a profile file under the name you provided to the renderer. Open this in your browser. Once loaded you should see a list of shading “hotspots”. These hotspots include elapsed time, source filename, line number and the calling function.

Clicking in the box on the left will open up items will allow you to see the calling context of the hotspots. Assuming more that the context has more than one caller, you will see the callers and a percentage that each of these callers contributed to the overall time spend in this call.

The line numbers are linked allowing you to browse your source code. Clicking on a line number will open up that source file and jump to that line number. This will be displayed in the bottom pane of the browser.

See the image below.

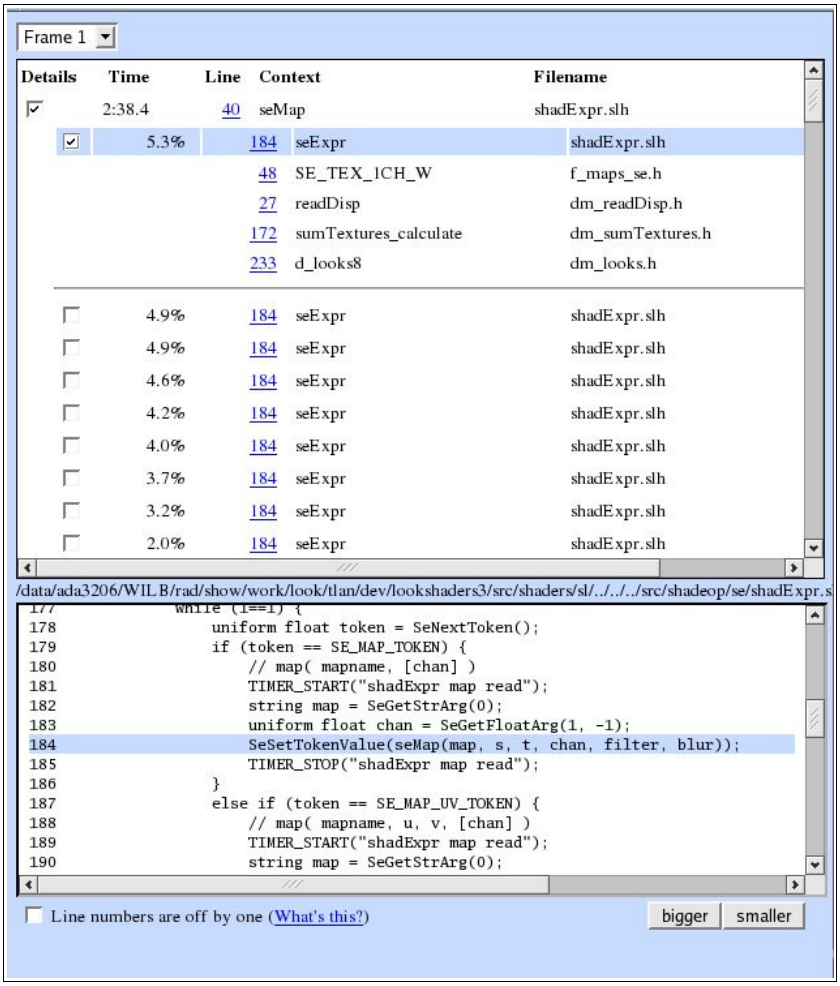


Image 75: Example XML Profile

Production Rendering Acceleration Techniques

Hal Bertram

Interacting With Renderers

The Problem

The main problem with rendering is the time that it takes to create an image. Render times have not changed much since the beginning - the scene and shading complexity have increased at about the same rate that the renderers and the hardware have got faster.

When you are tweaking a scene, even very simple modifications generally require the image to be re-rendered. The delay between you making a change and seeing its effect makes the process hard work. This section of the course is about reducing that delay, leading to a more flowing creative process.

The main cases where interactive rendering is useful are lighting and look development. These are the areas of the pipeline that benefit most from being able to see changes on a preview of the final image.

Solutions

Rendering is the process of producing an image from a scene description that consists of a vast number of variables. We can speed up interactive renders by agreeing not to modify a large subset of these variables, so we

can cache the parts of the render that do not depend on the changing variables.

The following techniques are divided into image-based and geometry-based. Each has advantages and disadvantages, largely defined by which set of variables they enable you to change. In almost all cases, the geometry, textures and shader code can not be changed during interaction.

