SIGGRAPHASIA 2010

Introduction to Using RenderMan

Course Notes

Saturday 18 December
Full Day

Instructors

Malcolm Kesson, Savannah College of Art and Design

Hosuk Chang, Blizzard Entertainment

Introduction to Using RenderMan

Course Summary

Saturday, 18 December, Full Day
Level: Beginner

The course will provide an overview of

- The structure of RenderMan scene descriptions.
- The implementation and application of custom shaders.
- The use of RenderMan by the Cinematics Team of Blizzard Entertainment.

The full-day course is an intensive, hands-on practical introduction to the RenderMan system and the use of Pixar's 'prman' and DNA's '3delight' renderers, both of which are widely used in animation and digital effects studios.

In the first part of the course attendees will gain sufficient familiarity with RenderMan's scene description protocol to enable them to edit and manipulate RIB files. RIB files enable modeling and animation applications to pass information about 3D scenes to RenderMan compliant renderers such as 'prman' and '3delight'.

The second part of the course introduces the use of the RenderMan Shading Language (RSL). Attendees are not expected to have prior programming experience. The intention of this part of the course is to provide an overview of the shading language to the point where attendees will be confident to independently explore the creative potential of RSL.

The course will be concluded with a detailed presentation of the advanced RenderMan techniques used by the Cinematics Team of Blizzard Entertainment for the next release of World of Warcraft (winter 2010).

Prerequisites
None

Intended Audience
This course is ideal for artists and designers who have prior experience using a 3D modeling and animation application but who wish to investigate the features of a graphics system that has become the de-facto standard of the feature film industry.

Instructors
Malcolm A. Kesson
Hosuk Chang

Introduction to Using RenderMan
Course Schedule

Many thanks to
Pixar
Blizzard Entertainment
The Savannah College of Art and Design

Session 1 Malcolm Kesson	
9.00 - 10.45 Rib Exercises	**RenderMan rib files** Options, Attributes, camera & world blocks Camera transformations & geometry Archived geometry (pre-baked ribs) AOVs - many outputs from a single render
10.45 - 11.00	Break
Session 2 Malcolm Kesson	
11.00 - 12.45 RSL Exercises	Stereo rendering Procedural primitives - creating geometry at render-time **RenderMan Shading Language** Language overview Patterns (st space)
12.45 - 14.15	Lunch
Session 3 Malcolm Kesson	
14.15 - 16.00	**RenderMan Shading Language** Point-based occlusion Sub-surface scattering
16.00 - 16.15	Break
Session 4 Hosuk Chang	
16.15 - 18.00	**RenderMan in Production** Hosuk will explain the advanced RenderMan techniques used for Blizzard Cinematics.

Instructor Bios

Malcom Kesson teaches graphics programming at the Savannah College of Art and Design. He has presented RenderMan workshops at all five Asia GRAPHITE conferences as well as SiggraphAsia 2008 and 2009.

Hosuk Chang is an effects technical director in the Cinematic Division of Blizzard Entertainment. He is responsible for the creation of elements such as smoke, water, dust and a variety of magical effects in support of the cinematics of games such as World of Warcraft, Star Craft and Diablo. Hosuk also contributes to the on-going development of Blizzard's cinematic FX pipeline.

Introduction to Using RenderMan
Preface

RenderMan is the dominant rendering technology used throughout the feature film industry. For many years the only publication available to CG artists who wished to learn how to use Renderman was "The RenderMan Companion" by Steve Upstill. Even after 19 years it is still a great source of information. However, the "Companion" focused on the use of the 'C' programming language to write RenderMan "programs" and as such it served the needs of programmers rather than CG artists. Perhaps as a consequence of it being seen as just another kind of programming, RenderMan acquired a slew of misconceptions about its role as a creative medium for CG artists.

The Siggraph "Advanced RenderMan" courses taught by Tony Apodaca and Larry Gritz both popularized the subject as well as catered to specialists already working in feature film studios. Their book, "Advanced RenderMan" published in 2000, greatly helped to make Pixar's technology accessible to CG artists. Several excellent publications now provide insights into almost every aspect of RenderMan.

Adoption of the RenderMan standard has led to the development of several compliant renderers. Some, like 'prman' by Pixar and '3delight' by DNA Research are commercial products. Other RenderMan compliant renderers have been developed as open-source projects.

Despite the wealth of information that is now available, this one day course offered at SiggraphAsia 2010 provides an invaluable introduction to those who wish to make a quick start on their exploration of RenderMan.

Malcolm Kesson
Savannah, Georgia.

Recommended Texts

The RenderMan Companion
A Programmers Guide to Realistic Computer Graphics
Steven Upstill
Addison-Wesley
ISBN: 0201508680
1989

Advanced RenderMan
Anthony Apodaca and Larry Gritz
Morgan Kaufmann
ISBN: 978-1-55860-618-0
2000

Essential RenderMan Fast
Ian Stephenson
Springer
ISBN-13: 9781852336089
2003

Texturing & Modeling
A Procedural Approach
David Ebert, F. Kenton Musgrave,
Darwyn Peachey, Ken Perlin & Steven Worley
Morgan Kaufmann
ISBN: 978-1-55860-848-1
2003

Rendering for Beginners
Saty Raghavachary
Focal Press
ISBN-13: 978-0-240-51935-7
2004

The RenderMan Shading Guide
Rudy Cortes & Saty Raghavachary
Course Technology
ISBN-13: 978-1-59863-286-6
2007

Recommended Websites

Hosuk's Personal website at http://s204357084.onlinehome.us.
RManNotes by Steve May at http://accad.osu.edu/~smay/RManNotes.
Selected student pages at www.sfdm.scad.edu/faculty/mkesson.
CG References & Tutorials at www.fundza.com.

Introduction to Using RenderMan
Index

Rib
Setting up a Basic Camera

```
# disk1.rib
# setting a perspective view

Display "disk1" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 320 240 1

WorldBegin
    Disk 3 0.25 360
WorldEnd
```

The purpose of this tutorial is to present a minimal renderman scene. The first two lines show the use of the hash symbol "#" for comments. `Display`, `Projection` and `Format` are rib statements that define the basic characteristics of the camera. These statements, and others that begin each line, must be spelled exactly as shown ie. a single word with one or more upper case letters.

`Display` has three parameters to specify

- the name of the image,
- where to put the image, and
- what information the image should contain.

The parameters for `Projection` specify,

- perspective or orthographic projection, and the
- field of view measured in degrees.

The parameters for `Format` specify,

- image width,
- image height, and
- the pixel aspect ratio.

`WorldBegin` notifies the renderer that objects comprising the 3D scene are about to be defined. `Disk` is a rib statement that defines, by its three numeric parameters a flat circular disk situated 5 units along the z axis, 0.5 units in radius and 360 degrees in circumference. Parameters must be separated by at least one space. They may, however, be spread over several lines and have comments at the end of each line. For example, the disk could have been specified as follows.

```
Disk
5   # units along the z axis
0.5 # units in radius
360 # degrees
```

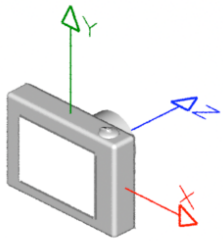
Finally, `WorldEnd` indicates the description of the scene, or world, has been completed. This small rib file is interesting not just for what it describes but also for what it omits. Although it does not specify the material characteristics of the disk, or how it is lit, the renderer is able to produce an image because, in the absence of specific information, it uses default settings.

```
# example1.rib
# setting a perspective view
```

Two comments about the scene.

```
Display "disk1" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 320 240 1
```

Set the camera to give a perspective view with a field of vision of 40 degrees and a frame size of 320 by 240 pixels.

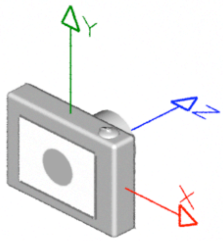


```
WorldBegin
```

Begin describing the contents of the 3D scene.

```
    Disk 3 0.25 360
```

Create a disk 3 units along the z axis of the camera, 0.25 units in radius and 360 degrees in circumference.



```
WorldEnd
```

Conclude the description of the 3D scene.

Rib
Transformations & Attributes

```
# disk2.rib
# setting the world coordinate system

Display "disk2" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 320 240 1

Translate 0 0 3  # a transformation
WorldBegin
    Color 1 0 0  # an attribute
    Disk 0 0.25 360
WorldEnd
```

The primary 3D coordinate system in RenderMan is the camera coordinate system. Until the renderer reads `WorldBegin` the camera defines the **current coordinate system**. From a users point of view, applications such as Maya and Houdini enable camera(s) to be moved within a fixed modeling (world) space. However, internally these applications behave in the same way as RenderMan in the sense that the world is defined after the characteristics of the camera have been established. Therefore, it is the scene that is orientated with respect to a fixed, or primary, camer coordinate system.

This tutorial explains what *transformations* and *attributes* mean in the RenderMan specification. The `Translate` command is one of four transformations. The others are `Rotate`, `Scale` and `Skew`. The effect of these transformations will be seen in the next couple of examples. For now we will focus on translations. The effect of the `Translate` statement is,

- to create a copy of the camera coordinate system,
- to move the copy 3 units along the z-axis of the camera.

When the renderer reads `WorldBegin` a copy of the camera coordinate system, nov moved out in front the camera, becomes the primary or current coordinate system. I is named the "**world**" coordinate system. Transformations are accumulative. For example, these two translations,

```
Translate 0 0 3
Translate 0 0 3
```

have exactly the same effect as a single translation ie.

```
Translate 0 0 6
```

The `Color` statement sets a RGB value. Attributes are not acculumative ie.

```
Color 1 0 0
Color 0 0 1
```

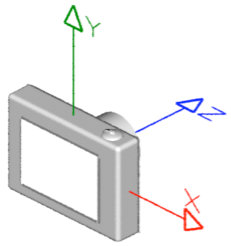
does not specify purple. The second `Color` statement makes blue the current color. In effect, the second statement replaces or hides the first color statement. The RGB color components must be in the range 0.0 to 1.0. Unlike Maya, geometry in RenderMan can be colorized without the use of a material. `Opacity` is another attribute that can effect the transparency of an object irrespective of an objects "material" properties.

```
# disk2.rib
# setting the world coordinate system
```

Two comments about the scene.

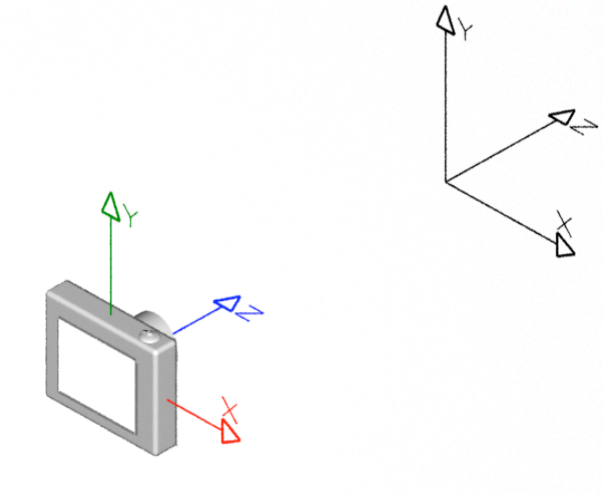
```
Display "disk2" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 320 240 1
```

Set the camera to give a perspective view with a field of vision of 40 degrees and a frame size of 320 by 240 pixels.



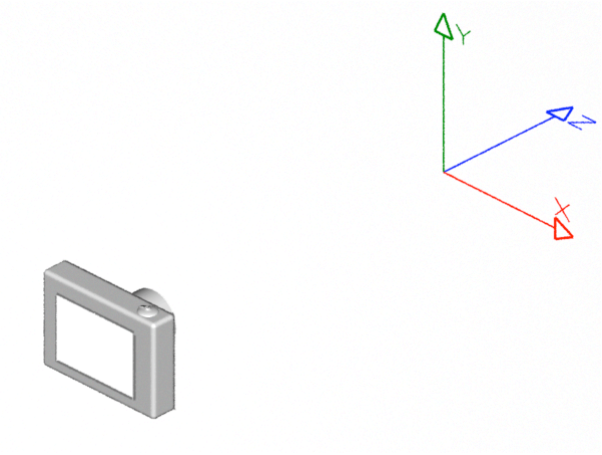
```
Translate 0 0 3  # a transformation
```

Create a copy of the camera coordinate system and move it 3 units along z-axis. Note: it is the copy that is translated NOT the camera coordinate system - it remains fixed.



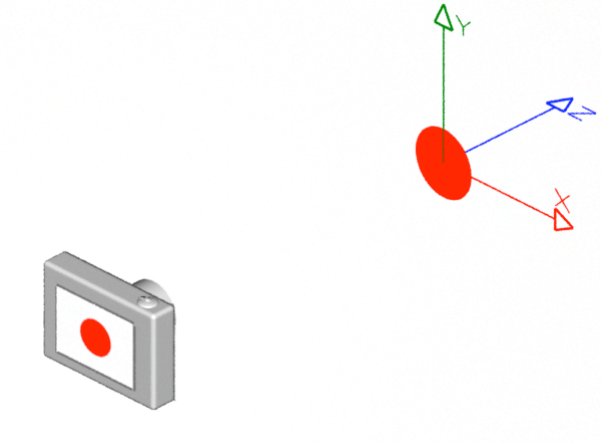
```
WorldBegin
```

The copy of the coordinate system now becomes the primary, "world", or current coordinate system.




```
Disk 0 0.25 360
```

Create a disk at the origin of the "world", 0.25 units in radius and 360 degrees in circumference.




```
WorldEnd
```

Conclude the description of the 3D scene.

Rib
Camera Transformations

```
# disk3.rib
# applying multiple transformations

Display "disk3" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 320 240 1

Translate 0 0 3
Rotate -40  1 0 0
Rotate -20  0 1 0
WorldBegin
    Color 1 1 0.7
    Polygon "P" [-0.5 0 -0.5  -0.5 0 0.5  0.5 0 0.5  0.5 0 -0.5]
            "st" [0 0  0 1  1 1  1 0]
    Color 1 0 0
    Disk 0 0.25 360
WorldEnd
```

The purpose of this rib file is to show the effect of applying additional transformations before `WorldBegin`.

```
    Translate 0 0 3
    Rotate -40  1 0 0
    Rotate -20  0 1 0
    WorldBegin
```

These transformations determine how the 3D scene will be viewed by the camera. For that reason they are know as the **view** or **viewing transformations**. Rib files written by Maya (requires Pixar's plugin) use a single transformation command,

```
    ConcatTransform [ 0.707 -0.331 -0.625  0.0
                      0.0    0.884 -0.469  0.0
                     -0.707 -0.331 -0.625  0.0
                      0.0    0.0   44.822  1.0 ]
```

This statement specifies a transformation matrix of 16 values. Fortunately, the RenderMan standard provides a more human-friendly way of setting the viewing transformation. The `Rotate` statement has four parameters,

- an angle measured in degrees, followed by
- the xyz coordinates of the axis of rotation.

A good way of understanding the last three values of the command is to consider the one and zero's to be switches ie.

```
Rotate -40    1   0   0
       angle  on  off off
```

Therefore, this statement specifies a rotation of -40 degrees around the x-axis. The direction of the rotation, clockwise or anti-clockwise, is explained in the next tutorial "Rib: Left-hand & Right-hand Coordinate Systems". For the moment, the key points to understand about transformations are that they are,

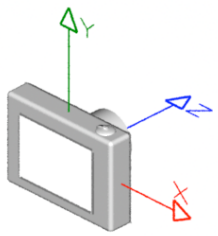
- applied in reverse order,
- applied to a copy of the current coordinate system,
- applied relative to the the current coordinate system.

```
# disk3.rib
# applying multiple transformations

Display "disk2" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 320 240 1
```

Two comments about the scene.

Set the camera to give a perspective view with a field of vision of 40 degrees and a frame size of 320 by 240 pixels.

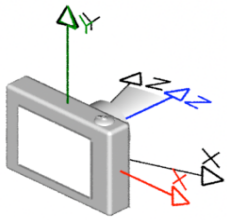


The camera coordinate system is the current (active) system.

```
Translate 0 0 3
Rotate -40  1 0 0
Rotate -20  0 1 0
```

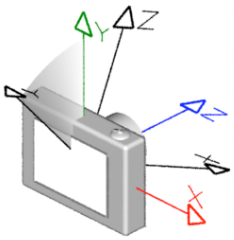
The transformations are applied in reverse order.

First, the negative rotation of 20 degrees around the y-axis is applied to a copy of the current coordinate system.



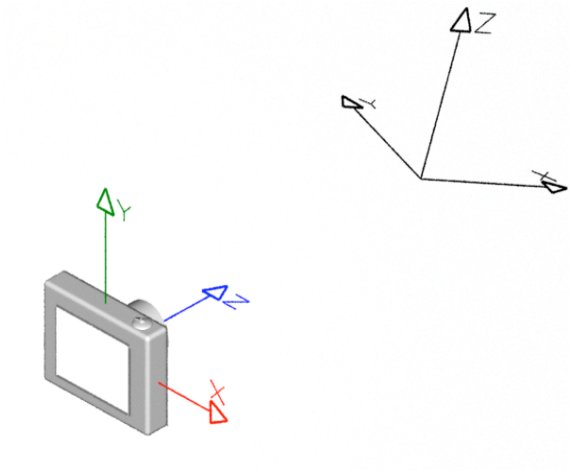

```
Translate 0 0 3
Rotate -40  1 0 0
Rotate -20  0 1 0
```

Next, the negative rotation of 40 degrees around the x-axis is applied to the copy.



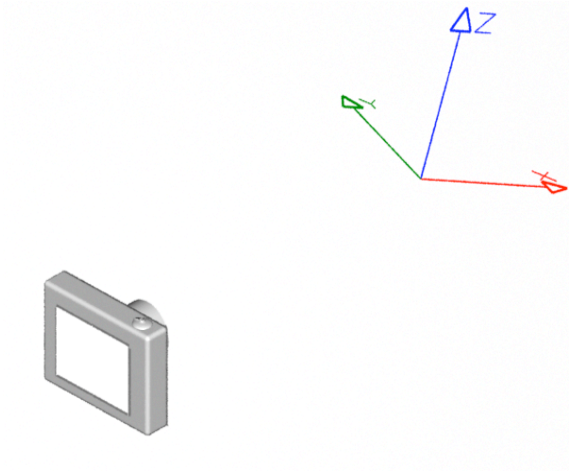
```
Translate 0 0 3
Rotate -40 1 0 0
Rotate -20  0 1 0
```

Move the transformed coordinate system 3 units along the z-axis of the camera.




```
WorldBegin
```

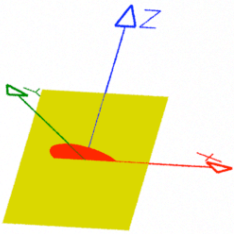
The copy of the coordinate system now becomes the primary, "world", or current coordinate system.

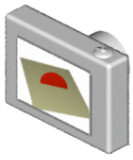



```
Color 1 1 0.7
Polygon "P" [-0.5 0 -0.5  -0.5 0 0.5
             0.5 0 0.5  0.5 0 -0.5]
        "st" [0 0  0 1  1 1  1 0]
Color 1 0 0
Disk 0 0.25 360
```

Make yellow the current color.
Insert a 1 x 1 polygon.

Assign the texture coordinates.
Make red the current color.
Insert the disk.





```
WorldEnd
```

Conclude the description of the 3D scene.

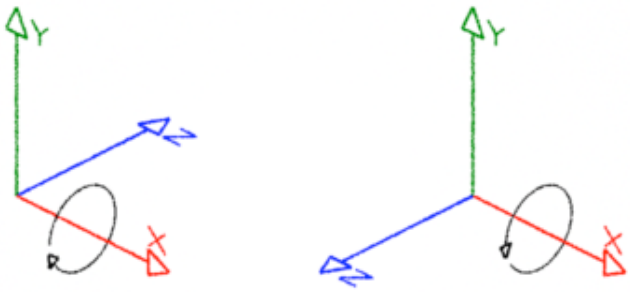
Rib
Left-hand & Right-hand Coordinate Systems

```
# disk_poly.rib
# left hand renderman coordinate system
# right hand world coordinate system

Display "disk_poly" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 320 240 1

Translate 0 0 3
Rotate -40  1 0 0
Rotate -20  0 1 0
Scale 1 1 -1  # <-- Note the negative z scale
WorldBegin
    Color 1 1 0.7
    Polygon "P" [-0.5 0 -0.5  -0.5 0 0.5  0.5 0 0.5  0.5 0 -0.5]
            "st" [0 0  0 1  1 1  1 0]
    Color 1 0 0
    Disk 0 0.25 360
WorldEnd
```

Coordinate systems come in two flavors, left-hand ("lh") and right-hand ("rh"). As shown below, the handedness of a coodinate system determines whether a positive rotation is clockwise or anti-clockwise. The thumb and fingers of either the left or right hand can be used to visualize the direction a positive rotation. For example, to decide whether a positive rotation around the x-axis of a right-hand coordinate system is clockwise or anti-clockwise point the thumb of the right hand along the x-axis; the "curl" of your fingers will indicate the direction of the rotation is clock-wise.



Left-hand (RenderMan) Right-hand (Maya & Houdini)

Rib Files Written by Pixar's RMS & RAT

Unlike RenderMan, most 3D modeling applications use right-hand coordinates.

Rib files generated by Maya, and Pixar's RenderMan Studio (RMS), apply a negative z scaling to the viewing transformations. The scaling ensures the coordinate system of the `WorldBegin` / `WorldEnd` block is right-handed. The predecessor to RMS, a product from Pixar known as RenderMan Artist Tools (RAT) also inserted a `ReverseOrientation` statement immediately after `WorldBegin`. This was necessary because RAT, despite the use of a right-hand world block, specified surface data (vertices of polygons etc) in left-hand order. The `ReverseOrientation` ensured that surface normals were correctly orientated. The Maya plugin for RMS, RenderMan for Maya Pro (RfM Pro) writes surface data in right-hand order and as such the call to `ReverseOrientation` is not required.

Rib Files Written by Side Effects Houdini

The "handedness" of surfaces written by Side Effects Houdini is similiar to RAT. However, instead of using `ReverseOrientation` their rib files use,

```
Orientation "rh"
```

immediately after WorldBegin to ensure that surface normals are correctly orientated.

Rib Files Written by Cutter

Rib files generated by Cutter as well as those listed in the Fundza tutorials conform to RMS in the sense that the world block is right-handed and vertices are listed in right-hand order.

Rib Files and Pixar's Technical Documentation

Traditionally, example rib files listed in Pixar's documentation specify a left-hand camera and a left-hand world block. In other words the negative z axis of the world is "pointing" at the camera. Whereas the rib files from RMS, RAT, Houdini and Cutter specify a right-hand world block in which the positive axis of the world is "pointing" at the camera.

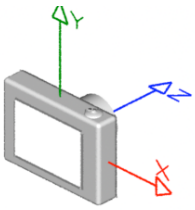
Issues with Surface Normals and Quadrics

While `ReverseOrientation` corrects the reversed normals in rib files written by RAT (actually, its Maya plugin called "mtor") it has an undesired side-effect on quardics and as such unfortunately causes them to become reversed! This is not an issue for Maya and Houndini because their tool kit of surfaces do not include quadrics. However, the unexpected flipping of the surface normals of quadrics is very significant when dealing with hand-written rib files that include the use of the `ReverseOrientation` statement. More information can be found about this issue in the tutorial "RenderMan: Reverse Orientation & Quadrics".

```
# disk_poly.rib                          Comments about the scene.
# left hand renderman coordinate system
# right hand world coordinate system
```

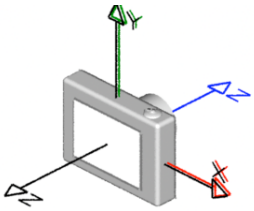
```
Display "disk_poly" "framebuffer" "rgb"
Projection "perspective" "fov" 40
Format 320 240 1
```

Set the camera to give a perspective view with a field of vision of 40 degrees and a frame size of 320 by 240 pixels.



The camera coordinate system is the current (active) system.

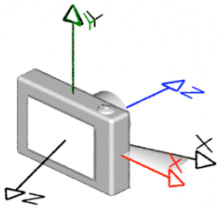
```
Translate 0 0 3
Rotate -40  1 0 0
Rotate -20  0 1 0
Scale 1 1 -1
```



The transformations are applied in reverse order.
First, the negative z-scaling.

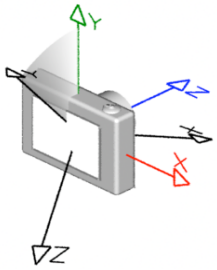
```
Translate 0 0 3
Rotate -40  1 0 0
Rotate -20  0 1 0
Scale 1 1 -1
```

Second, the negative rotation of 20 degrees around the y-axis is applied.



```
Translate 0 0 3
Rotate -40 1 0 0
Rotate -20  0 1 0
Scale 1 1 -1
```

Third, the negative rotation of 40 degrees around the x-axis is applied.

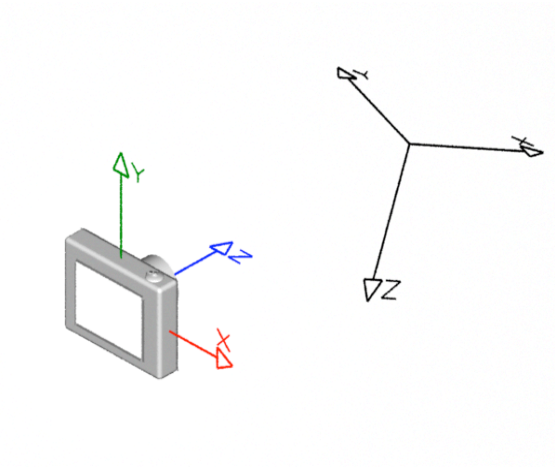


```
Translate 0 0 3
Rotate -40 1 0 0
```

Finally, the transformed coordinate

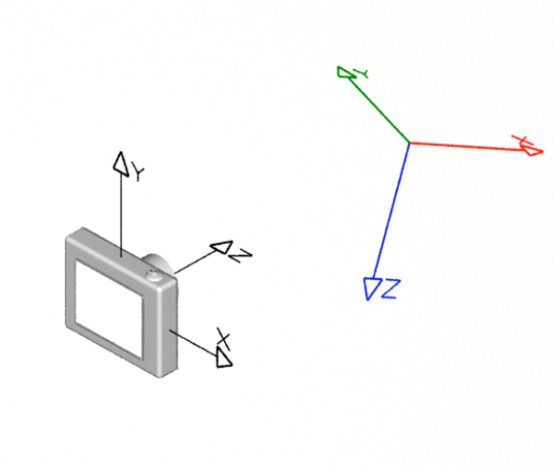
```
Rotate -20  0 1 0
Scale 1 1 -1
```

system is moved 3 units along the z-axis of the camera.




```
WorldBegin
```

The copy of the coordinate system now becomes the "world" coordinate system - also refered to as the current coordinate system.

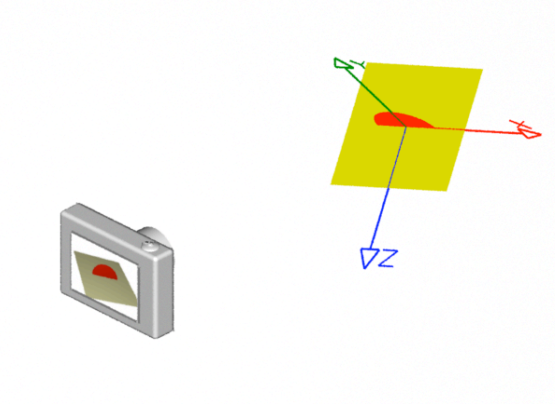



```
Color 1 1 0.7
Polygon "P" [-0.5 0 -0.5  -0.5 0 0.5
             0.5 0 0.5  0.5 0 -0.5]
        "st" [0 0  0 1  1 1  1 0]

Color 1 0 0
Disk 0 0.25 360
```

Make yellow the current color.
Insert a 1 x 1 polygon and specif its
'st' texture coordinates.

Make red the current color.
Insert the disk.

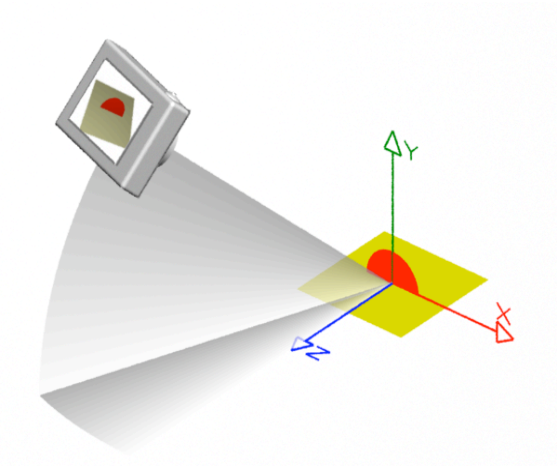



```
WorldEnd
```

Conclude the description of the 3D scene.

The illustrations shown above visualize the effects of each rib statement or group of statements. The visualizations have been made on the assumption the camera remains in a fixed upright position. Consequently, the contents of the 3D scene ie. the polygon and the disk specified in the world block, have been drawn in a tilted position. As shown below, the orientation of the camera and the world can be adjusted so that the y-axis of the world is seen in the more familiar upright position.

Whether you imagine the world is oriented relative to a fixed camera, or the camera is oriented relative to a fixed world is entirely up to you. What is important are the camera transformations that establish the **relative** relationship of the camera and the world.



Rib
Pre-baked RIB's

Introduction

Before attempting to produce a pre-baked RIB (ie. an archive) with Maya it is useful to gain an understanding of what archive files are and how they are referenced by other rib files. This tutorial assumes you are using the Cutter text editor.

What is an archive?

An archive, or pre-baked, RIB is similiar to a "regular" file except that it does not contain any camera statements or the statements `WorldBegin` & `WorldEnd`. Often an archive also omits shading and lighting statements as well. Generally, archive files only contain the data relating to the geometry of an object.

Figure 1 - original model

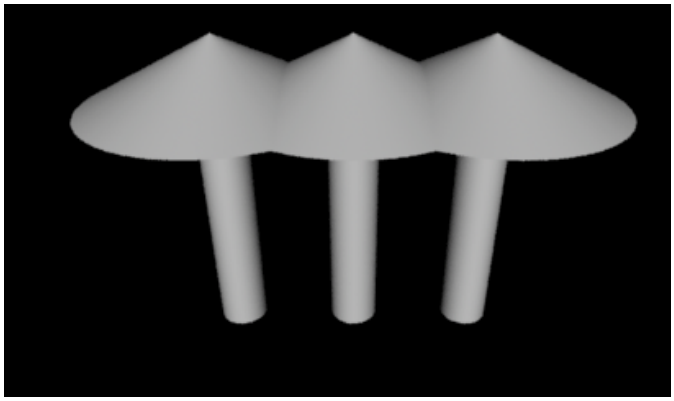


Figure 2 - archive read 3 times

For the purpose of this tutorial a regular RIB file has been converted into an archive as a result of removing all the text from the beginning of the RIB file down to the first occurance of the statement **TransformBegin**. The "tail" of the RIB file has also had its concluding **WorldEnd** statment removed. After inserting **AttributeBegin** at the head and **AttributeEnd** at the tail of the document we have effectively created an archive RIB file.

The original RIB and the converted archive can be viewed here and here. The RIB file [view here] that imports the archive does so using a ReadArchive statement.

Archive Rib File

```
AttributeBegin
    TransformBegin
        Rotate -90 1 0 0
        Cylinder 0.2 0 2 360
    TransformEnd
    TransformBegin
```

```
        Translate 0 2 0
        Rotate -90 1 0 0
        Cone 0.5 1 360
    TransformEnd
AttributeEnd
```

Although the archive (pre-baked) rib shown above is trivial, it does conform to the format of a correctly structured file. Apart from being much more complicated, archives generated by professional 3D applications such as Maya/RMS or Houdini follow the same format. They also include commented text at the head of their archive files.

Rib File Using an Archive

```
Display "untitled" "framebuffer" "rgb"
Format 427 240 1
Projection "perspective" "fov" 40
ShadingRate 1

LightSource "distantlight" 1 "intensity" 1.5
                    "from" [0 0 0] "to" [0 0 1]

Translate  0 -1 5
Rotate -30 1 0 0
Rotate 0   0 1 0
Scale 1 1 -1

WorldBegin
    Surface "plastic"
    TransformBegin
        Translate -1 0 0
        ReadArchive "archive.rib"
    TransformEnd
    TransformBegin
        Translate 0 0 0
        ReadArchive "archive.rib"
    TransformEnd
    TransformBegin
        Translate 1 0 0
        ReadArchive "archive.rib"
    TransformEnd
WorldEnd
```

Rib
Secondary Images - AOVs

Introduction

The output of a RenderMan beauty pass is a full colored image displayed in a window, or saved to an image file. In either case, "rgb" or "rgba" data is "piped" through the **primary display channel** using the rib `Display` statement.

```
Display "PATH/untitled.tif" "tiff" "rgba"
```

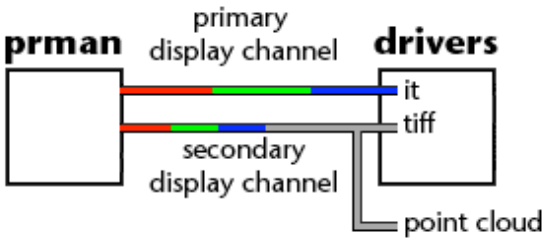


Figure 1

With prman 12.0 the `DisplayChannel` statement was introduced. It enables variables that store any RSL numeric data to be associated with their own "pipe" or so-called **secondary display channel**. For example, a secondary image that displays texture coordinate "`t`" data can be generated with the following two rib statements.

```
DisplayChannel "float t" "quantize" [0 0 0 0] "dither" [0]
Display "+PATH/untitled_t.tif" "tiff" "t"
```

The two `Display` statements shown above were used to generate the following images.

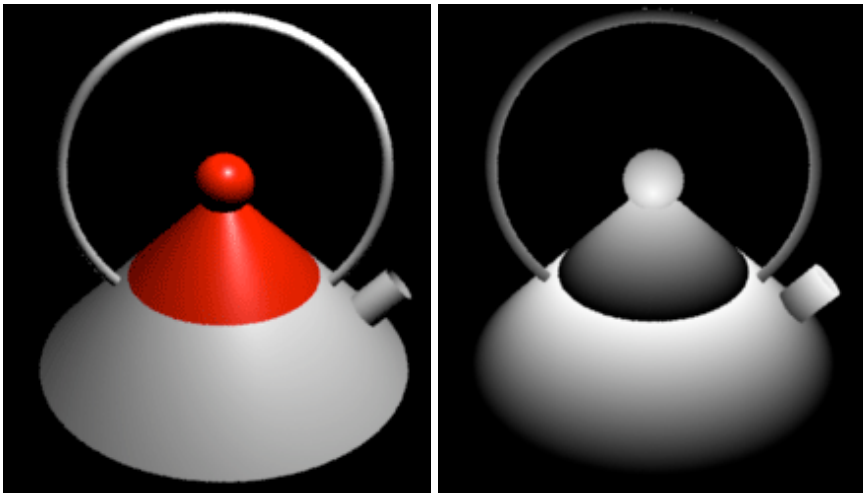


Figure 1

Rib files generated by Pixar's mtor plugin have 42 pre-defined `DisplayChannel` statements that enable 17 of the renderers primitive variables (P, N, s, t etc) and 32 shader output variables to be automatically referenced as sources of data for secondary images.

Mtor Predefined Display Channels

Primitive Variables	Shader Output Variables
color Ci	color Ambient
color Cs	color Backscattering
color Oi	color DiffuseColor
color Os	color DiffuseDirect
float s	color DiffuseDirectShadow
float t	color DiffuseEnvironment
float u	color DiffuseIndirect
float v	color Incandescence
normal N	color OcclusionDirect
normal Ng	color OcclusionIndirect
point P	color Refraction
vector E	color Rim
float du	color SpecularColor
float dv	color SpecularDirect
vector dPdtime	color SpecularDirectShadow
vector dPdu	color SpecularEnvironment
vector dPdv	color SpecularIndirect
	color Subsurface
	color Translucence
	color _Ci
	color _Oi
	color _albedo
	color _color
	color _diffusemeanfreepath
	color _indirectdiffuse
	color _radiance_t
	color _radiosity
	float __CPUtime
	float _area
	float _float
	float _occlusion
	vector _environmentdir

DisplayChannel's also enable data to be saved to a point cloud (.pct) file.

Quantization

Within the renderer, data such as floats, the rgb components of colors and the xyz values of points, vectors and normals are stored with **floating point precision**. When saving such data into an 8 bit per channel tiff file it is necessary to convert the floating point data into integer values - a process known as quantitization.

```
DisplayChannel "float t" "quantize" [0 0 0 0] "dither" [0]
Display "+PATH/untitled_t.tif" "tiff" "t"
        "quantize"  [0 255 0 255] "dither" [0.5]
```

Without any quantization the output image would be a 32bit/channel (4,294,967,296 values per channel) tiff file ie.

```
Display "+PATH/untitled_t.tif" "tiff" "t"
        "quantize"  [0 0 0 0] "dither" [0.0]
```

Or a 16bit/channel (65,536 values per channel) tiff file ie.

```
Display "+PATH/untitled_t.tif" "tiff" "t"
        "quantize"  [0 65535 0 65535] "dither" [0.0]
```

For an excellent explanation of quantization refer to page 41 of "Advanced RenderMan" by Tony Apodaca and Larry Gritz. Also refer to,

```
Using Arbitrary Output Variables in PhotoRealistic Renderman
 - prman_technical_rendering/AppNotes/appnote.24.html
```

Generating a secondary image that "encodes" a primitive variable is straight forward because such variables are inherently known to the renderer. This is not true for shader output variables or AOV's (arbitary output variables). For example, figure 2 shows a secondary image (render pass) that contains "_specular" data.

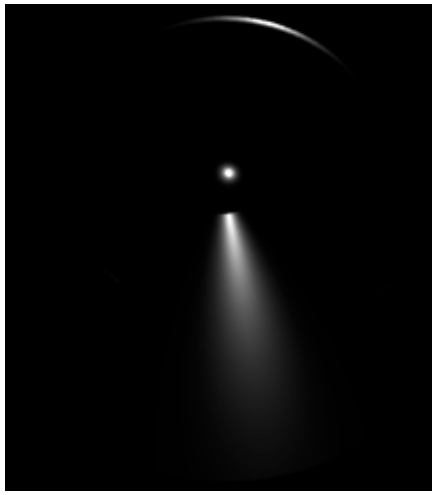


Figure 2

Only by rendering the teapot using a shader that assigns values to a variable that it declares as,

```
output varying color _specular = 0;
```

and that also specifies a `DisplayChannel` in the rib file as...

```
DisplayChannel "color _specular" "quantize" [0 0 0 0] "dither" [
```

can a secondary image (pass) be generated. Alternatively, if Pixar's Slim is used to

make a shading network an AOV can be outputted that way. Whether a hand coded Slim generated shader is used unless it declares an `output varying` variable, and course assigns values to the variable, can data be "piped" through a secondary display channel. For example, it would be futile to attempt to output a secondary imag containing specular data using the classic "plastic" shader because it does not assig specular values to an output variable. The shader shown in listing 2 outputs specular data through an AOV named _specular.