Demos

The demos presented (other than renderers) are created by a group of interaction tools available from http://halbertram.com/siggraph/

Image-Based Techniques

The great advantage of image-based techniques is that the size of the data you are manipulating is proportional to the resolution of the image. Even if your scene data is many gigabytes, the images are still fairly small. This obviously leads to quicker interaction.

Clearly the main disadvantage is that you can't move the camera. Given that you are generally adjusting lighting, this is not too much of a restriction as the camera is likely to have been already placed. Also, by rendering the scene with several cameras, it is possible to see the effects of your changes in multiple views simultaneously.

These approaches all follow the same pattern. An initial render is performed, and several resulting images are stored. These images are then recombined rapidly as the user adjusts variables.

The type of images rendered, and how they are recombined varies between techniques. We'll look at them in order of complexity.

Render Passes

This is a very old technique, and has traditionally been used to allow a compositor to adjust the levels of aspects of the render in the final shot without having to re-render it. For example in the plastic shader, the final line combines some complex calculations and some very simple ones.

```
Ci = Os * (Cs * (Ka * ambient() + Kd * diffuse(Nf) )
        + specular_color * Ks * specular(Nf, V, roughness) );
```

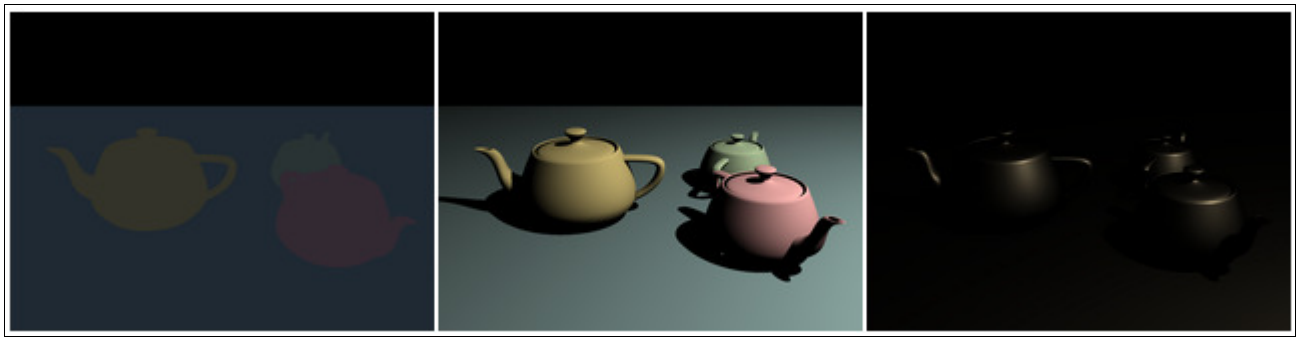
Depending on the scene, it is likely the `diffuse` and `specular` calls involve several lights and shadows. In

comparison multiplying those results by some colors and floats is trivial. But as it stands, we can not change one of the floats without recomputing everything.

The solution is to factor out the time-consuming calculations and store the results in images, so that the simple part can be re-calculated in the compositing system at an interactive rate.

```
Cambient = Cs * (Ka * ambient() );
Cdiffuse = Cs * (Kd * diffuse(Nf) );
Cspecular = specular_color * Ks * specular(Nf, V, roughness);

Ci = Os * (Cambient + Cdiffuse + Cdiffuse);
```

The three components are output as AOVs (below), and in a compositing system the final image can be recreated using a few mix nodes. In some productions these passes will always be combined in compositing, but in others a balance between the passes will be settled on, the mix levels incorporated into the scene, and a single final image rendered.