Listing 2

```
surface
spec_out(float  Ks = 0.7,
                roughness = 0.1;
         color  hilitecolor = 1;
output varying color _specular = 0)
{
normal  n = normalize(N);
normal  nf = faceforward(n, I);

Oi = Os;

vector  i = normalize(-I);
_specular = Ks * specular(nf, i, roughness)
               * hilitecolor;
Ci = Oi * Cs * _specular;
}
```

The shader shown in listing 3 uses an output varying variable to "pipe" data into a point cloud.

Listing 3

```
surface
bake_out(float  Ks = 0.7,
                roughness = 0.1;
         color  hilitecolor = 1;
        string  channel = "",
                bakefile = "")
{
normal  n = normalize(N);
normal  nf = faceforward(n, I);
Oi = Os;
vector  i = normalize(-I);
color spec = Ks * specular(nf, i, roughness)
                * hilitecolor;
if(bakefile != "")
    bake3d(bakefile, channel, P, n, "_specular", spec);
Ci = Oi * Cs * spec;
}
```

A rib file that uses the "bake_out" shader might do so in this way.

```
Surface "bake_out" "channel" ["_specular"]
     "bakefile" ["spec.ptc"]
```

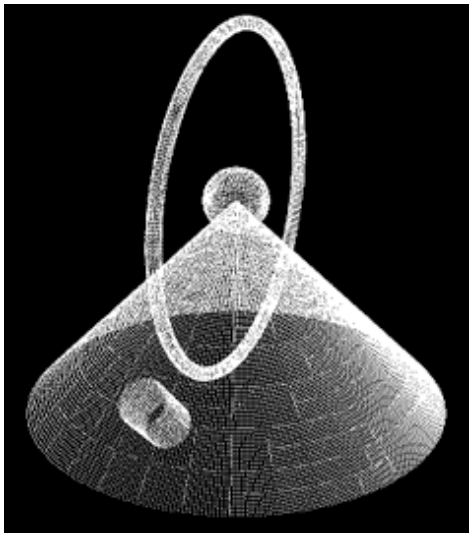


Figure 3
Specular data stored in a point cloud

Rib
Stereo Rendering & Anaglyphs

Note:

This tutorial is in an early stage of development. The tutorial assumes the reader has access to prman version 13.5 and higher. For convenience the reader will also require access to Shake.
MK Jan 2008.

```
# output the alpha channel to camera_right.tif
DisplayChannel "float a"
# ditto surface color
DisplayChannel "color Ci"

Projection "perspective" "fov" [40]
Format 640 480 1
ShadingRate 5

# Right camera viewing transformations
TransformBegin
    Translate  0 0 20
    Rotate  -10  1 0 0
    Rotate  0  0 1 0
    Scale 1 1 -1
    Camera "right"
TransformEnd
# activate this line for on-screen viewing
#Display "camera_left.tif" "it" "rgba"

# Save left and right camera images to file
Display "camera_left.tif" "tiff" "rgba"
Display "+camera_right.tif" "tiff" "Ci,a"
        "quantize" [0 255 0 255] "string camera" ["right"]

# Left camera viewing transforms
Translate  -0.05 0 20
Rotate  -10  1 0 0
Rotate  0  0 1 0
Scale 1 1 -1

WorldBegin
    TransformBegin
        Translate -3.5 0 0
        ReadArchive "rotated_cylinders.rib"
    TransformEnd
    TransformBegin
        Translate 3.5 0 0
```

```
        Rotate 45  0 0 1
        Rotate 180 0 1 0
        ReadArchive "rotated_cylinders.rib"
    TransformEnd
WorldEnd
```

Introduction

With release 13.5 of Pixars prman the renderer can now render two camera simultaneously. The primary documentation about this facility can be found at,

```
Pixar_docs/prman_technical_rendering/AppNotes/multiCamera.html
```

Pixars documentation addresses several technical issues related to **view-dependent** shading. This tutorial ignores the impact that stereo rendering has on shader writing. Instead this tutorial concentrates on the production of "cheap and cheerful" colored images suitable for viewing with (old fashioned) red and cyan spectacles. The tutorial is intended to help a reader experiment with stereo rendering without having to bother with the paraphenalia of stereo projectors, or high performance flat panel displays and "passive" spectacles, or "active" (switched) spectacles.

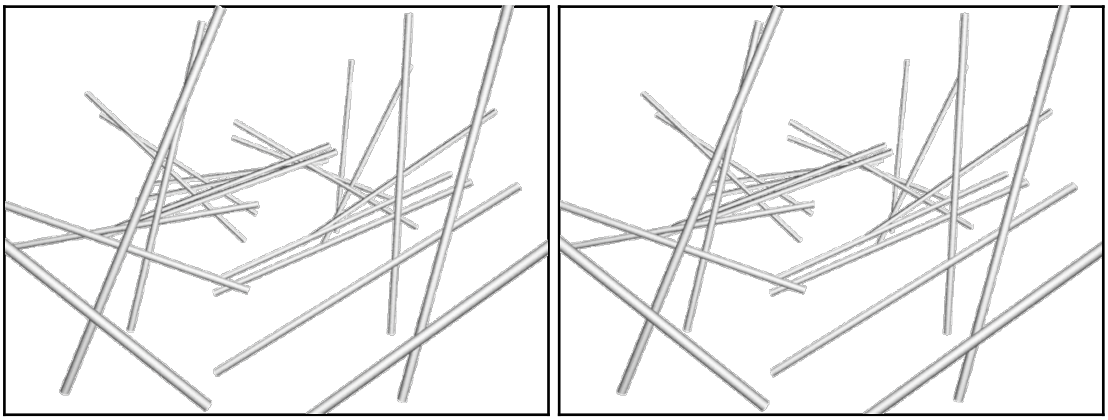


Figure 1 - Left and Right Stereo Images

The rib file shown above will act as a template for two-camera rendering. It assumes the scene to be rendered can be "imported" using the `ReadAchive` statement. As can be seen from figure 1 the (test) scene consists of 20 rotated cylinders. Two other assumptions are also made, namely,

1. the left and right cameras will remain parallel,
2. the fixed camera, defined by the following statements,

```
TransformBegin
    Translate  0 0 20
    Rotate  -10  1 0 0
    Rotate  0  0 1 0
    Scale 1 1 -1
    Camera "right"
TransformEnd
```

will always define the "right" camera. Consequently, adjustments to the separation between the cameras will be made to the x-translation of the principle (left) camera. For example, the current separation is -0.05 units.

```
Translate  -0.05 0 20
```

In all other respects it is up to the reader to ensure, when they edit the rib file, the transforms applied to the left and right camera remain the same.

Converting Stereo Images to an Anaglyph

Listing 2 presents a Shake script that colorizes and combines two rendered tif files into a single red/cyan anaglyph - figure 2.

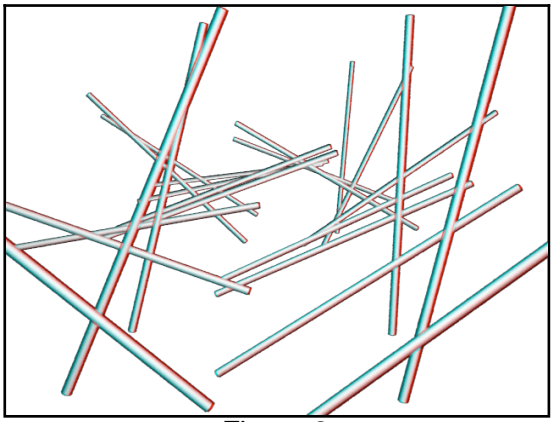


Figure 2

For a description of how to do the conversion using PhotoShop refer to "Mark Newbold's Stereo 3D Stuff" at,

`http://dogfeathers.com/3d/3dhowto.html`

Listing 2 (anaglyph.shk)

```
image left_src = FileIn("camera_left.tif");
image left_no_red = Mult(left_src, 0, 1, 1, 1, 1);

image right_src = FileIn("camera_right.tif");
image right_red_only = Mult(right_src, 1, 0, 0, 1, 1);

image final = IAdd(left_no_red, right_red_only);
FileOut(final, "output_name.png");
```

The Shake script produces an output image that is suitable for viewing on a web page - hence the "png" format, although "jpg" could also be specified. It is possible to automate the process of rendering, image conversion and viewing by using two `System` statements after the `WorldEnd`. For example, the first `System` statement should invoke Shake,

```
System "shake -script ./anaglyph.shk"
```

When using Cutter, the rendered images produced by the rib file will be saved by `prman` in Cutter's directory. A relative path to "anaglyph.shk" can be used as long as the Shake script is also saved in the Cutter directory. Listing 3 gives the code for a simple web page that can be used to view an anaglyph. Again it is assumed the html file will be saved in the same directory as Cutter.

Listing 3 (camera_viewer.html)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
    "http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
<TITLE>3D Viewer</TITLE>
<LINK rel=StyleSheet href="RELATIVE_PATH.css"
    TYPE="text/css" TITLE="YOUR_STYLE_NAME">

</HEAD>
<BODY BGCOLOR="#666666">
<BR>
<img src="image_name.png" alt="" border="1">
</BODY>
</HTML>
```

The text for the second `System` statement depends on the readers OS. For example,

Linux/Mozilla

```
System "mozilla ./camera_viewer.html"
```

Windows/Mozilla

```
System "ixplore ./camera_viewer.html"
```

OSX/Safari

```
System "open -a /Applications/Safari.app ./camera_viewer.html"
```

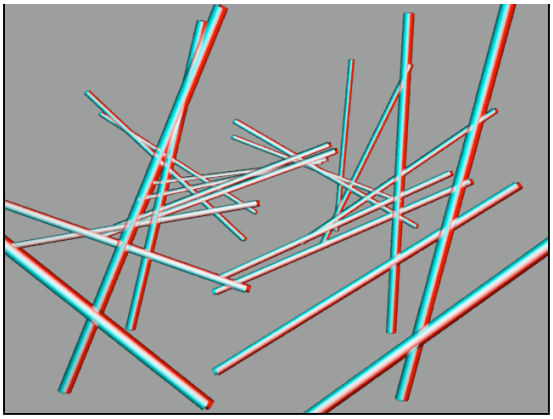
Camera Separation

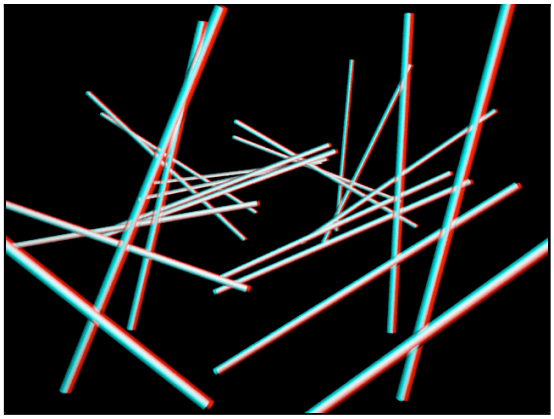
Deciding on an appropriae value for the separation of the left and right cameras is a subject for experimentation. Factors that contribute to the choice of camera separation are,

- camera to world distance
- camera "fov"
- image Size
- image background color

For example, figures 3, 4, 5 and 6 were all rendered with a camera separation of 0.1 units yet the perception of depth is very different

depending on image size and background color.





Figures 3 and 4 - 320x240 pixels

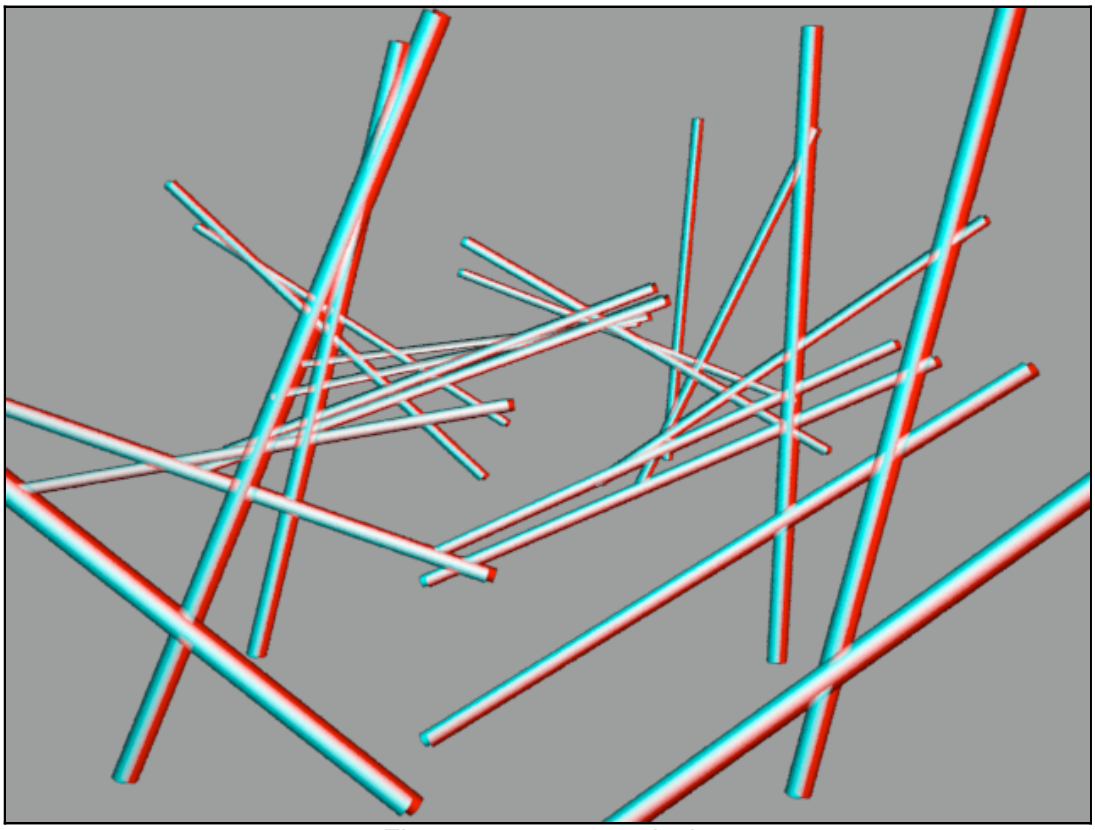
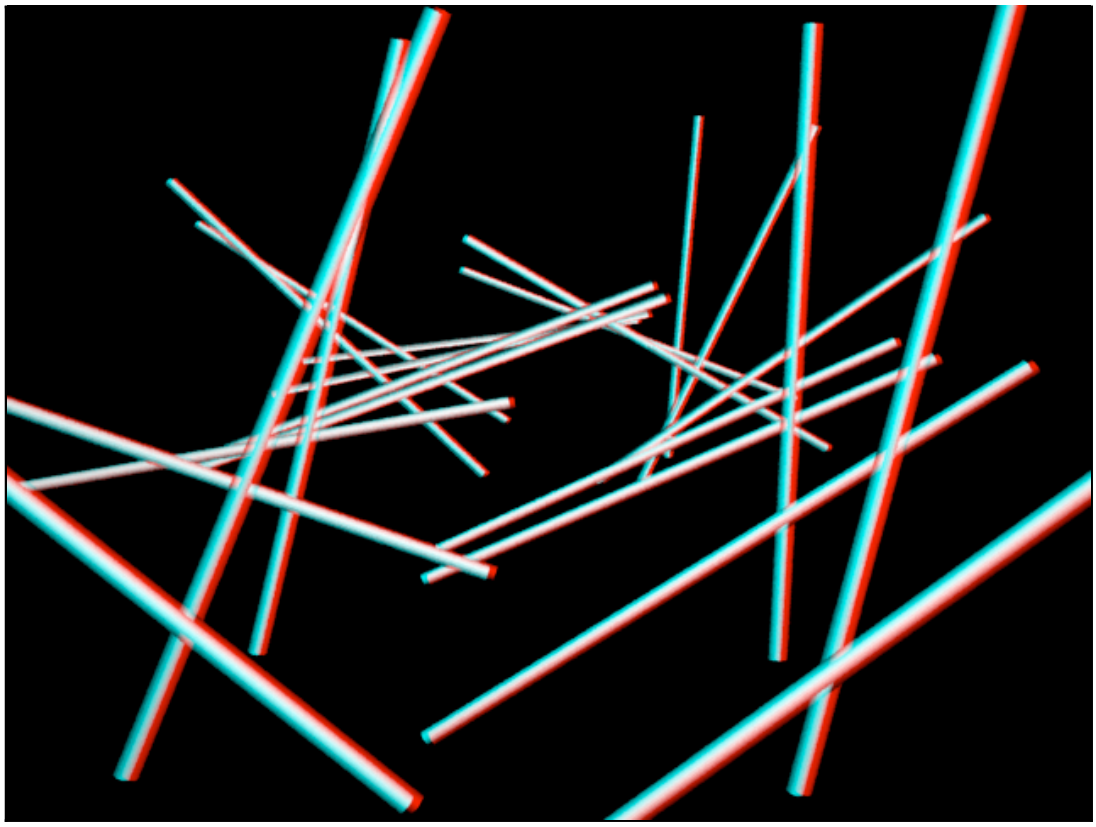


Figure 5 - 640x480 pixels



Figures 6 - 640x480 pixels

Colored Glasses

Good photography stores will sell colored "lighting filters" or "gels" - sheets of colored acetate. `Bogan Imaging Inc`. manufactures such filters in many different colors. Unfortunately, they do not produce a cyan filter. The anaglyphs produced by the Shake script (listing 2) for the purposes of this tutorial were viewed using primary red and green filters. The filters worked quite well. There are, however, several companies that advertise on the internet that sell red/cyan glasses designed specifically for viewing anaglyphs.

Rib
Curve Basics

References

"Nurb Curves: A Guide for the Uninitiated"
devworld.apple.com/dev/techsupport/develop/issue25/schneider.html

Introduction

This tutorial covers the basic issues of dealing with the RenderMan's curve primitive. Figure 1 shows four colored `Curves`, that have been rendered using different "curve types" but sharing identical control vertices (cv's).

b-spline
bezier
catmull-rom
hermite

The cv's are colored white and gray connecting lines show the sequence of cv's.

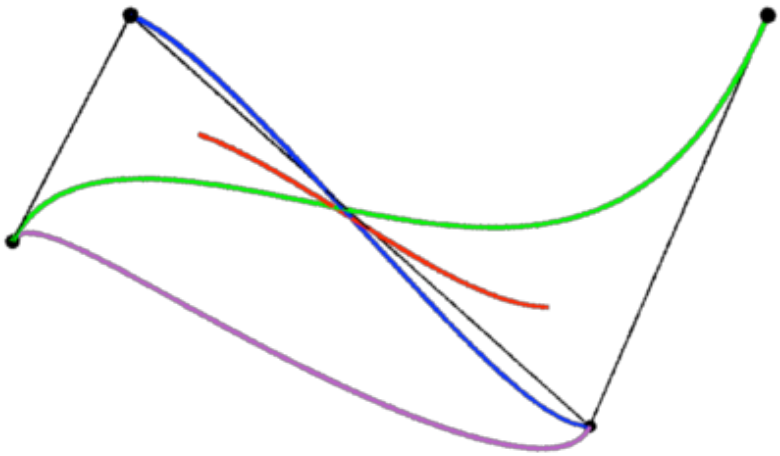


Figure 1

There is a clear difference between each of the curves. In particular, notice the green **bezier** curve is the only one to begin and end at the first and last cv. The RIB statements used to render the bezier curve are,

```
Basis "bezier" 3 "bezier" 3
Curves "cubic" [4] "nonperiodic"
                   "P" [-0.75 0 0.5  -0.45 0 1  0.5 0 0  0.75 0 1]
                   "constantwidth" [0.005]
                   "Cs" [0 1 0   0 1 0]
```

The rib file used to render this image can be viewed here. Curves generated by

Maya and mtor/Rfm the curve type is "b-spline" ie.

```
Basis "b-spline" 1 "b-spline" 1
```

Refer to rendering Maya curves with prman.

Assumptions

Although a RenderMan curve might look as if it can be defined by an arbitary number of cv's - in the example shown above the "`P`" list contains four lots of xyz's. Infact, the number of cv's must conform to formula determined by the type of curve being rendered. The simpliest curve type to use is a "b-spline" because it can be defined by any number of cv's - greater than 4. Unfortunately, in general a "b-spline" curve does not begin and end at the first and last cv.

A "bezier" curve does have the desirable property of starting and finishing at the first and last cv. However, the number of cv's, less 1, must be exactly divisible by 3. For instance, we might use 4, 7, 10 or 13 cv's for a "bezier" curve. The only time when the restriction on the number of cv's can be ignored is when a "periodic" curve is produced, in which case, the end of the curve wraps around to coincide with the beginning of the curve. A single RenderMan `Curves` statement can define **multiple** separate curves.

RenderMan Procedural Primitives
Implementations in Python, Tcl and 'C'

Introduction

A helper app is an executible that is loaded by a RenderMan complient renderer as a result of reading a `Procedural "RunProgram"` statement from a rib file or a rib stream. For example,

```
Procedural "RunProgram"
           ["H:/rman/helpers/hairball" "2.0  200  0.03"]
           [-2 2 -2 2 -2 2]
```

In this example, the call to `Procedural` informs the renderer that it should run a program called hairball.exe located in the "H:/rman/helpers/" directory. The three parameters,

```
"2.0 200 0.03"
```

are values that are passed to the program. The six values,

```
"-2 2 -2 2 -2 2"
```

define the size of the bounding box of the geometry the helper app will create. When the renderer calls a helper app it not only passes the parameter values, it also calculates the number of pixels the bounding box of the object will cover in the rendered image. This value can help the app make **level of detail** decisions ie. should it generate a low, medium or a hi-res version of the geometry?

This tutorial provides the reader with simple example of the implementation and use of a helper app. Three implementations are given. Listing 2 is in the 'C' programming language, listing 3 is in Python, and listing 4 is in Tcl. All three implementations create the same geometry, a single quadric sphere, and all three implementations expect two parameters. The first, pixel coverage, is ignored. The second parameter specifies the radius of the sphere. It will be assumed the binary or script for each of the example implementations is located at one of the following locations,

```
H:/rman/helpers/demo <.c .py .tcl>
/home/$USER/rman/helpers/demo <.c .py .tcl>
/Users/$USER/Documents/rman/helpers/demo <.c .py .tcl>
```

Basic Code

The following template code is based on the web notes, "Writing Procedural Primitives with RenderDotC"

```
www.dotcsw.com/doc/procedurals.html#runprogram
```

Other sources of information about helper apps can be found by searching the RenderMan documents that accompany Pixars prman. Pages 119/120 of "Advanced RenderMan" by Tony Apodaca and Larry Gritz is also an excellent source of information about this topic. Listing 1 is a simple rib file that can be used to run each of the helper apps provided in listings 2, 3 and 4.

Listing 1 (rib)

```
Display "untitled" "framebuffer" "rgba"
Format 400 400 1
Projection "perspective" "fov" 30
ShadingRate 5
# Head light
LightSource "distantlight" 1 "intensity" 1.5
            "from" [0 0 0] "to" [0 0 1]

Translate  0 0 5
Rotate -30 1 0 0
Rotate 20   0 1 0
Scale 1 1 -1
WorldBegin
    TransformBegin
        Surface "plastic"
        # Procedural "RunProgram" goes next.
        # Refer to the notes on each helper app
        # implementation for examples of the correct syntax.

    TransformEnd
WorldEnd
```

Python Implementation

Listing 2 (demo.py)

```
import sys

args = sys.stdin.readline()
while args:
    arg = args.split()
    pixels = float(arg[0])
    rad = float(arg[1])
    print 'TransformBegin'
    print 'Sphere %s %s %s 360' % (rad, -rad, rad)
    print 'TransformEnd'
    sys.stdout.write('\377')
    sys.stdout.flush()
    # read the next set of inputs
    args = sys.stdin.readline()
```

In the rib file the script would be invoked as follows,

```
# Windows
Procedural "RunProgram"
["python H:/rman/helpers/demo.py" "1"]
[-1 1 -1 1 -1 1]
# Linux
Procedural "RunProgram"
["/usr/bin/python /home/$USER/rman/helpers/demo.py" "1"]
[-1 1 -1 1 -1 1]
# MocOSX
Procedural "RunProgram"
["/usr/bin/python /Users/$USER/Documents/rman/helpers/demo.py" "1"]
[-1 1 -1 1 -1 1]
```

Tcl Implementation

Listing 3 (demo.tcl)

```
fconfigure stdout -translation binary

while { [gets stdin args] != -1 } {
    set pixels [lindex $args 0]
    set rad [lindex $args 1]

    puts "TransformBegin"
    puts "Sphere $rad -$rad $rad 360"
    puts "TransformEnd"

    puts "\377"
    flush stdout
    }
```

In the rib file the script would be invoked as follows,

```
# Windows
Procedural "RunProgram"
["tclsh H:/rman/helpers/demo.tcl" "1"]
[-1 1 -1 1 -1 1]
# Linux
Procedural "RunProgram"
["/usr/bin/tclsh /home/$USER/rman/helpers/demo.tcl" "1"]
[-1 1 -1 1 -1 1]
# MocOSX
Procedural "RunProgram"
["/usr/bin/tclsh /Users/$USER/Documents/rman/helpers/demo.tcl" "1"]
[-1 1 -1 1 -1 1]
```

'C' Language Implementation

The code in listing 2 will build a helper app called **demo**.

Listing 4 (demo.c)

```
#include <stdio.h>

void main()
{
float pixels, rad;
char  args[256];

while(gets(args))
    {
    sscanf(args, "%f %f", &pixels, &rad);
    printf("Sphere %f %f %f 360\n", rad, -rad, rad);
    printf("%c", '\377');
    fflush(stdout);
    }
}
```

In the rib file this app would be invoked as follows,

```
# Windows
Procedural "RunProgram"
["H:/rman/helpers/demo" "1"]
[-1 1 -1 1 -1 1]
# Linux
Procedural "RunProgram"
["/home/$USER/rman/helpers/demo" "1"]
[-1 1 -1 1 -1 1]
# MocOSX
Procedural "RunProgram"
["/Users/$USER/Documents/rman/helpers/demo" "1"]
[-1 1 -1 1 -1 1]
```

How it Works

Although it is not apparent from the code, the demo app automatically has access to three **streams** by which it can receive and send data, namely, **stdin**, **stdout** and **stderr**. Ordinarily, the first stream is connected to the keyboard, the others are connected to the console.

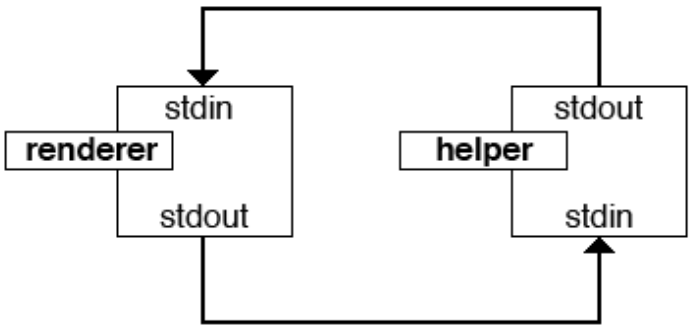


Figure 1

When the renderer invokes a helper app it **redirects** its own the stdin and stdout streams to those of the helper app so that, for example, input and output from

functions/procs such as `printf`(), `print` and `puts` no longer use the console but instead "feed" data to the renderer.

RenderMan Procedural Primitives
RiPoints on a Sphere

Introduction

This tutorial follows on from the tutorial "Procedural Primitives: Basics". Procedural primitives, or helper apps, are ideal when complex surfaces can be defined procedurally. A procedural primitive can be loosely described as a shape made from geometry that has been assembled according to the application of one or more rules. This tutorial introduces a simple procedural primitive made from light-weight ie. fast to render, RenderMan points. The spherical shell shown in figure 1 consists of 5000 points.

Using a Intel MacOSX 667 MHz the following timings were obtained,
- 1,000,000 points generated in 1 min 12 seconds
- 100,000 points generated in 7 seconds
- 10,000 points generated in 0 seconds!

Figure 1
10,000 random points generated by ripoint.py

Figure 2
10,000 random points generated by ripoint.tcl

The rib statement that requests the renderer to produce, say, three points of uniform radius, colored red, green and blue is,

```
Points "P" [-0.5 0 0   0 0 0   0.5 0 0]
       "constantwidth" [0.01]
       "Cs" [1 0 0   0 1 0   0 0 1]
```

There can be any number of xyz's following the "`P`" parameter, each triplet specifies the position of a point. If the points are to have a specific color there must be as many rgb color values following the "`Cs`" parameter as there are xyz's in the

"P" list.

Spherical Shell of Points

The method for producing a spherical shell of randomly placed colored points is,

1. generate a random vector in a unit cube
2. normalize the vector
3. scale the vector by "radius"
4. use the vectors components to locate a point in space
5. generate a random color

By repeating these steps we obtain a spherical shell of points. The cloud proc uses three functions that were developed in other tutorials,

Rib File for Testing

Listing 1 is a rib file that can be used to test both the python and the Tcl implementations of the helper app. The rib file is setup for use on MacOSX. Refer to the previous tutorial for examples of how python and Tcl helper apps should be called when using Windows and Linux.

Listing 1 (ripoints.rib)

```
#Option "statistics" "endofframe" [1]
Display "shader_tester" "it" "rgba"
Format 250 250 1
Projection "perspective" "fov" 40
ShadingRate 1

Translate  0 0 6
Rotate 0   1 0 0
Rotate 0   0 1 0
Scale 1 1 -1
WorldBegin
    AttributeBegin
        Surface "constant"
        Procedural "RunProgram"
        ["/usr/bin/python FULL_PATH/ripoints.py" "2 10000 0.02"]
        [-2 2 -2 2 -2 2]
        #Procedural "RunProgram"
        #["/usr/bin/tclsh FULL_PATH/ripoints.tcl" "2 10000 0.02"]
        #[-2 2 -2 2 -2 2]
    AttributeEnd
WorldEnd
```

Python Implementation

Listing 2 (ripoints.py)

```
import sys, math, random
```

```
random.seed(5)
def randBetween(min, max):
    return random.random() * (max - min) + min
def length(x, y, z):
    return math.sqrt(x*x + y*y + z*z)
def normalize(x, y, z):
    len = length(x, y, z)
    return x/len, y/len, z/len
def scaleVector(x, y, z, sc):
    return x*sc, y*sc, z*sc
def cloud(radius, num, width):
    print 'Points \"P\" ['
    for n in range(num):
        x = random.random() * 2 - 1;
        y = random.random() * 2 - 1;
        z = random.random() * 2 - 1;
        x,y,z = normalize(x, y, z)
        x,y,z = scaleVector(x, y, z, radius)
        print '%s %s %s' % (x, y, z)
    print '] \"constantwidth\" [%s]' % width
    print '\"Cs\" ['
    for n in range(num):
        r = randBetween(0, 1)
        g = randBetween(0, 1)
        b = randBetween(0, 1)
        print '%s %s %s' % (r, g, b)
    print ']'

def main():
    args = sys.stdin.readline()
    while args:
        arg = args.split()
        pixels = float(arg[0])
        rad = float(arg[1])
        num = int(arg[2])
        width = float(arg[3])

        print 'TransformBegin'
        cloud(rad, num, width)
        print 'TransformEnd'
        sys.stdout.write('\377')
        sys.stdout.flush()
        # read the next set of inputs
        args = sys.stdin.readline()

if __name__ == "__main__":
    main()
```

Tcl Implementation

Listing 3 (ripoints.tcl)

```
fconfigure stdout -translation binary

proc randBetween { min max } {
    return [expr rand() * ($max - $min) + $min]
    }
proc length { x y z } {
    return [expr sqrt($x*$x + $y*$y + $z*$z)]
    }
proc normalize { x y z } {
    set len [length $x $y $z]
    return [list [expr $x/$len] [expr $y/$len] [expr $z/$len]]
    }
proc scaleVector { vect sc } {
    set X [expr [lindex $vect 0] * $sc]
    set Y [expr [lindex $vect 1] * $sc]
    set Z [expr [lindex $vect 2] * $sc]
    return [list $X $Y $Z]
    }
proc cloud { radius num width } {
    puts "Points \"P\" \["
    for {set n 0} {$n < $num} {incr n} {
        set x [expr rand() * 2 - 1]
        set y [expr rand() * 2 - 1]
        set z [expr rand() * 2 - 1]
        set vec [normalize $x $y $z]
        set vec [scaleVector $vec $radius]
        puts "[lindex $vec 0] [lindex $vec 1] [lindex $vec 2]"
        }
    puts "\] \"constantwidth\" \[$width\]"
    puts "\"Cs\" \["
    for {set n 0} {$n < $num} {incr n} {
        set r [randBetween 0 1]
        set g [randBetween 0 1]
        set b [randBetween 0 1]
        puts "$r $g $b "
        }
    puts "\]"
    }

while { [gets stdin args] != -1 } {
    set pixels [lindex $args 0]
    set rad [lindex $args 1]
    set num [lindex $args 2]
    set width [lindex $args 3]

    puts "AttributeBegin"
    cloud $rad $num $width
    puts "AttributeEnd"
    puts "\377"
    flush stdout
    }
```

Animation

If radius of the cloud is increased over several frames it would create the illusion of a fireworks explosion. The ripoints scripts implemented in this tutorial could be instanced by each particle in a Maya or Houdini particle system. When viewed within the modeler, such a particle system might look relatively unimpressive, however, the final particle animation would appear to be very complex.

RenderMan Procedural Primitives
Randomness

Introduction

Being able to generate points distributed randomly within or over a surface can be useful when modeling phenomena such as fireworks. This tutorial presents in listings 1 to 9 some useful utility procs implemented in Python, Tcl and the 'C' programming language. The implementations of the utility procs will be given followed by examples of their use.

Proc randBetween

This proc returns a random value between two input values. Although it is a very simple it is surprisingly useful for positioning objects, such as RenderMan points and curves, as well as setting randomized rgb components of colors.

Listing 1 - Python Implementation

```
import random

def randBetween(min, max):
    return random.random() * (max - min) + min
```

Listing 2 - Tcl Implementation

```
proc randBetween { min max } {
    return [expr rand() * ($max - $min) + $min]
    }
```

Listing 3 - 'C' Implementation

```
#include <stdlib.h>

double randBetween(double min, double max)
{
return ((double)rand()/RAND_MAX) * (max - min) + min;
}
```

Procs length & normalize

The proc `length` returns the length of a vector. The proc `normalize` normalizes a vector. Typically, the input values to this proc are the xyz position of a geometric

point. However, the location of the geometric point can be considered to represent the "head" of a vector and as such it can be converted to a unit vector.

Listing 4 - Python Implementation

```
import math

def length(x, y, z):
    return math.sqrt(x*x + y*y + z*z)

def normalize(x, y, z):
    len = length(x, y, z)
    return x/len, y/len, z/len
```

Listing 5 - Tcl Implementation

```
proc length { x y z } {
    return [expr sqrt($x*$x + $y*$y + $z*$z)]
    }

proc normalize { x y z } {
    set len [length $x $y $z]
    return [list [expr $x/$len] [expr $y/$len] [expr $z/$len]]
    }
```

Listing 6 - 'C' Implementation

```
#include <stdlib.h>
#include <math.h>

double length(double pnt[3])
{
return sqrt((pnt[0] * pnt[0]) +
            (pnt[1] * pnt[1]) +
            (pnt[2] * pnt[2]));
}

void normalize(double pnt[3])
{
double len = length(pnt);
pnt[0] /= len;
pnt[1] /= len;
pnt[2] /= len;
}
```

Proc scaleVector

This proc returns the xyz values of a vector re-sized to a specified length.

Listing 7 - Python Implementation

```
def scaleVector(x, y, z, sc):
    return x*sc, y*sc, z*sc
```

Listing 8 - Tcl Implementation

```
proc scaleVector { vect sc } {
    set X [expr [lindex $vect 0] * $sc]
    set Y [expr [lindex $vect 1] * $sc]
    set Z [expr [lindex $vect 2] * $sc]
    return [list $X $Y $Z]
    }
```

Listing 9 - 'C' Implementation

```
void scaleVector(double pnt[3], double sc)
{
pnt[0] *= sc;
pnt[1] *= sc;
pnt[2] *= sc;
}
```

Examples of Use

This section provides some simple examples of how the procs given in listing 1 to 9 can be used with RenderMan's point primitive. For brevity, the examples are only given in Python.

RiPoints in a Rectangular Volume

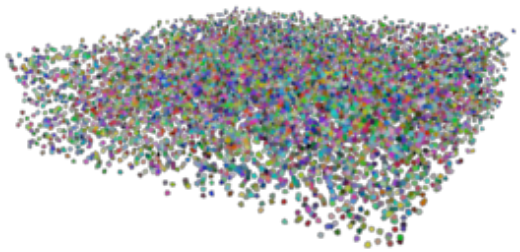


Figure 1

Listing 10 - rectangular box

```
import sys, math, random

def box(width, height, depth, num, size):
```

```
    print 'Points \"P\" ['
    for n in range(num):
        x = randBetween(-width/2, width/2)
        y = randBetween(-height/2, height/2)
        z = randBetween(-depth/2, depth/2)
        print '%s %s %s' % (x, y, z)
    print '] \"constantwidth\" [%s]' % size
    print '\"Cs\" ['
    for n in range(num):
        r = randBetween(0, 1)
        g = randBetween(0, 1)
        b = randBetween(0, 1)
        print '%s %s %s' % (r, g, b)
    print ']'

def main():
    args = sys.stdin.readline()
    while args:
        arg = args.split()
        pixels = float(arg[0])
        width = float(arg[1])
        height = float(arg[2])
        depth = float(arg[3])
        num = int(arg[4])
        size = float(arg[5])

        print 'TransformBegin'
        box(width, height, depth, num, size)
        print 'TransformEnd'
        sys.stdout.write('\377')
        sys.stdout.flush()
        # read the next set of inputs
        args = sys.stdin.readline()

if __name__ == "__main__":
    main()
```

RiPoints in a Ring

Figure 2

Listing 11 - Ring