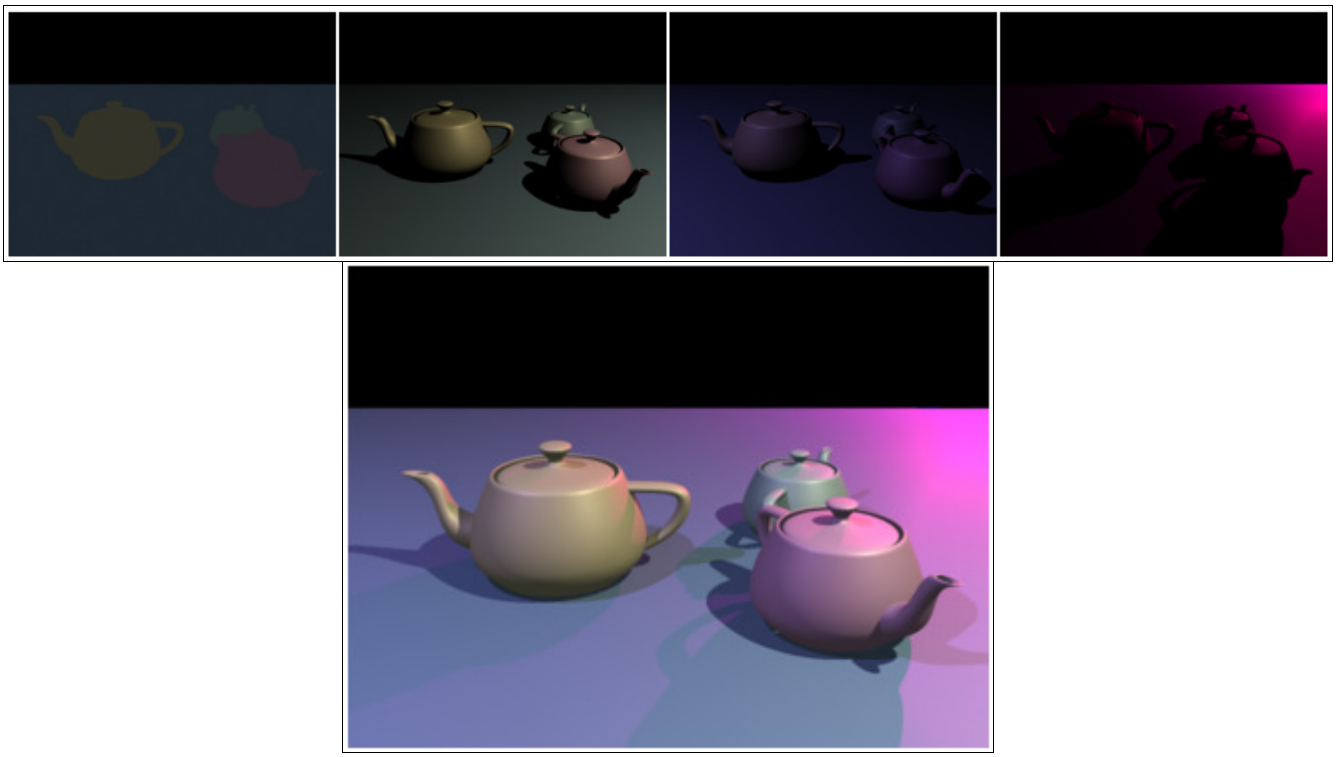
Advantages: simple, easy to implement, and can be manipulated with simple mix nodes in a compositing package. Not limited on the complexity of the lighting system - caustics etc.

Disadvantages: can't alter the intensity, color or position of individual lights.

Light Passes

A variation on render passes is to output the illumination of each light separately. For example, for the scene:

The ambient and three point light images are:



Again they can very simply be combined in a compositing system. Whether you use this approach to produce lighting passes for all your shots, or to determine a lighting setup and fix it into the scene will depend on how much control you want over your scenes after rendering versus the required disc and compositing resources.

Advantages: you can now alter the intensity and color of lights individually. Lighting complexity is still not limited.

Disadvantages: it is more difficult to incorporate into the shaders - they need to output an AOV per light and so must be inside the light loop. The images for this section were created simply by rendering once for each light, with the others turned off. We still can't move the lights.

Light and Render Passes

These two approaches can be combined to render a separate set of passes for each light's contribution. However, very quickly the number of AOV images becomes harder to manage.

A Shading Language Compositing System

There are many compositing systems that can be used for the image manipulation. However, these demos are being produced by a slightly different system - one based on SL rather than node graphs.