```
import sys, math, random

def ring(rad, num, size):
    print 'Points \"P\" ['
    for n in range(num):
        x = randBetween(-1, 1)
        y = 0
        z = randBetween(-1, 1)
        x,y,z = normalize(x,y,z)
        x,y,z = scaleVector(x,y,z,rad)
        print '%s %s %s' % (x, y, z)
    print '] \"constantwidth\" [%s]' % size
    print '\"Cs\" ['
    for n in range(num):
        r = randBetween(0, 1)
        g = randBetween(0, 1)
        b = randBetween(0, 1)
        print '%s %s %s' % (r, g, b)
    print ']'

def main():
    args = sys.stdin.readline()
    while args:
        arg = args.split()
        pixels = float(arg[0])
        rad = float(arg[1])
        num = int(arg[2])
        size = float(arg[3])

        print 'TransformBegin'
        ring(rad, num, size)
        print 'TransformEnd'
        sys.stdout.write('\377')
        sys.stdout.flush()
        # read the next set of inputs
        args = sys.stdin.readline()

if __name__ == "__main__":
    main()
```

RiPoints on a Disk

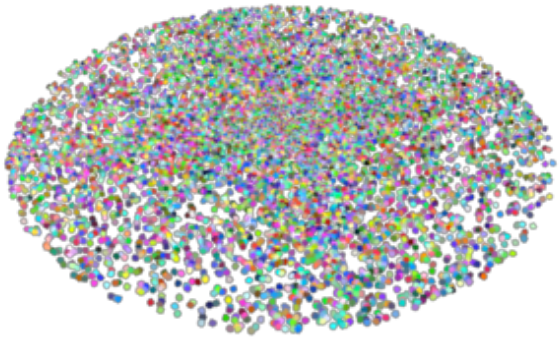


Figure 3

Listing 12 - Disk

```
import sys, math, random

def disk(rad, num, size):
    print 'Points \"P\" ['
    for n in range(num):
        x = randBetween(-1, 1)
        y = 0
        z = randBetween(-1, 1)
        x,y,z = normalize(x,y,z)
        randRadius = randBetween(0, rad)
        x,y,z = scaleVector(x,y,z,randRadius)
        print '%s %s %s' % (x, y, z)
    print '] \"constantwidth\" [%s]' % size
    print '\"Cs\" ['
    for n in range(num):
        r = randBetween(0, 1)
        g = randBetween(0, 1)
        b = randBetween(0, 1)
        print '%s %s %s' % (r, g, b)
    print ']'

def main():
    args = sys.stdin.readline()
    while args:
        arg = args.split()
        pixels = float(arg[0])
        rad = float(arg[1])
        num = int(arg[2])
        size = float(arg[3])

        print 'TransformBegin'
        disk(rad, num, size)
        print 'TransformEnd'
        sys.stdout.write('\377')
        sys.stdout.flush()
        # read the next set of inputs
        args = sys.stdin.readline()

if __name__ == "__main__":
```

```
    main()
```

RiPoints on a Cone

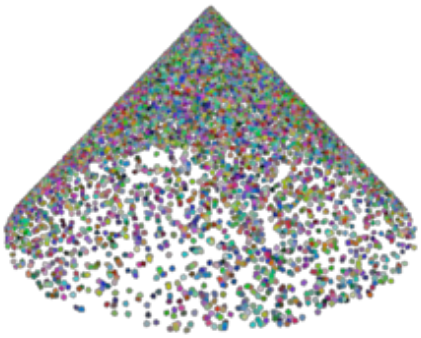


Figure 4

Listing 13 - Cone

```
import sys, math, random

def cone(rad, num, size):
    print 'Points \"P\" ['
    for n in range(num):
        x = randBetween(-1, 1)
        y = 0
        z = randBetween(-1, 1)
        x,y,z = normalize(x,y,z)
        randRadius = randBetween(0, rad)
        x,y,z = scaleVector(x,y,z,randRadius)
        y += 1 - randRadius # <<---
        print '%s %s %s' % (x, y, z)
    print '] \"constantwidth\" [%s]' % size
    print '\"Cs\" ['
    for n in range(num):
        r = randBetween(0, 1)
        g = randBetween(0, 1)
        b = randBetween(0, 1)
        print '%s %s %s' % (r, g, b)
    print ']'

def main():
    args = sys.stdin.readline()
    while args:
        arg = args.split()
        pixels = float(arg[0])
        rad = float(arg[1])
        num = int(arg[2])
        size = float(arg[3])

        print 'TransformBegin'
```

```
            cone(rad, num, size)
            print 'TransformEnd'
            sys.stdout.write('\377')
            sys.stdout.flush()
            # read the next set of inputs
            args = sys.stdin.readline()

if __name__ == "__main__":
    main()
```

RiPoints on a Cylinder

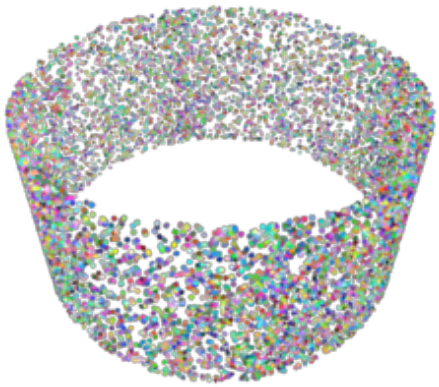


Figure 4

Listing 14 - Cylinder

```
import sys, math, random

def cylinder(rad, depth, height, num, size):
    print 'Points \"P\" ['
    for n in range(num):
        x = randBetween(-1, 1)
        y = 0
        z = randBetween(-1, 1)
        x,y,z = normalize(x,y,z)
        y = randBetween(depth, height) # <<---
        print '%s %s %s' % (x, y, z)
    print '] \"constantwidth\" [%s]' % size
    print '\"Cs\" ['
    for n in range(num):
        r = randBetween(0, 1)
        g = randBetween(0, 1)
        b = randBetween(0, 1)
        print '%s %s %s' % (r, g, b)
    print ']'

def main():
    args = sys.stdin.readline()
    while args:
        arg = args.split()
```

```
        pixels = float(arg[0])
        rad = float(arg[1])
        depth = float(arg[2])
        height = float(arg[3])
        num = int(arg[4])
        size = float(arg[5])

        print 'TransformBegin'
        cylinder(rad, depth, height, num, size)
        print 'TransformEnd'
        sys.stdout.write('\377')
        sys.stdout.flush()
        # read the next set of inputs
        args = sys.stdin.readline()

if __name__ == "__main__":
    main()
```

Spheres on a Sphere

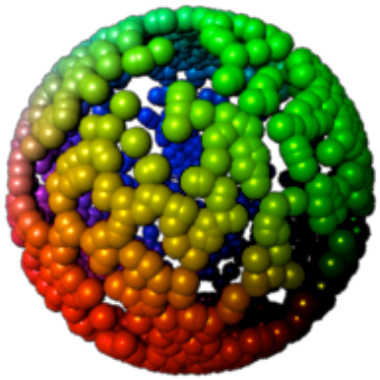


Figure 5

Listing 15 - Sphere

```
import sys, math, random

def spheres(rad, num, size):
    for n in range(num):
        x = randBetween(-1.0, 1.0)
        y = randBetween(-1.0, 1.0)
        z = randBetween(-1.0, 1.0)
        x,y,z = normalize(x,y,z)
        print 'TransformBegin'
        print 'Translate %s %s %s' % (x, y, z)
        print 'Color %s %s %s' % (x, y, z)
        print 'Sphere %s %s %s 360' % (size, -size, size)
        print 'TransformEnd'

def main():
    args = sys.stdin.readline()
```

```
    while args:
        arg = args.split()
        pixels = float(arg[0])
        rad = float(arg[1])
        num = int(arg[2])
        size = float(arg[3])

        print 'TransformBegin'
        spheres(rad, num, size)
        print 'TransformEnd'
        sys.stdout.write('\377')
        sys.stdout.flush()
        # read the next set of inputs
        args = sys.stdin.readline()

if __name__ == "__main__":
    main()
```

RenderMan Procedural Primitives
Blobbies

Introduction

Blobby objects, otherwise known as soft objects or iso-surfaces, are part of the RenderMan Specification. They are also refered to as RiBlobby because of the name of the function in the 'C' language binding of the RenderMan interface. Within Pixar's RenderMan Studio, there is also a Mel proc called RiBlobby. Blobby, or smooth blending effects can also be obtained with shaders, refer to,

```
fundza.com/rman_shaders/blobbies/blobbies.html
```

Specification of a Blobby Surface

The way that blobbies are specified in a rib file can become very complex. Using variations of the `Blobby` rib statement it is possible, for example, to "partition" the ellipsoids that make up a blobby surface into groups that merge with each other but do not merge with other groups. The code presented here considers a blobby to be made of a single homogeneous group whose ellipsoid elements "blob" together.

For full details of the Blobby specification refer to,

```
ProServerDocs\prman_technical_rendering\AppNotes\appnote.31.html
```

Before looking at the implementation of a help app that generates a complex cluster of blobbies, the reader should experiment with the simple blobby given in the rib file of listing 1.

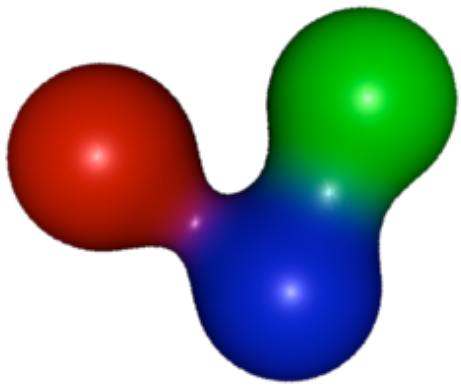


Figure 1

Listing 1

```
Display "untitled" "framebuffer" "rgba"
Format 250 250 1
Projection "perspective" "fov" 30
```

```
ShadingRate 1
LightSource "distantlight" 1 "intensity" 1.5 "from" [0 0 0]
                           "to" [0 0 1]
Translate  -0.2 0 5
Rotate 0   1 0 0
Rotate 0   0 1 0
Scale 1 1 -1
WorldBegin
    TransformBegin
        Surface "plastic"
        Blobby 3
            [1001 0
             1001 16
             1001 32
             0 3   0 1 2]
            [1 0 0 0  0 1 0 0  0 0 1 0 -0.5  0.2  0.0 1
             1 0 0 0  0 1 0 0  0 0 1 0  0.9  0.5  0.0 1
             1 0 0 0  0 1 0 0  0 0 1 0  0.5 -0.5  0.0 1]
            [""]
            "Cs" [1 0 0  0 1 0  0 0 1]
    TransformEnd
WorldEnd
```

Blobby Helper App

Listing 2 and 3 implement a blobby helper app in Python and Tcl. A rib that references the helpers is shown in listing 4.

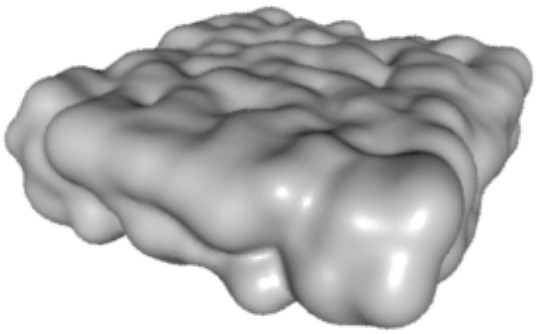


Figure 2
1000 ellipsoids in a volume 2 x 1 x 2 units.
The parameters in the rib file were "1000 0.5 4.0 1.0 4.0"

Listing 2 (blobby.py)

```
import sys, math, random

random.seed(5)
def randBetween(min, max):
    return random.random() * (max - min) + min

def getMatrix(size, width, height, depth):
    mat = '%s 0 0 0  0 %s 0 0  0 0 %s 0' % (size,size,size)
```

```
        x = randBetween(-width/2, width/2)
        y = randBetween(-height/2, height/2)
        z = randBetween(-depth/2, depth/2)
        mat += ' %s %s %s 1' % (x,y,z)
        return mat

    def blobby(num, size, width, height, depth):
        ellipsoid_ID = "1001 "
        index = 0
        out = 'Blobby %s ' %  num

        # begin the "code" block
        out += "[\n"
        for n in range(num):
            out += '%s' % ellipsoid_ID
            out += '%s ' % index
            index += 16
            out += "\n"
        # define the blobby operator and indices of
        # the blobs forming a "set" ie. group
        add_ID = 0
        blob_count = num
        out += '%s %s ' % (add_ID, blob_count)
        for n in range(num):
            out += '%s ' % n
        out += "]\n"

        # begin the transforms block
        out += "[\n"
        for n in range(num):
            out += '%s \n' % (getMatrix(size,width,height,depth))
        out += "]\n"

        # begin the depth map block
        out += "[\"\"]\n"
        return out

    def main():
        args = sys.stdin.readline()
        while args:
            arg = args.split()
            pixels = float(arg[0])
            num = int(arg[1])
            size = float(arg[2])
            width = float(arg[3])
            height = float(arg[4])
            depth = float(arg[5])

            print 'TransformBegin'
            print '%s' % blobby(num, size, width, height, depth)
            print 'TransformEnd'
            sys.stdout.write('\377')
            sys.stdout.flush()
            # read the next set of inputs
```

```
        args = sys.stdin.readline()

if __name__ == "__main__":
    main()
```

Listing 2 (blobby.tcl)

```
fconfigure stdout -translation binary

#-------------------------------------------------
proc randBetween { min max } {
    return [expr rand() * ($max - $min) + $min]
    }
#-------------------------------------------------
proc getMatrix { size width height depth } {
set mat "$size 0 0 0  0 $size 0 0  0 0 $size 0  "
append mat [format "%1.3f " [randBetween -$width/2 $width/2]]
append mat [format "%1.3f " [randBetween -$height/2 $height/2]]
append mat [format "%1.3f 1\n" [randBetween -$depth/2 $depth/2]]
return $mat
}
#-------------------------------------------------
proc blobby { num size width height depth} {
set ellipsoid_ID "1001 "
set index 0

set out ""
append out "Blobby $num \n"

# begin the "code" block
append out "\[\n"
for { set n 0 } { $n < $num } { incr n 1 } {
    append out $ellipsoid_ID
    append out $index
    incr index 16
    append out "\n"
    }

# define the blobby operator and indices of
# the blobs forming a "set" ie. group
set add_ID 0
set blob_count $num
append out "$add_ID $blob_count "
for { set n 0 } { $n < $num } { incr n 1 } {
    append out "$n "
    }
append out "\]\n"

# begin the transforms block
append out "\[\n"
for { set n 0 } { $n < $num } { incr n 1 } {
```

```
    append out [getMatrix $size $width $height $depth]
    }
append out "\]\n"

# begin the depth map block
append out "\[ \"\" \]\n"
return $out
}
#-----------------------------------------------
while { [gets stdin args] != -1 } {
    set pixels [lindex $args 0]
    set num [lindex $args 1]
    set size [lindex $args 2]
    set width [lindex $args 3]
    set height [lindex $args 4]
    set depth [lindex $args 5]

    puts "TransformBegin"
    puts [blobby $num $size $width $height $depth]
    puts "TransformEnd"

    puts "\377"
    flush stdout
    }
```

Listing 4 (helper_test.rib)

```
Display "shader_tester" "framebuffer" "rgba"
Format 300 200 1
Projection "perspective" "fov" 40
ShadingRate 10
LightSource "distantlight" 1 "intensity" 1.5 "from" [0 0 0]
                           "to" [0 0 1]
Translate  0 0.4 18
Rotate -30   1 0 0
Rotate 30   0 1 0
Scale 1 1 -1
WorldBegin
   AttributeBegin
      Surface "plastic"
      Procedural "RunProgram"
            ["/usr/bin/python FULL_PATH/blobby.py" \
            "1000 0.5 4.0 1.0 4.0"]
            [-2 2 -2 2 -2 2]
   AttributeEnd
WorldEnd
```

RSL
Shading Language Overview

Introduction

Because rib files are used to convey information from a modeling application to a renderer, RenderMan implicitly stresses an important distinction between, what is referred to in the Pixar literature, as

- **shape** - the geometry of an object ie. the output of a modeler, and
- **shading** - the appearance of an object ie. the output of a renderer.

For example, figure 1, despite its appearance, consists only of two closely spaced square polygons that been been transformed by the "shading" techniques of displacement, texture, specular and transparency mapping.

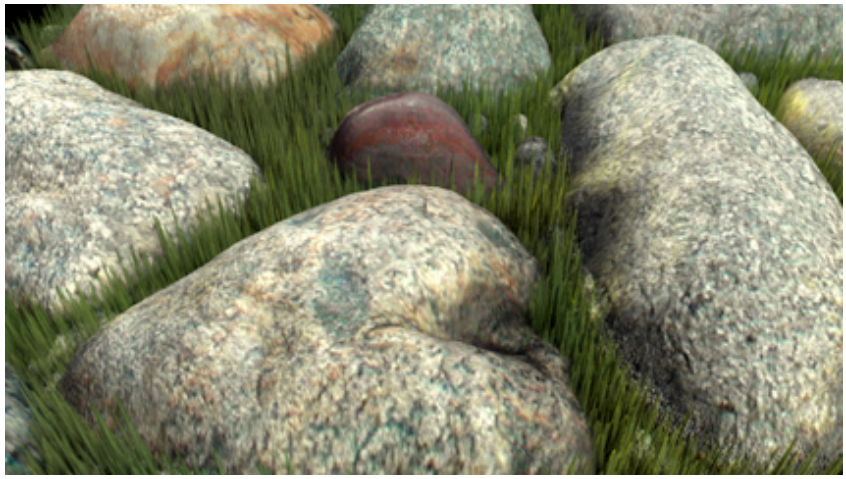


Figure 1 Image by Stephen Cody of the Savannah College of Art and Design

Shaders

Clearly, shaders play a crucial creative role in defining the appearance of CG a production. RenderMan has been adopted by many leading studios because it allows special purpose shaders to be added to those that already exist.

Individual shaders are small sub-routines (functions) written in a specialised programming language called the RenderMan Shading Language (RSL). The language enables new shaders to extend the creative possibilities of the renderer; it allows computer artists to find endless ways of controlling the appearance of a 3D scene through the use of custom shaders. The only limit is their imagination, their ability to write new, or adapt existing shaders and their creative flare at adjusting the parameters that control the visual effect of a shader.

In some respects RenderMan shaders are analagous to plugins for, say,

PhotoShop and AfterEffects. Plugins for those applications provide extra functionality to their host program. Likewise, shaders "work" within the environment of a renderer.

Shader Types

While shaders are generally independent of each other ie. any surface shader may be used with any displacement shader, each type of shader has a specific role in the rendering process. Therefore, a surface shader such as plastic cannot be used as a displacement shader.

RenderMan divides the rendering process into six tasks each of which uses a distinctive type of shader. They are,

- light source shaders,
- surface shaders,
- displacement shaders,
- volume shaders,
- transformation shaders, and
- imager shaders.

Shaders calculate specific values at more or less regular intervals across the surfaces being shaded by a renderer. A RenderMan complient renderer sub-divides each object in a 3D scene into a fine mesh of **micro-polygons**.

A renderer, as a consequence of processing a rib file, or some other source of rib information, makes data available to the shader so that the shader can calculate specific values. A displacement shader, for example, calculates a displaced location and orientation for each micro-polygon. A surface shader, on the other hand, determines the apparent color and opacity of each micro-polygon.

Shading Language Variables

Data going "into" a shader, as well as the values calculated by a shader are stored in memory locations that hold, what is referred to, as **variables**. Each time a shader is called ie. used by the renderer, it gets data from,

1. specific parameter values assigned to a shader, perhaps as a result of using Maya and Pixars SLIM shader interface - these are stored in a shaders **instance variables**,
2. internal data the renderer calculates and then makes available to a shader - these are stored in **global variables**, and finally,
3. private data that temporarily stores the results of its own calculations are kept in **local variables**.

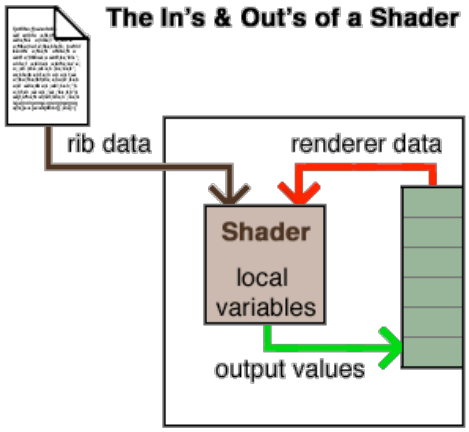


Figure 2

Shaders use global variables to **read data from** the renderer (shader input), for example, the color of a surface.

Shaders also use some global variables to **write data to** the renderer (shader output), for example, the apparent color of the light leaving the surface being shaded.

Shading Language Data Types

The Shading Language uses the following data types:

- float, same as the 'C' language,
- string, similiar to an array of characters in the 'C' language,
- point, stores the xyz coordinates of a location in 3D space,
- normal, stores the xyz coordinates of a surface normal,
- vector, stores the xyz coordinates of a vector,
- color, represents the color and opacity of a light source or a surface,
- matrix, a list of 16 floats.

Notice that integers are not supported by the language, therefore, all single values must be declared as a float. Also local variables cannot use the string data type.

Writing and Compiling a Shading Language File

Writing a shader in the Shading Language is similiar to writing an application in the 'C' language. Like a 'C' language source file (.c), code in the Shading Language is written with a text editor, but named with a .sl file extension. The easiest way to write a shader is to use the Cutter text editor because it has the following facilities.

- access to simple shader templates - figure 3,
- syntax coloration of RSL code,
- alt + e + double click on a RSL keyword key to access Pixar's documentation,
- alt + e hot key compilation of shader source code.

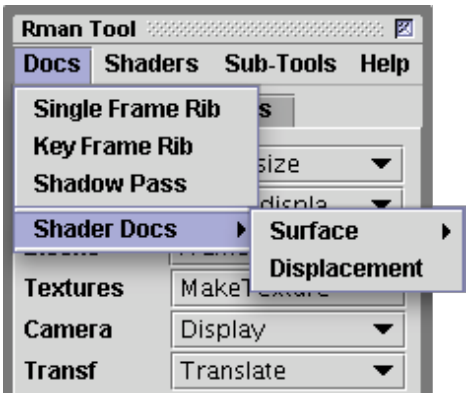


Figure 3 - accessing shader templates

Successful compilation produces a shading language object file ie. a shader. The extension of the shader file will vary from one RenderMan complient system to another. For example, Pixar's system uses ".slo" as the file extension for shaders compiled with their "shader" compiler. The name of the shader file will match the name of the shader defined by the code rather than the name of the .sl file. When using Cutter, shader files will be saved to a "shaders" directory specified by the user in Cutter's preferences.

SL source code files can be compiled from a command prompt window (Windows) or a shell (linux and MacOSX). For example, compiling a file called test.sl, using Pixars compiler, is done as follows,

```
(prompt%) shader test.sl
```

A Basic Rib File For Testing a Shader

Once a shader has successfully compiled you will want to test it. This can be accomplished using Maya (plus a Pixar's mtor or Rfm plugin) or Houdini (does not require a plugin). Alternatively, a sample rib file might be edited so that it uses your new shader. Again, Cutter speeds up development by generating and rendering either single frame or multiple frame rib files.

The following rib file can be used to test a shader. A file of this type is generated automatically by Cutter.

Listing 1

```
Display "shader tester" "framebuffer" "rgba"
Format 427 240 1
Projection "perspective" "fov" 40
ShadingRate 1

Translate 0 0 5
Rotate -30 1 0 0
Rotate   0 0 1 0
Scale 1 1 -1
```

```
WorldBegin
    LightSource "pointlight" 1 "intensity" 35
                "from" [1 4 1]
    AttributeBegin
        Surface "YOUR_SHADER"
        Scale 4 4 1
        Polygon "P" [-0.5 0 -0.5  -0.5 0 0.5
                      0.5 0 0.5  0.5 0 -0.5]
                "st" [0 0  0 1  1 1  1 0]
    AttributeEnd
WorldEnd
```

When this rib file is rendered via Cutter a dialog box will prompt the user to add three Options to the beginning of the document. The Options will be automatically added to the users rib file. For example,

```
Option "searchpath" "texture" "../../textures"
Option "searchpath" "shader" "@:../../shaders"
Option "searchpath"
       "archive" "../archives:Cutter_Help/templates/Rib"
```

Without these `Option` lines the user will be required to specify full paths to their custom shaders, textures and rib archives. By default, rib files generated by Cutter always begin with Options based on the users settings in Cutter's preferences.

RSL
What is a Surface Shader?

Surface Shader Algorithm

Writing a shader is like preparing a meal. While different recipes use different ingredients, all recipes use a general set of rules as well as applying specific rules that make a particular dish unique. An algorithm is a list of rules that must be followed to achieve a certain result. The purpose of a surface shader is to

- calculate the apparent surface opacity,
- calculate the apparent surface color
  ie. the color of the light leaving an object,

The renderer uses the opacity information to 'mix' background and foreground surface colors so that background objects in a 3D scene will "show through" any semi-transparent foreground objects. The following algorithm (recipe) lists the four steps that the standard shader, **plastic**, follows in order to set the appropriate opacity and color of the point on the surface of an object that is being shaded,

1	Make a copy (n) of the surface normal (N) then, using the viewing vector (I), ensure another copy (nf) faces the camera	
2	Set the apparent opacity of the surface (Oi)	
3	Find the colors of the light that is coming directly from the light sources and set the 'response' of the surface to those colors. An overall color is found by (generally) making three (ambient, diffuse and specular) lighting calculations.	
	3.1	add the colors of all the light sources that contribute ambient light,
	3.2	add the colors of the light sources that contribute to the diffuse appearance of the surface,
	3.3	add the colors of the light sources that contribute to the specular (shiny) highlights of the surface,
	Before being added, the ambient, diffuse and specular components are scaled by "Ka", "Kd" and "Ks". This enables an artist to control how an object responds to the lights in a scene.	
4	Set the apparent color (Ci) of the surface by combining the light color found in step 3 and the opacity found in step 2.	

The Geometry of Shading

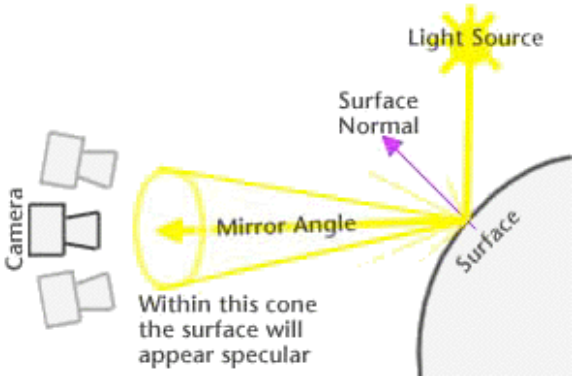


Figure 1

The Components of Lighting

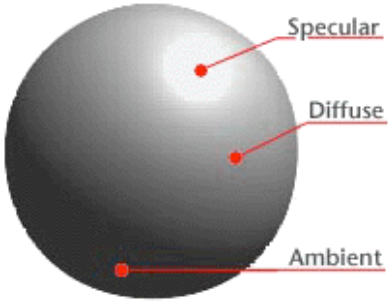


Figure 2

Surface Shading & Global Variables

The following table lists the global variables accessible to a surface shader. Those shown in red are "read-only", those in green are the variables to which a surface shader must assign values.

Global Variable	Meaning
Ci	apparent color of the surface (output)
Oi	apparent opacity the surface (output)
Cs	true surface color (input)
Os	true surface opacity (input)
N	surface shading normal
s, t	surface texture coordinates
P	surface position
Ng	surface geometric normal
u,v	surface parameters
du, dv	change in u, v across the surface
dPdu,dPdv	change in position with u and v
L	direction from surface to light source
Cl	light color
I	direction of a ray stricking a surface point
E	position of the camera

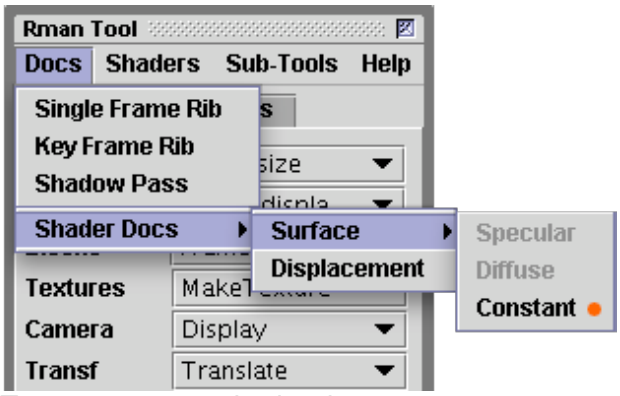
RSL
Writing Surface Shaders

Overview

This tutorial covers the basics of using the RenderMan Shading Language for the purpose of writing surface shaders. Several shaders are presented that can serve as starting points for the readers own explorations. The reader is encouraged to use the Cutter text editor for shader writing. Details of how it should be set up are given in the tutorial "Cutter: Shader Writing" This tutorial develops a series of variations of a basic constant shader. Finally, issues of diffuse lighting are addressed. The shading techniques used in this tutorial do not require ray tracing.

A Basic Surface Shader

Prior to shading an object a RenderMan complient renderer subdivides the surface of an object into mirco-polygons. The role of a surface shader is to determine the apparent surface opacity and color of each micro-polygon. The first set of shaders in this section are based on Cutter's "Constant" template surface shader.



To create a new shader document select either "Diffuse" or "Constant"

Although a constant shader does not consider the effect of lighting it is, nonetheless, a very good starting point for learning about shader writing.

The Constant Color Shader

```
/* Shader description goes here */
surface
constant_test(float  Kfb = 1  /* fake brightness */)
{
color    surfcolor = 1;

/* STEP 1 - set the apparent surface opacity */
Oi = Os;
```

```
/* STEP 2 - calculate the apparent surface color */
Ci = Oi * Cs * surfcolor * Kfb;
}
```

In `STEP 1` the apparent surface opacity is assigned the same value of the true surface opacity. In other words, this shader ensures a surface will conform exactly to the value of the `Opacity` statement in the rib file.

```
Opacity 1 1 1    # this will define the value of "Os"
Color   1 1 1    # this will define the value of "Cs"
Surface "constant_test" "Kfd" 1
Polygon "P" [data....]
```

The global variables that defines the apparent surface opacity and the true surface opacity are `Oi` and `Os`. Hence, the assignment,

```
Oi = Os;
```

ensures that nothing fancy is being done to the opacity of an object.

In `STEP 2` the apparent surface color (`Ci`) is assigned the value of the true surface color (`Cs`) tinted by an internally defined variable called `surfcolor`. Colors are "combined" by their red, green and blue components being multiplied together. Color multiplication, filters (or tints) one color by another color. Because `surfcolor` is white ie. its rgb components are all equal to 1.0, it has no effect on the resulting color. In later examples, `surfcolor` will have a noticable effect on the final apparent color of a surface.

The multiplication by the apparent surface opacity (`Oi`) ensures the resulting color of each micro-polygon is **pre-multiplied** by its opacity. This enables the renderer to correctly composite the micro-polygons of foreground surfaces over micro-polygons of surfaces in the background.

Assigning a Color

Colors may be assigned as a single value (grayscale) or three individual (component) values, for example,

```
color c;   /* declare a variable of data type color    */
c = 0.8;                    /* grayscale color         */
c = color(0.3, 0.9, 0.5);   /* assign a specific color */
```

"RGB" is the default color space. Listing 1 applies a pale green color to a surface.

Listing 1

```
surface
constant_test1(float  Kfb = 1)
{
color  surfcolor = color(0.3, 0.9, 0.5);
```

```
Oi = Os;
Ci = Oi * Cs * surfcolor * Kfb;
}
```

Figure 1 - a polygon of constant color

Adding Color Parameters

Two color parameters have been added to listing 2. The `mix()` function uses these colors to create a ramp based on the '`t`' texture coordinate.

Listing 2

```
surface
constant_test2(float    Kfb = 1;
               color    top = 1,
                        lower = 0)
{
color    surfcolor = mix(top, lower, t);

Oi = Os;
Ci = Oi * Cs * surfcolor * Kfb;
}
```

The `Surface` statement in the rib file that referenced the shader is shown below.

```
Surface "constant_test2"
        "Kfb" 1.0
        "top"   [0.878 0.996 0.474]
        "lower" [0.580 0.690 0.988]
```

Figure 2 - a color ramp based on the 't' texture coordinate

A color may also be modified by adjusting one of its components ie.

```
setcomp(c, 0, 0.9); /* reset red to 0.9 */
```

In the example shown above, the function `setcomp()` has been used to set the red component to 0.9. The indices 0, 1 and 2 reference the red, green and blue components respectively.

Declaring an Array of Color Parameters

A color parameter may also be declared as an array. Listing 3 also uses the 't' texture coordinate but this time in conjunction with the `spline()` function.

Listing 3

```
surface
spline_test(float    Kfb = 1;
                color   c[4] = {1,0,1,0})
{
color    surfcolor = spline(t,c[0],c[0],c[1],c[2],c[3],c[3]);

Oi = Os;
Ci = Oi * Cs * surfcolor * Kfb;
}
```

For simplisity, the declaration of the default color values have been set to white and black ie `{1,0,1,0}`. However, the `color()` function can be used within the array declaration to set specific values ie.

```
color    c[4] = {color(1,1,1), color(1,1,0),
                 color(1,0,0), color(0,1,0}
```

The `Surface` statement in the rib file that referenced the shader is shown below.

```
Surface "spline_test"
        "Kfb" 1.0
        "c" [0.878 0.996 0.474   0.580 0.690 0.988
             0.623 0.305 0.658   0.349 0.674 0.427]
```

Figure 3a - a color spline based on the 't' texture coordinate

Cutter provides a simple color picker to help users interactively define the "rgb" values of a color. To use the picker, select the three values of a color and click the right mouse button (Windows & Linux) or Control + mouse click (MacOSX).

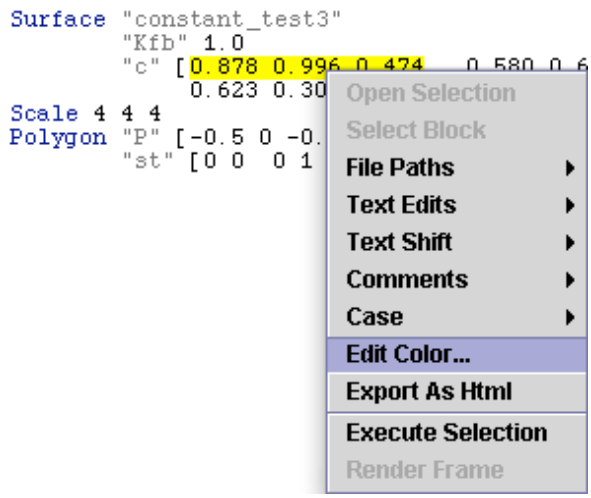


Figure 3b

Using Cutter's popup menu to edit "rgb" values.

Adding Uniform Noise

In this example the `mix()` function is again used to create a ramp based on the 't' texture coordinate but this time the `noise()` function is used to smoothly "jitter" the ramp.

Listing 4

```
surface
noise_test1(float    Kfb = 1,
                        amp = 0,    /* amplitude of the noise */
                        freq = 4;   /* frequency of the noise */
                color   top = 1,
                        lower = 0)
{
// Noise values range from 0 to 1.
float    ns =noise(s * freq, t * freq);

// Offset the true value of 't'. The 'amp' parameter will allow
// the artist to strengthen or weaken the visual effect.
float    tt = t + ns * amp;

color    surfcolor = mix(top, lower, tt);
Oi = Os;
Ci = Oi * Cs * surfcolor * Kfb;
}
```

The `Surface` statement that referenced the shader is shown below.

```
Surface "noise_test1"
        "Kfb" 1.0
        "amp" 0.8
        "freq" 9
        "top" [0.984 0.976 0.364]
        "lower" [0.925 0.317 0.317]
```

Figure 4a - a noisey color ramp

To ensure the "jittering" occurs around the mid-point of the range of values generated by the `noise()` function it is common practice to subtract 0.5 from the "raw" noise value. A slightly different visual effect, figure 4b, is obtained by using the code shown below.

```
float    ns = abs(noise(s * freq, t * freq) - 0.5);
```

Figure 4b

Adding Non-Uniform Noise

The following shader is similiar to listing 3 except that the two input colors are noisely mixed, more or less uniformly, across a surface. The frequency of the noise in 's' and 't' can be individually controlled - hence the stretching shown in figure 5a.

Listing 5

```
surface
noise_test2(float    Kfb = 1,
                     sfreq = 4,     /* s frequency of the noise */
                     tfreq = 4,     /* t frequency of the noise */
                     lo = 0.4,
                     hi = 0.5;
            color    hiColor = color(0.490,0.894,0.478),
                     loColor = color(0.286,0.411,0.678))
{
float    ns = noise(s * sfreq, t * tfreq);

float    blend = smoothstep(lo, hi, ns);
color    surfcolor = mix(loColor, hiColor, blend);
Oi = Os;
Ci = Oi * Cs * surfcolor * Kfb;
}
```

The `smoothstep` function ensures that `mix` returns either "hiColor" or "loColor" above and below the thresholds of "lo" and "hi". However, between those thresholds, `smoothstep` returns a value between 0.0 and 1.0. The shader blends the two colors in the transition "zone" between "lo" and "hi" and gives a fairly good anti-aliased pattern.

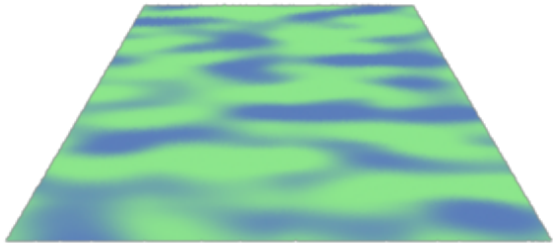


Figure 5a

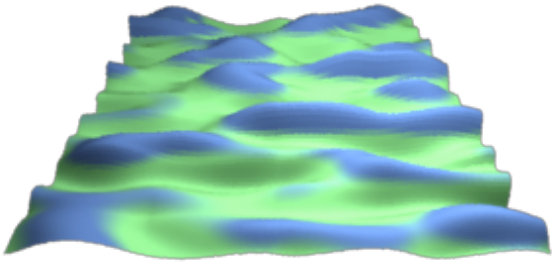


Figure 5b - noise visualized as a height field.

3D Noise

The previous two shaders generated noise values on the basis of the '`st`' texture coordinates of a surface and as a result their patterns were based on 2D noise. Using the '`st`' coordinates in this way generates a pattern that is "stuck" to the surface of an object to which the shader is assigned. There are ocassions, however, when a pattern based on 3D noise is required. The shader in listing 6 uses the surface point (`P`) as an input to the `noise` function.

Listing 6

```
surface
noise_test3(float    Kfb = 1,
                     freq = 4,     /* frequency of the noise */
                     lo = 0.4,
                     hi = 0.5)
{
float    ns = noise(P * freq);