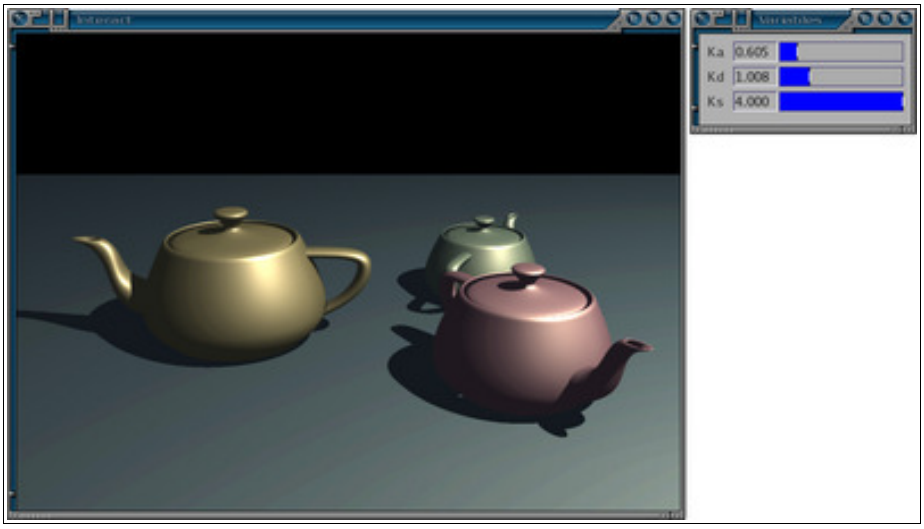
For example the compositing of the first render pass AOVs is described as follows:

```
surface main(
        varying color Cambient = 0;
        varying color Cdiffuse = 0;
        varying color Cspecular = 0;

        float Ka = 1;                   // INTERACT: range=0:4
        float Kd = 1;                   // INTERACT: range=0:4
        float Ks = 1;                   // INTERACT: range=0:4
        )
        {
        Ci = Ka * Cambient + Kd * Cdiffuse + Ks * Cspecular;
        }
```

And the images are brought together in a RIB-like script:

```
Surface "passes" "Ka" [1] "Kd" [0.5]

Geometry "image" "filename" "ambient.tif" "rgb" "Cambient"
Geometry "image" "filename" "diffuse.tif" "rgb" "Cdiffuse"
Geometry "image" "filename" "specular.tif" "rgb" "Cspecular"
```

The idea of the compositing 'finishing off' the surface shader can clearly be seen. This produces a composite image and a simple UI:



This system is not intended to be a complete compositing solution with features like rotoscoping and tracking, just to be a convenient way to interact with composite images combined with shaders. It is designed to be easily scriptable and simple to integrate as a component - a host application can communicate with it via a command port to control it or query the user's UI values.

Implementation

The system translates a subset of RSL to Cg and compiles it for the GPU. This results in high performance at the cost of limits on shader complexity and image number and size.

Shader Metadata

The '`INTERACT:`' comments in the shader above are cues to the system as it builds a UI. There is not yet a standard approach to including metadata in shaders.

Surface Attribute Passes

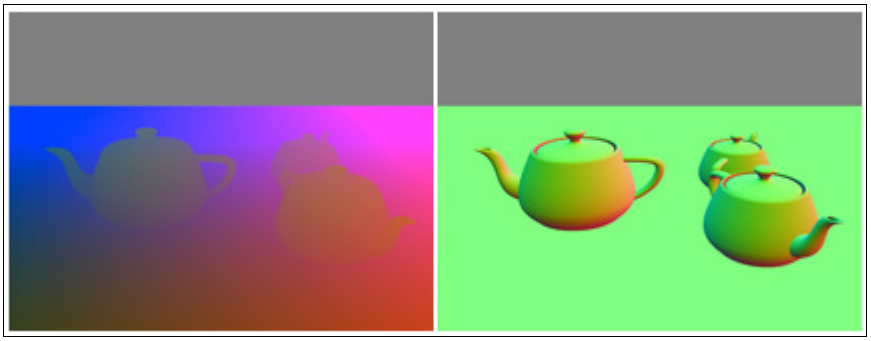
The render and light pass techniques assume that all we can do with the images is multiply them with various colors and intensities and add them together. For a long time, this was a reasonable assumption. However, with modern hardware, we can transfer more of the shading equation to the interactive system.