Oi = smoothstep(lo, hi, ns);
Ci = Oi * Cs * Kfb;
}
```

To illustrate what is meant by "3D noise" the shader modifies the apparent opacity of a surface - in effect acting as an irregular "cookie-cutter". To further emphasize the 3D nature of the effect, figure 6 shows a rendering of a stack of square polygons all of which share the "noise_test3" shader.

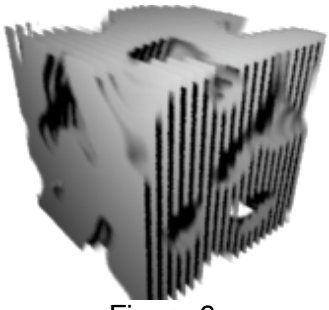
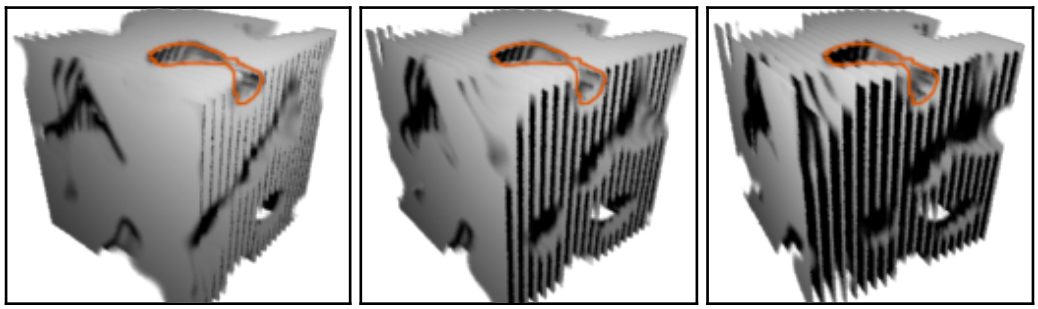


Figure 6

3D Noise & Coordinate Space

The problem with the "noise_test3" shader is that the xyz location of surface point `P` is measured from the origin of the camera coordinate system - it is said to be in "camera space". Careful comparison of the following three images shows there is a problem with noise that uses a point defined in camera space. Although the polygonal objects have been rotated, the "holes" created by 3D noise are in the same location relative to the picture frame. For emphasis, one of the static features is outlined in red.



Rotations of 40, 50 and 60 degrees

Th `noise`() function can calculate a value based on xyz values measured from the origin of any coordinate system. The `transform`() function is used to convert a location defined in one coordinate system into the corresponding location measured in another coordinate system. The result of using a copy of point `P` that has been transformed (re-measured) is that a visual effect based on 3D noise can be "parented" to any (named) coordinate system. Listing 7 demonstrates the use of the `transform`() function.

Listing 7

```
surface
noise_test4(float    Kfb = 1,
                     freq = 4,     /* frequency of the noise */
                     lo = 0.4,
                     hi = 0.5;
            string   space = "shader")
{
point    pp = transform(space, P);
float    ns = noise(pp * freq);
float    blend = smoothstep(lo, hi, ns);
```

```
Oi = blend;
Ci = Oi * Cs * Kfb;
}
```

Because the 3D noise is parented to "shader" space, which in this example is effectively the same as "object" space, the irregular holes remain in fixed locations relative to the stack of polygons - figure 7a.

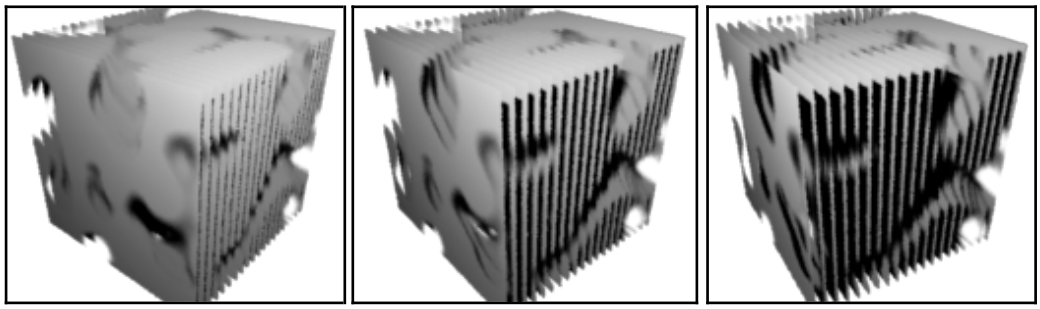


Figure 7a

User Defined Coordinates Systems

In addition to using any of the four predefined space names ie. "camera", "world", "object", and "shader", users can create custom coordinate systems with which they can control a shader. Cutter's Rman Tools palette enables the rib statements that define a coordinate system to be conveniently inserted into a rib file.

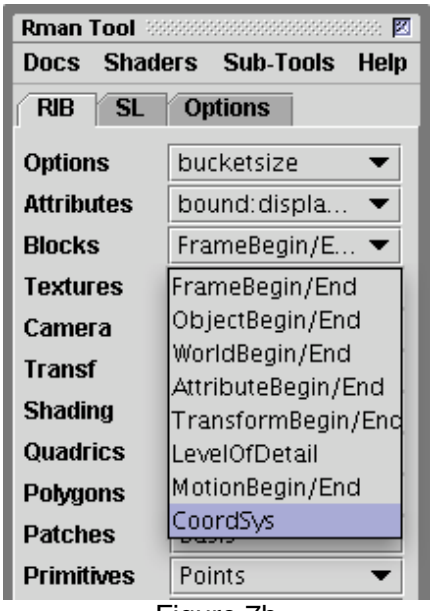
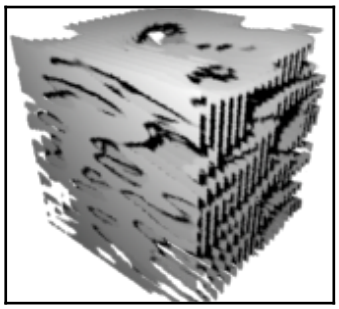


Figure 7b

```
TransformBegin
  Translate 0 0 0
  Rotate  0 1 0 0
  Rotate  0 0 1 0
  Rotate  0 0 0 1
  Scale   1 0.25 1
  CoordinateSystem "myspace"
TransformEnd
TransformBegin
  Surface "constant_test7" "Kfb" 1.0
        "freq" 1 "space" ["myspace"]
  ReadArchive "stack.rib"
TransformEnd
```

Using Cutter's drop-down menu to add a custom coordinate system to a rib file and accessing the custom coordinate system via the shaders "space" parameter.

Figure 7c shows the effect of scaling a user-defined coordinate system named "myspace".



Diffuse Illumination

The next set of shaders calculate the color of the diffuse illumination on a micro-polygon. The diffuse, also known as Lambert, illumination is derived from the angle between the surface normal and the (incident) ray of light striking a surface. When a micro-polygon directly "faces" the incident light it receives maximum illumination. When its normal makes an oblique angle to the incident light the illumination on the micro-polygon diminishes (drops off) in proportion to the cosine of the angle.

The `diffuse`() function "steps over" all the lights in a scene and returns a single color that represents the combined diffuse illumination striking a micro-polygon.

Listing 8

```
surface
diffuse_test1(float  Kd = 1,
                     doFace = 1)
{
/* STEP 1 - make a unit copy of the surface normal */
normal    n = normalize(N);
normal    nf = n;

/* STEP 2 - force the surface normal to face the camera */
if(doFace)
    nf = faceforward(n, I);

/* STEP 3 - set the apparent surface opacity */
Oi = Os;

/* STEP 4 - calculate the diffuse lighting component */
color    diffusecolor = Kd * diffuse(nf);

/* STEP 4 - calculate the apparent surface color */
Ci = Oi * Cs * diffusecolor;
}
```

The `faceforward`() function returns a copy of the true surface normal forced to face the incident ray. The xyz coordinates of the incident ray are stored in the global variable `I`. Because ray tracing is not being used the incident ray will always be the camera ray, otherwise known as the viewing vector.

The effect of not using `faceforward`() can be seen on the quadric sphere shown below on the left. The darkness of the interior surface of the sphere represents the diffuse illumination of the **rear** of the object. When an interior surface is viewed in this way we are, in effect, viewing the front of the rear surface! Unless the stereo rendering capabilities of Pixar's prman renderer are being used, it is traditional for shaders always to flip their normals using the `faceforward`() function. In general, the first two lines of code of a surface shader are usually these,

```
normal    n = normalize(N);
normal   nf = faceforward(n, I);
```

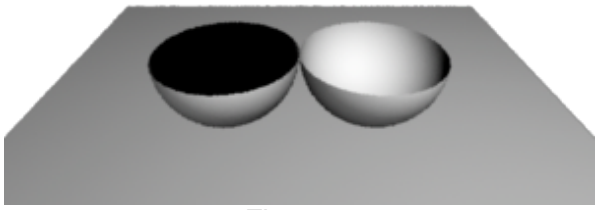


Figure 8
Illumination with and without the use of `faceforward(n,I)`

Cartoon Shading

High contrast or cartoon-like shading can be obtained by thresholding the diffuse illumination. In the rendering shown in figure 9 values of diffuse less than 0.5 are treated as if they were black. Values slightly higher ie. adjusted by the "blur" parameter, are considered to be white. Using the `smoothstep`() function ensures there is a narrow trasition zone of gray between the white and black areas.

Listing 9a

```
surface
cartoon_test1(float  Kd = 1,
                     midpoint = 0.5,
                     blur = 0.02)
{
normal    n = normalize(N);
normal    nf = faceforward(n, I);
color     surfcolor = 1;

/* Calculate the diffuse lighting component */
color     diffusecolor = Kd * diffuse(nf);

/* Get the brightness of the diffuse lighting */
float      value = comp(ctransform("hsv", diffusecolor), 2);

/* Apply a black and white cutoff around a "midpoint" */
color    bw = smoothstep(midpoint, midpoint + blur, value);
Oi = Os;
Ci = Oi * Cs * surfcolor * bw;
```

```
}
```

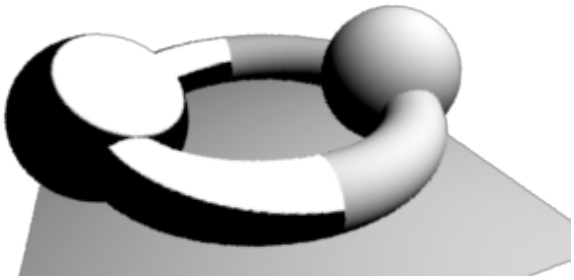


Figure 9a
Illumination with and without high contrast

Figure 9b
Banding using the mod() functio

As a variation of the cartoon "theme" the next shader, listing 9b, applies a repeating pattern to the threshold to produce a series of bands. For more information about the use of the mod() function and repeat patterning refer to the tutorial, RSL: Repeating Patterns.

Listing 9b

```
surface
cartoon_test2(float  Kd = 1,
                     midpoint = 0.5,
                     blur = 0.2,
                     repeats = 5)
{
normal   n = normalize(N);
normal   nf = faceforward(n, I);
color    surfcolor = 1;

/* Calculate the diffuse lighting component */
color    diffusecolor = Kd * diffuse(nf);

/* Get the brightness of the diffuse lighting */
float     value = comp(ctransform("hsv", diffusecolor), 2);

/* Apply a repeat factor */
value = mod(value * repeats, 1);

/* Apply a black and white cutoff around a "midpoint" */
color   bw = smoothstep(midpoint, midpoint + blur, value);
Oi = Os;
Ci = Oi * Cs * surfcolor * bw;
}
```

Inside/Outside Shading

Some unusual visual effects can be obtained by combining the high contrast

shading of listings 9a/9b with the diffuse shading of listing 8. The next shader, listing 10, uses the high contrast values to alter the apparent opacity of a surface. In effect, the shader causes the light that strikes a surface to behave like a "cookie-cutter". However, the apparent surface color is not effected by the high contrast but instead is shaded by the color returned from the `diffuse`() function.

Listing 10

```
surface
in_out_test1(float  Kd = 1,
                    midpoint = 0.5,
                    blur = 0.2,
                    repeats = 5)
{
normal    n = normalize(N);
normal    nf = faceforward(n, I);
color     surfcolor = 1;

/* Calculate the diffuse lighting component */
color     diffusecolor = Kd * diffuse(nf);

/* Get the brightness of the diffuse lighting */
float      value = comp(ctransform("hsv", diffusecolor), 2);

/* Apply a repeat factor */
value = mod(value * repeats, 1);

/* Apply a black and white cutoff around a "midpoint" */
color    bw = smoothstep(midpoint, midpoint + blur, value);

/* Modify the opacity */
Oi = bw * Os;

/* Use the regular diffuse color for the surface */
Ci = Oi * Cs * surfcolor * diffusecolor;
}
```

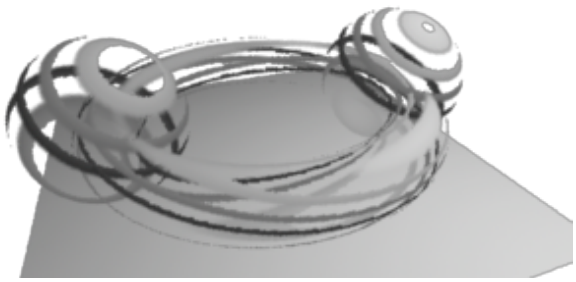


Figure 10
High contrast shading controls surface opacity while diffuse shading is used for the surface color.

Texture Mapping

When a surface is texture mapped the '`st`' coordinates of its micro-polygons are used to sample a color from the corresponding '`st`' location of an image. A slightly blurred (anti-aliased) color sample from the image is returned as a single color by the `texture`() function. The '`st`' texture coordinates are equivalent to latitude and longitude - figure 11a.

RenderMan's '`st`' texture space is the equivalent to the '`uv`' texture coordinates of Maya and Houdini. Note, however, the origin of the '`st`' space of the image is in the top-left whereas for Maya and Houdini the origin of '`uv`' space is in the lower-left hand corner of an image.

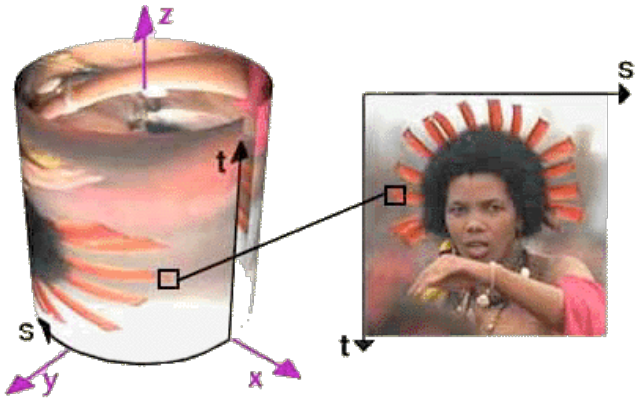


Figure 11a
Texture coordinates, mapping from an image to a quadric cylinder.

With the introduction of Pixar's RenderMan Studio (RMS) the situation with regard to the relative orientation of '`st`' and '`uv`' space is now different compared to the way that '`st`' was handled by their earlier product, RenderMan Artist Tools. Figure 11b illustrates the issue of '`st`' orientation for several Maya surfaces. It appears that RMS is swapping the '`s`' and '`t`' axes!

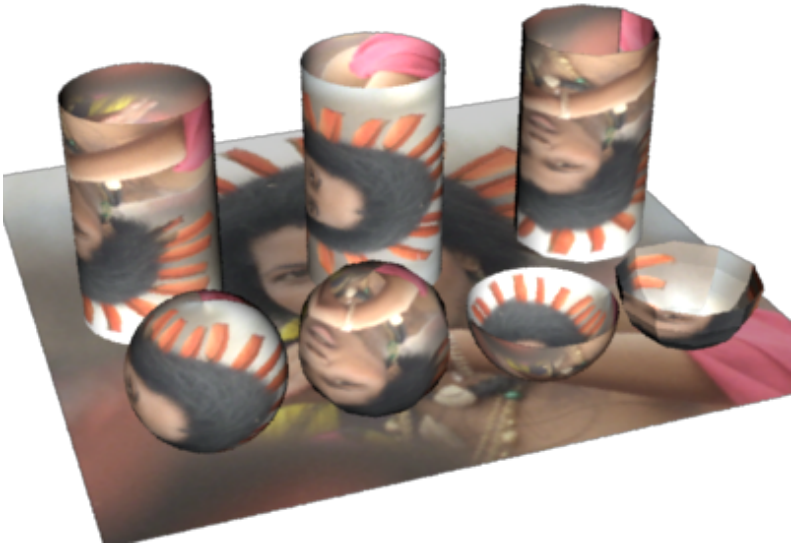


Figure 11b
Left to Right

Reading Texture Maps for Surface Coloration

The majority of RenderMan complient renderers do not directly use an image file for texture mapping but insted require the file to be converted to a texture file. Texture files contain representations of the original image at different scales. During convertion the pixel data in the texture file is filtered and this, combined with the texture files multiple images (mip maps), results in the renderer being able to perform efficient anti-aliased texturing.

Most RenderMan complient rendering systems have a utility application that converts images to textures. In the case of Pixar's system the utility is called `txmake`. Cutter has a simple Texture Tool, figures 11e and 11f, that enables image files to be converted and automatically saved to the users textures directory - refer to section "Setting up User Paths" at the beginning of this tutorial.

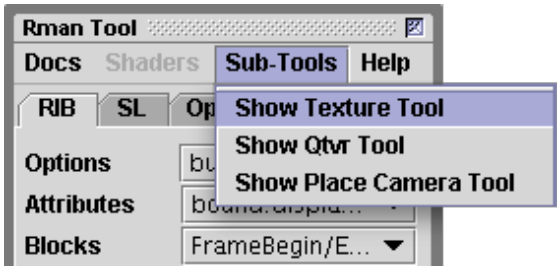


Figure 11e

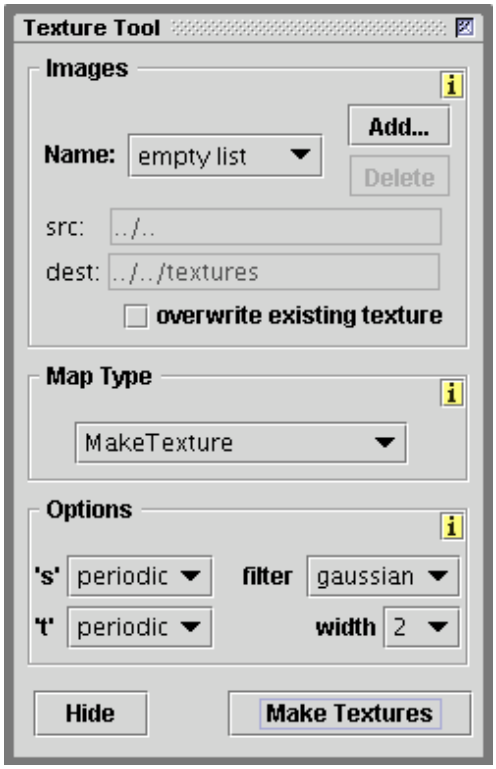


Figure 11f

Rather than using the Texture Tool it is often more convenient to execute a line of text. For example, selecting the following line of text and using the keyboard short cut Alt+e, Control+e or Apple+e is the same as executing the `txmake` command from the command prompt, shell or terminal.

```
# txmake G:/renderman/tiffs/swazi.tif G:/renderman/textures/swazi.tex
```

A comment at the beginning of a line is ignored when Cutter executes the text. Text may also be broken over sever lines.