Instead of storing surface illumination values in the AOV images, we store enough data AOVs to recompute the illumination later. All the complex displacement and shading to calculate the exact surface characteristics of

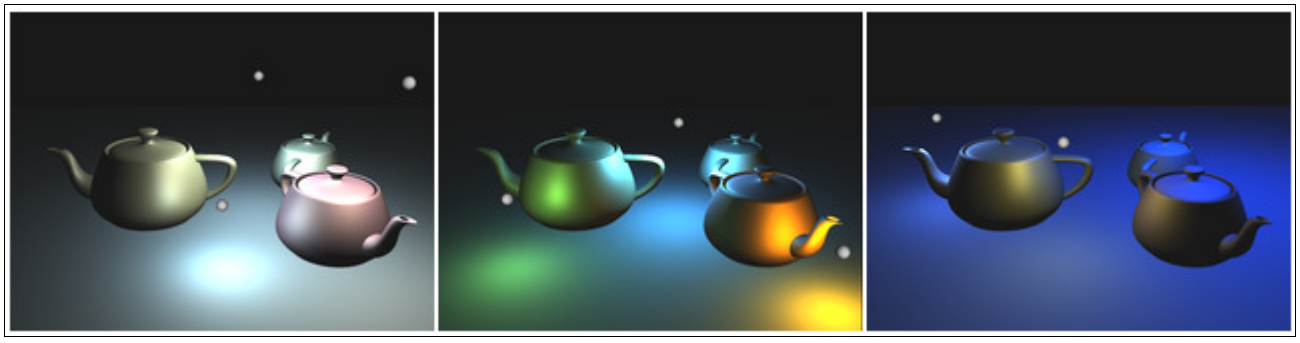
each pixel is calculated by the renderer, but how the light interacts with that surface can be recalculated at an interactive rate.

For example, given a scene composed entirely of plastic objects, we can output AOVs for ambient, diffuse, specular colors, and roughness:

Additionally, as we are discarding the scene data, we need to retain some information about the geometry of the shading point, so we also output AOVs for positions and normals in world space:



There is enough information here to apply whatever basic lighting we want to this scene. Finally we have the ability to control our lights. For example various arrangements of point lights:



To implement the lighting calculation here, we need to mimic the shader illumination code that is in the RSL light shader. The best place to do this is by translating the light shader to Cg and executing it on the GPU.

More Complex Materials

Obviously restricting your scene to only plastic surfaces is not viable, so you need to develop a larger set of AOVs that can characterize all your materials. It makes the system much simpler if there is one set of variables that can be used for every material, otherwise you need to perform multi-pass rendering with different compositing shaders.

Also, if you are prepared to wait long enough for the initial render, you could add global illumination effects like ambient occlusion and color bleeding.

Shadows

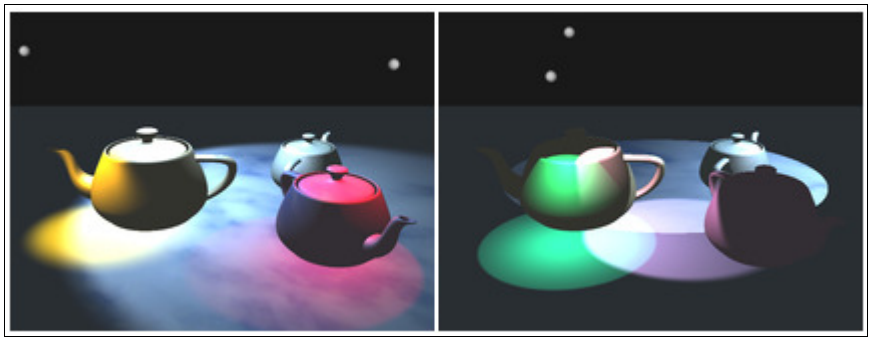
Implementing shadows adds another layer of complexity. This is because it is not possible to capture enough information in the image to recalculate a shadow from a moving light. In fact the object casting the shadow may not appear in the image at all.

As in RenderMan, there are two ways to create shadows. Both need a representation of the shadow casting geometry.

Shadow maps are fairly simple to produce in hardware. The filtering and resolution are not as high quality as in a software render, but are good enough to place the shadow. We will look at this in the geometry section.

Complex Lights

The point light example above is far too simple for production use. We need to be able to handle the complexities of a production light shader, or at least a good enough approximation of it. To produce these images, we are just using an existing SL spotlight shader and a projector light:



More complex shaders are possible, but there is a limit to how much processing the GPU can handle in a shader

(certainly to maintain an interactive response). For complex light shaders, it is necessary to create a simplified shader that creates a representative effect, enough to make lighting judgments with.

Other Issues

The surface attribute technique works well for a lot of scenes. It runs into difficulty when a number of different shading points contribute to the final pixel. There are solutions, but each increases the complexity of the system.

Antialiasing

The problem with antialiasing in any form is that the final pixel is calculated from a number of samples. Each of these samples is the result of the shading equation with a slightly different set of parameters. An antialiased pixel is approximately the average of these results. However, if the parameters are being written to images, the averaging will happen at parameter level. When composited the pixel will be the single result of the shading equation with a set of averaged parameters. Depending on the type of parameters, this may be a very different result.

To reduce the effects of this, you can render the AOV images with as little antialiasing as possible: no DOF, no motion blur and limited sub sampling.

Transparency

The problem with transparency is that two or more different surfaces are contributing to the final pixel, so the interactive system has to recompute both illuminations separately and blend them together.

Effectively you need a whole second set of AOV images for surfaces behind transparent objects. Beyond two surfaces would become increasingly inefficient.

Reflection and Refraction

This is similar to transparency, but the extra set of images would need to store the characteristics of the surface that the traced ray hit. This would ensure that as the lighting changed at the reflected surface, its effect would be seen on the reflecting surface.

Hair and Fine Geometry

When potentially hundreds of pieces of geometry are contributing to an image pixel, the AOVs are unlikely to be able to store a representative sample that will produce a meaningful recomputed illumination. In this case, either

omitting the element or using stand-in geometry would be advisable.

Volumetrics and Deep Shadows

Though it would be possible to extend the surface attributes to include these rendering features, the system and the AOV data would probably become too complex to be worth the effort.

Conclusion

Advantages: we can adjust light intensity, color, and position. In fact, we can adjust any lighting parameter.

Disadvantages: fixed camera, shadow rendering difficult. Some minor discrepancies due to differences between software and GPU shading.

Geometry-Based Techniques

Geometry-based techniques free you from the restriction of a static camera, but at quite a cost in resources. Whereas image-based techniques effectively compress scene data of any complexity into a number of images, to be able to move the camera we need to retain a lot of that scene data.

The rest of this section describes two very different approaches to this problem.

Baking Micropolygons

The idea of this approach is to store all of the micropolygons that were shaded in a scene, and then use the GPU to display them all from different views. When this was first used it seemed counterintuitive, given the usual concerns about keeping the number of active micropolygons to a minimum. Now it is similar to a mainstream technique - storing all shaded points using `bake3d` for global illumination caches. However, as well as the shading points, we need the grid topology as well. Renderers are beginning to have features to enable this - initially it needed DSO shadeop support.

Apart from the difference in data collection, the geometry can be treated simply as an image that allows the camera to move, with micropolygons taking the place of pixels. As such, similar choices need to be made. We could just store `Ci` for each point, resulting in a completely baked lighting solution but with camera freedom. If there are few specular effects or only subtle camera movement, that may be enough. Alternatively we could output the geometry with all the surface characteristics which would allow us to completely relight the geometry, while being able to move the camera.

When displaying the geometry we still use the same SL shaders as the image-based examples, but in their Cg translation, they pick up their varying inputs from vertex attributes instead of texture maps.

Now we can display the geometry from any angle.

As with `bake3d` processes, we need to be careful with the dicing, culling, and the placement of the camera in the initial render. Geometry that is not shaded will not be in the captured dataset.

Shadows

Since we now have the real scene geometry, we can create shadows using the GPU shadow map functions. They will not be as refined as software shadows, but easily enough to place them in the scene.

Hybrid Techniques

It is still likely that we will move the camera less often than we will adjust the lighting, and since rendering all the geometry takes longer than just rendering a textured quad, we can use the geometry to create images similar to the AOVs from the renderer.

Production Rendering

As well as interaction, this technique can been used as a production final render for a certain class of shot - the camera doesn't need to move too much, nothing in the scene moves, and the lighting is fixed. It is especially effective when the scene is constructed from lots of complex geometry, mostly generated with procedural displacement, and takes an extremely long time to render.

Using this baking technique, all the geometry is output to a large file. The system then reads back the file and hardware renders the scene from a moving camera giving a speed up of several orders of magnitude.

If the scene data is too large to fit into the memory of the workstation, it can be stored on disc in volume tiles, and streamed through the renderer. If the camera movement is slow, several frames and passes can be rendered simultaneously - as each volume tile is loaded, it is rendered onto a number of images. This produces a much higher throughput than having to load all the tiles per image.

Conclusion

Advantages: camera freedom, lighting freedom.

Disadvantages: scene data size, display speed much slower than images, shadow rendering not as good as images containing shadows. Some minor discrepancies due to differences between software and GPU shading.

The Interaction Trick

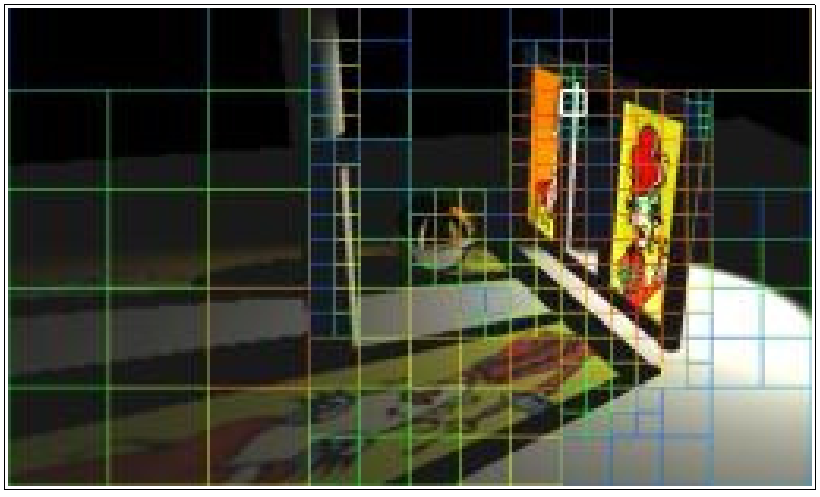
This approach is different from the baking approach because it doesn't manage the scene data itself - it lets the renderer deal with that, and merely asks the renderer to trace rays that it is interested in to produce the image.

Outline

The system consists of a viewer component which handles the display and sends requests down a network socket to a ray server DSO running in the renderer. The ray server in turn takes those requests, traces the required rays, and sends the results back to the viewer display.

This arrangement is similar to the days when PRMan used BMRT to trace rays, before it had native support for ray tracing.

Viewer



The main role of the viewer is to handle the camera position and the progressive refinement of the image. The viewer maintains a pyramid of images and a list of buckets that still need to be rendered, continuously sorting them by a contrast metric. The buckets can be made visible with a temperature scale - when one bucket finishes, the next reddest one is started.

The viewer uses OpenGL to render the multi-resolution images.

Shader

What do we mean by asking the renderer to trace some rays for us? Obviously a renderer usually operates by using your shaders and DSOs as and when it needs to render the final image, and there is no DSO API to trace whatever you want in the scene.

The core of the approach is inverting this relationship and taking control of the renderer. We place a piece of geometry in front of the lens with a very simple shader:

```
surface hb_interact_lens()
    {
    point pos;
    vector dir;
    color col;

    while (1)
        {
        pos = hb_interact_get_origin();
        dir = hb_interact_get_direction();

        col = trace(pos, dir);

        hb_interact_put_color(col);
        }
    }
```

All it does is keep asking the DSO for a position and direction, traces the ray, and sends the result back to the DSO. It never finishes - once this is running, the renderer will trace whatever the DSO wants it to.

Since this is effectively a single threaded program, we don't want the renderer to do any SIMD processing. So, the easiest way to do this is just to put a huge point in front of the camera - since that will only be shaded at one location, rather than a standard grid.

DSO

The DSO handles the other end of the socket to the viewer.

It receives simple packets with enough information in them to describe a camera and section of the image plane:

```
struct Packet
    {
    float origin[3];

    float x[3];
    float y[3];
    float z[3];

    float lx, ly;
    float hx, hy;

    int sx, sy;

    int samples;
    };
```

Once all the required pixels have been returned to the DSO, it sends them back to the viewer as a small image. To interact with the camera the viewer simply adjusts the matrix in the packets sent to the DSO.

Materials

We also want to be able to interact with the materials. This can be done by changing the material shader to pick up the interaction variables from a linear array of floats in the DSO. From:

```
surface test_material(
            uniform color ambient=1;      // INTERACT: color
            uniform color diffuse=1;      // INTERACT: color
            uniform color specular=1;     // INTERACT: color
            )
```

```
    {
    ...
```

To:

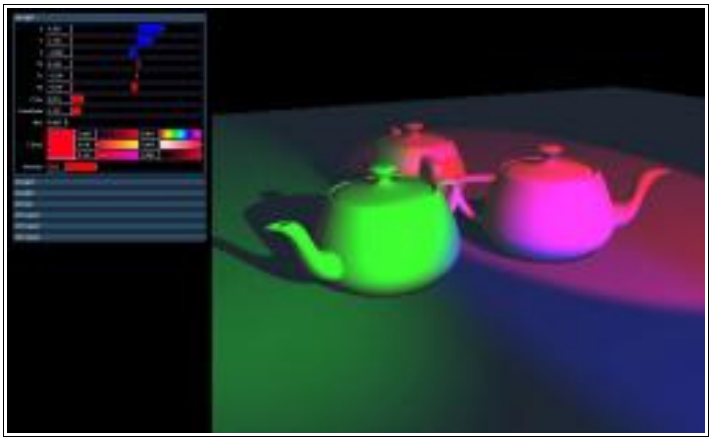
```
surface test_material(uniform float interact_offset = 1)
    {
    uniform color ambient = hb_interact_variable(interact_offset + 0);
    uniform color diffuse = hb_interact_variable(interact_offset + 3);
    uniform color specular = hb_interact_variable(interact_offset + 6);
    ...
```

This system is not particularly elegant, but it is much faster than trying to pass the DSO computed strings or something similar. The usual gains of `uniform`s being calculated much less often than `varying`s do not apply so much when tracing. Originally these lines were inserted manually. Now the system can insert them into shaders automatically.

From the '`INTERACT:`' comments, a UI is constructed.

Unfortunately we can only change variables - we can't switch the shaders or rewire them.

Lights

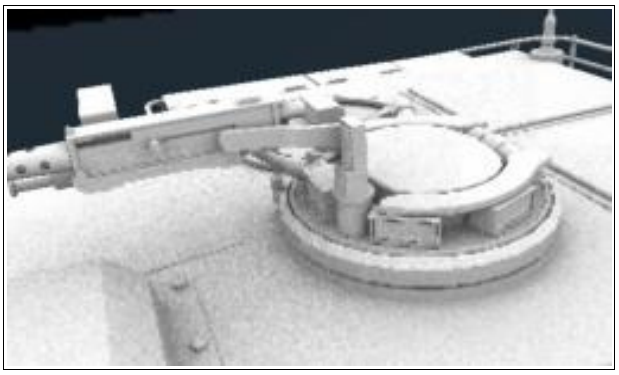


Tweaking lights is very similar to materials.

```
point from = hb_interact_variable(interact_offset);
```

The light positions are retrieved from the DSO and the ray tracing does the rest. Shadow maps could still be used for static lights - variables such as distribution could be changed, but moving the light would produce incorrect results.

Sampling



Lots of shaders sample something else in the scene - like lighting or occlusion. In this case, we want high speed feedback initially, followed by high quality results eventually. To achieve this, the DSO exposes the level of the bucket being processed to the shader so it can decide how many samples to use for the bucket.

```
uniform float samples = hb_interact_level() + 1;
samples = 4 + pow(samples, 3);
```

In the case of occlusion, PRMan tries hard to speed up the rendering by using a cache for occlusion queries, so they are often calculated by interpolating nearby past requests. The interactive shader has to disable this so later requests aren't calculated from previous requests with old parameter values.

Multiple Hosts

A fairly simple extension of this approach is to use multiple hosts to spread the rendering load. As with any interactive application running over a network, you quickly get into limitations such as bandwidth and latency. Ignoring those details, each host behaves just like the single host in that it loads the complete scene and listens for trace requests.

It is then up to the viewer to hand out the buckets that need to be rendered to the list of hosts that are waiting. The image above shows the viewer controlling four hosts, with the thick squares showing buckets in progress, colored by host.

Limitations

There are several significant limitations to this approach. The main one is that we are raytracing everything and so the whole scene must be in memory. It is possible to get almost a million polygon model into a 1GB machine without it swapping. The performance will decrease drastically if the scene does not fit in memory - all the techniques PRMan uses for caching and discarding tessellated geometry will get confused because the viewer keeps firing rays in fairly arbitrary directions.

Also, by the time we are in control, all geometry has been tessellated and displaced, so there is no way to interact with geometry placement or displacements. In theory you could restart a new frame without the user noticing, but it would still take a noticeable amount of time to re-tessellate the scene.

Conclusion

Advantages: freedom of camera, lighting and shader parameters.

Disadvantages: limited scene complexity before renderer runs out of memory.