```
# txmake G:/renderman/tiffs/swazi.tif
# G:/renderman/textures/swazi.tex
```

Listing 11 provides the code for the texture mapping shader used to render figure

11b.

Listing 11

```
surface
texture_test1(float  Kd = 1;
              string   texname = "")
{
normal    n = normalize(N);
normal    nf = faceforward(n, I);
color     surfcolor = 1;

if(texname != "")
    surfcolor = texture(texname);

Oi = Os;

color  diffusecolor = Kd * diffuse(nf);
Ci = Oi * Cs * surfcolor * diffusecolor;
}
```

Using a Texture Map for Surface Opacity

The `texture`() function can return a color or a float. In the case of a float the value corresponds to the red channel of the texture map. For example, the image shown in figure 12a was used as a texture map to render the square polygon seen in figure 12b. The red shape and the white border have contributed full opacity while the green and blue shapes have been ignored. The border is opaque because white has a red channel value of 1.0.

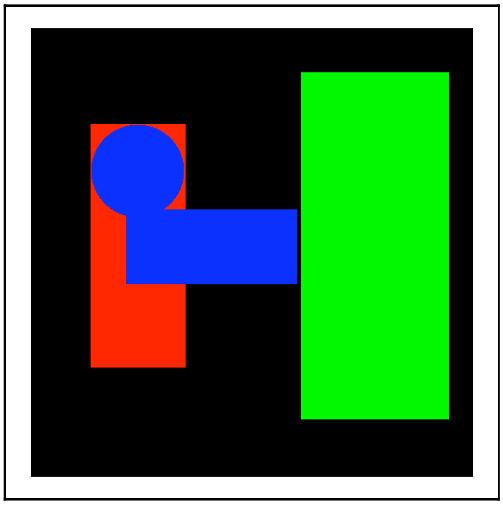


Figure 12a

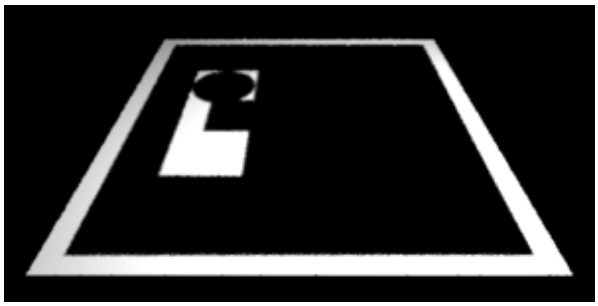


Figure 12b

Listing 12

```
surface
texture_test2(float  Kd = 1;
              string   texname = "")
```

```
{
normal    n = normalize(N);
normal    nf = faceforward(n, I);
color     surfcolor = 1;

Oi = Os;
if(texname != "")
    Oi = float texture(texname) * Os;

color  diffusecolor = Kd * diffuse(nf);
Ci = Oi * Cs * surfcolor * diffusecolor;
}
```

To ensure an opacity mapper, such as `texture_test2` shown in listing 12, handles a colored image map "properly" it would be better to use the average value of the red, green and blue channels ie.

```
    if(texname != "") {
        color c = texture(texname);
        float ave = (comp(c, 0) + comp(c, 1) + comp(c, 2))/3;
        Oi = ave * Os;
        }
```

Introduction

This tutorial focuses on some of the issues relating to the writing of directional light source shaders ie. those that use the `solar()` shadeop. Some of the shaders presented here demonstrate how data can be passed to custom surface shaders for the purpose of producing secondary images - AOV's. The tutorial, "RenderMan: AOVs - Secondary Images", provides an overview of this topic.

Before continuing the reader should refer to the tutorial "Cutter Setup: RenderMan Shader Writing". It explains how to configure Cutter for shader writing. In particular, the section, "Cutter's Open Shader Source Facility", should be reviewed because it explains how Cutter can quickly access the source code of the shaders that ship with different RenderMan compliant rendering systems.

For a brief description of how to use a custom light shader with RenderMan Studio and Maya refer to the tutorial "RMS:Using Custom Light Source Shaders".

Parallel Illumination

Listing 1 shows a slightly modified version of Pixar's classic "distantlight" shader. It is an example of a light source intended to mimic sunlight.

Listing 1 (basic_distant.sl)

```
light
basic_distant(float  intensity = 1;
              color  lightcolor = 1;
              point  from = point "shader" (0,0,0);
              point  to = point "shader" (0,0,1);
)
{
vector direction = to - from;
solar(direction, 0.0) {
    Cl = intensity * lightcolor;
    }
}
```

A rib file that implements a simple scene (figure 1) that can be used to test the shader is given in listing 2.

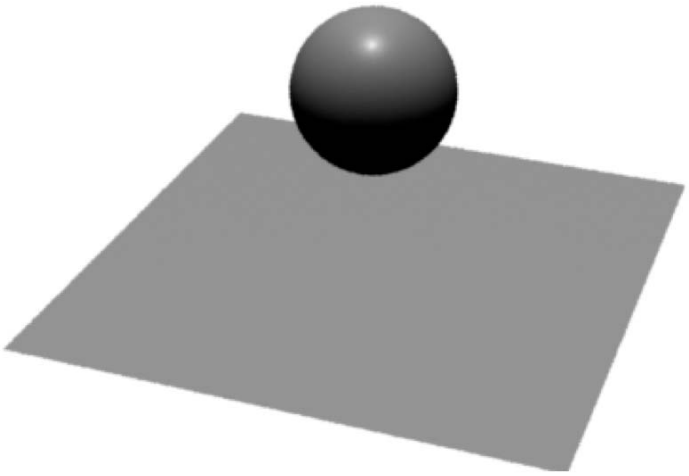


Figure 1

The reader should ensure the paths shown in red "point" to the directories in which they store their custom shaders, textures and archive (pre-baked) rib files.

Listing 2 (basic_distant.rib)

```
Option "searchpath" "shader" "@:../shaders"
Option "searchpath" "texture" "../textures"
Option "searchpath" "archive" "../archives:Cutter_Help/templates/Rib"

Display "distant_test" "framebuffer" "rgba"
Format 400 250 1
Projection "perspective" "fov" 17
ShadingRate 2

Translate  0 -0.05 3
Rotate -30  1 0 0
Rotate  20  0 1 0
Scale 1 1 -1
WorldBegin
    TransformBegin
        LightSource "basic_distant" 2 "intensity" 1
                    "from" [0 1 0] "to" [0 0 0]
    TransformEnd

    Surface "plastic"
    AttributeBegin
        Polygon "P" [-0.5 0 -0.5  -0.5 0 0.5  0.5 0 0.5  0.5 0 -0.5]
                "st" [0 0  0 1  1 1  1 0]
        Translate 0 0.35 0
        Sphere 0.15 -0.15 0.15 360
    AttributeEnd
WorldEnd
```

Setting a Fixed Direction

The first thing to note about the basic_distant shader is that it has two parameters that enable it's direction to be adjusted. Modern directional lights are written on the assumption they "face" toward negative Z. For example, the default direction of a Maya "directional" light is shown in figure 2.

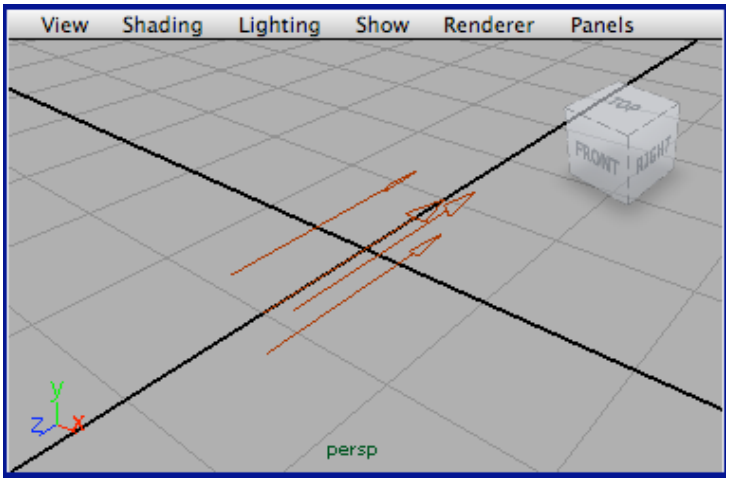


Figure 2

In the case of Maya, the light's transform node (figure 3) enable its orientation to be adjusted. A modified version of the basic_distant shader is shown below. Note the "from" and "to" parameters have been removed and it's direction has been "hard wired" to be negative Z.

Listing 3

```
light
basic_distant(float  intensity = 1;
              color  lightcolor = 1;
             )
{
vector direction = vector "shader"(0, 0, -1);
solar(direction, 0.0) {
    Cl = intensity * lightcolor;
    }
}
```

The rib file, basic_distant.rib, should be edited so that two transformations provide the rib "equivalent" of the transformations shown in figure 3.

```
TransformBegin
    Translate 0 1 0
    Rotate -90 1 0 0
    LightSource "basic_distant" 2 "intensity" 1
TransformEnd
```

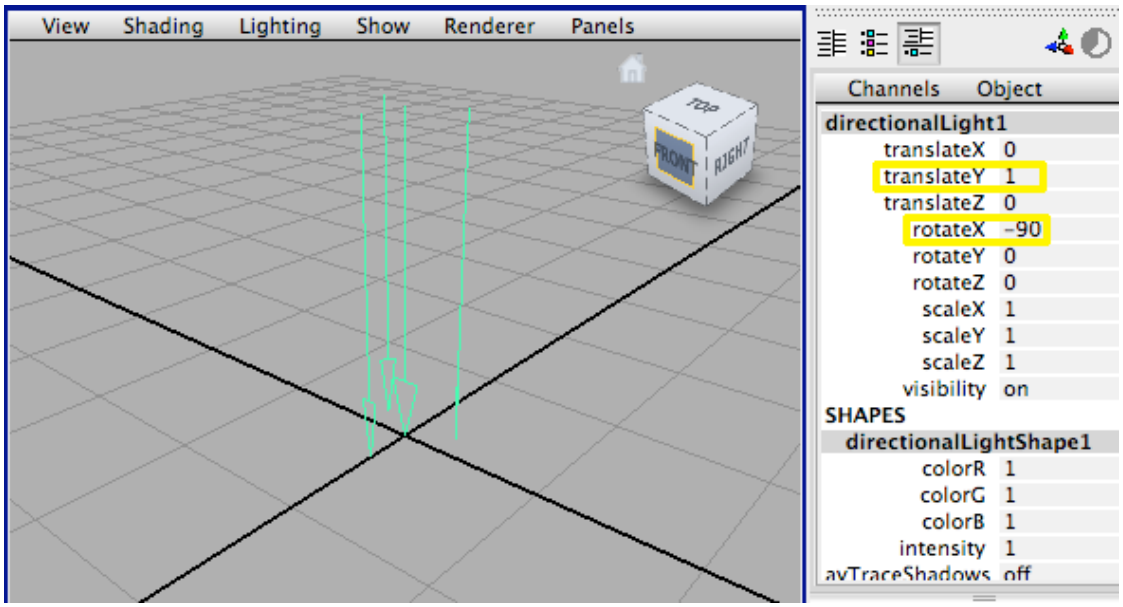


Figure 3

The "shader" Coordinate System

Note that in listings 1 & 2 the `direction` variable has been assigned xyz values in a pre-defined coordinate system named "shader" ie.

```
    vector direction = vector "shader" (0,0,-1);
```

However, had the vector been declared as,

```
    vector direction = vector(0,0,-1);
```

it would would have been, by default, defined in the camera coordinate system ie. "camera" space. In other words, the xyz values would be distances meaasured from the origin of the camera. So, what is "shader" space and why has it been used?

The "shader" coordinate system is identical to the coordinate system that was active when the light was instanced in the Rib file. Using "shader" space to define the direction vector, in effect, "parents" the light to the coordinate system that was created by the `TransformBegin` statement, and subsequently transformed by the `Rotate` and `Translate` commands that appear immediately before the `LightSource` statement. Consequently, only by defining the direction vector within the "shader" coordinate system will the light "respond" to the transformations intended to orientate it. As an experiment, the reader should reset the direction to,

```
    vector direction = vector(0,0,1);
or
    vector direction = vector "camera" (0,0,1);
```

The shader should be re-compiled and the scene re-rendered. Notice that no matter what values are specified for the `Rotate` and `Translate` statements in the Rib file the illumination of the scene remains constant. In particular, note the specular hilite on the sphere (figure 4) indicates the illumination is coming from the camera.

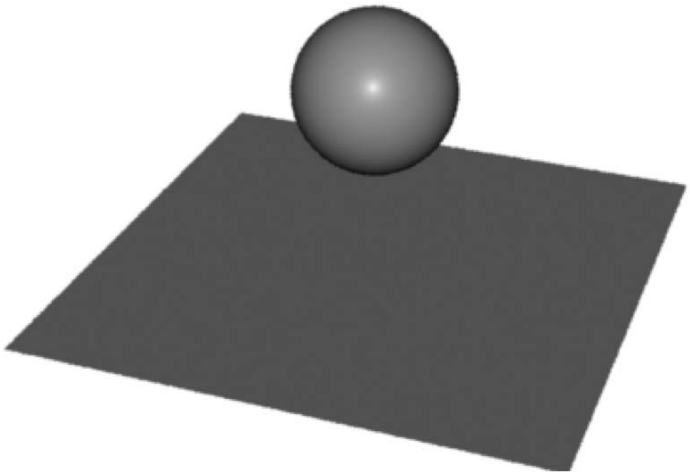


Figure 4

But it Does Not Work in Maya & Houdini!

If the reader assigns the shader from listing 3 to a directional light source in Maya, using RenderMan Studio, or in Houdini they will see the illumination is reversed - a downward facing light will illuminate the scene as if it were pointing upward! So why does the shader behave correctly when tested with the rib file shown in listing 2 but not when used in Maya or Houdini?

For reasons known only to the software engineers at Pixar and Side Effects their products insert, into the rib stream, a negative scaling on the z-axis immediately prior to instancing the light source shader. For example, the snippet shown below is from a rib file generated by RenderMan for Maya Pro (RenderMan Studio).

```
TransformBegin
    Attribute "identifier" "string name" ["directionalLightShape1"]
    Transform [ 1 0 -0 0  -0 0 -1 0  0 1 0 0  0 0 0 1 ]
    Scale 1 1 -1   # ???
    LightSource "basic_distant" "RenderManLight2"
                "float intensity" [1] "color lightcolor" [1 1 1]
TransformEnd
```

As a consequence, for the shader to behave correcly with Maya or Houdini it is necessary to declare the direction vector as pointing toward **positive** Z.

```
vector direction = vector "shader" (0,0,1);
```

This means the transform block containing the light in listing 2 must also apply a negative Z scaling.

```
TransformBegin
    Rotate -90 1 0 0
    Scale 1 1 -1
    LightSource "basic_distant" 2 "intensity" 1
TransformEnd
```

Thus proving that "two wrongs can make a right"!

The solar() Function

At the heart of a directional light source shader is the `solar()` function. The syntax of the function is rather strange because it has a code block and as such it looks

more like a a looping statement, such `while()` or `for()`, rather than a "regular" RSL shadeop.

```
solar(direction, 0.0) {
    Cl = intensity * lightcolor;
    }
```

There are two special global variables that can be accessed within the function's block of code. It is the responsibility of the shader to assign a color to `Cl` - the output light color. The second global, `L`, is a vector that specifies the direction from the light source to the surface it is illuminating - the input light direction. However, `L` is not used in the code sample shown above.

The value of the second parameter to `solar` is usually zero. The parameter specifies the deviation angle between the rays. Hence, a value of zero indicates the rays of light are parallel.

Listing 4

```
light
basic_distant(float  intensity = 1;
              color  lightcolor = 1;
              float  angle = 0;
             )
{
vector direction = vector "shader"(0,0,1);
solar(direction, radians(angle)) {
    Cl = intensity * lightcolor;
    }
}
```

In listing 4 an "angle" parameter has been added to the shader. The effect of different values of "angle" are shown in figure 5. For moderate values of "angle" the visual effect is to wrap the illumination around the sphere. However, for large values of "angle" the wrapping becomes noticably "distorted".

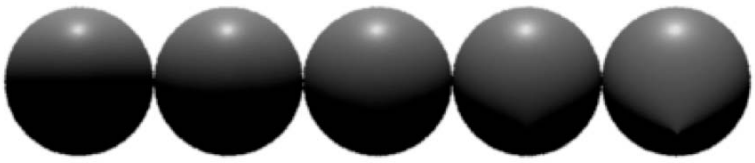


Figure 5
"angle" from left to right: 0, 20, 40, 60 and 90 degrees

Shadows

Listing 5 demonstrates the use of the `shadow()` function to read a value from a shadow map. Because the function returns 1.0 when the surface point is "in shadow" and 0.0 when "not in shadow" the return value is subtracted from 1.0 before modifying the output light color.

Listing 5 (shd_distant.sl)

```
light
shd_distant(float  intensity = 1;
            color  lightcolor = color(1,1,1);
            string shadowname = "";
            float  width = 1;
            float  samples = 16 )
{
vector direction = vector "shader" (0,0,1);
solar(direction, 0.0) {
    Cl = intensity * lightcolor;
    // Attenuate the output light color by the value
    // returned from a shadow (texture) map
    if(shadowname != "")
        Cl *= 1 - shadow(shadowname, Ps, "samples", samples, "width", width);
    }
}
```

The rib file shown below can be used to generate the two depth maps required for the beauty pass render - listing 7.

Listing 6 (shadow_pass.rib)

```
Option "searchpath" "shader" "@:../shaders"
Option "searchpath" "texture" "../textures"
Option "searchpath" "archive" "../archives:Cutter_Help/templates/Rib"

PixelSamples 1 1
PixelFilter "box" 1 1
Hider "hidden" "jitter" [0]
Clipping 1 20
Format 512 512 1
Projection "orthographic"
ShadingRate 1

FrameBegin 1
    Display "./shd_map_80.tex" "shadow" "z"
    Translate  0 0 5
    Rotate -80 1 0 0
    Scale 1 1 -1
    WorldBegin
        AttributeBegin
            Translate 0 0.5 0
            Sphere 0.15 -0.15 0.15 360
        AttributeEnd
        AttributeBegin
            Polygon "P" [-0.5 0 -0.5  -0.5 0 0.5  0.5 0 0.5  0.5 0 -0.5]
                    "st" [0 0  0 1  1 1  1 0]
        AttributeEnd
    WorldEnd
FrameEnd
FrameBegin 2
    Display "./shd_map_100.tex" "shadow" "z"
    Translate  0 0 5
    Rotate -100 1 0 0
    Scale 1 1 -1
    WorldBegin
```

```
        AttributeBegin
            Translate 0 0.35 0
            Sphere 0.15 -0.15 0.15 360
        AttributeEnd
        AttributeBegin
            Polygon "P" [-0.5 0 -0.5  -0.5 0 0.5  0.5 0 0.5  0.5 0 -0.5]
                    "st" [0 0  0 1  1 1  1 0]
        AttributeEnd
    WorldEnd
FrameEnd
```

Listing 7 (beauty_pass.rib)

```
Option "searchpath" "shader" "@:../shaders"
Option "searchpath" "texture" "../textures"
Option "searchpath" "archive" "../archives:Cutter_Help/templates/Rib"

PixelSamples 8 8
Display "./shadow_test" "framebuffer" "rgba"
Format 400 250 1
Projection "perspective" "fov" 17
ShadingRate 1

Translate  0 -0.05 3
Rotate -30  1 0 0
Rotate  20  0 1 0
Scale 1 1 -1
WorldBegin
    TransformBegin
        Rotate -85 1 0 0
        Scale 1 1 -1
        LightSource "shd_distant" 1 "string shadowname" ["./shd_map_80.tex"]
                    "float intensity" 0.5 "float width" 4 "float samples" 32
    TransformEnd
    TransformBegin
        Rotate -95 1 0 0
        Scale 1 1 -1
        LightSource "shd_distant" 2 "string shadowname" ["./shd_map_100.tex"]
                    "float intensity" 0.5 "float width" 4 "float samples" 32
    TransformEnd

    Surface "plastic"
    AttributeBegin
        Polygon "P" [-0.5 0 -0.5  -0.5 0 0.5  0.5 0 0.5  0.5 0 -0.5]
                "st" [0 0  0 1  1 1  1 0]
    AttributeEnd
    AttributeBegin
        Translate 0 0.35 0
        Sphere 0.15 -0.15 0.15 360
    AttributeEnd
WorldEnd
```

The beauty_pass.rib shown in listing 7 will generate the image shown below.

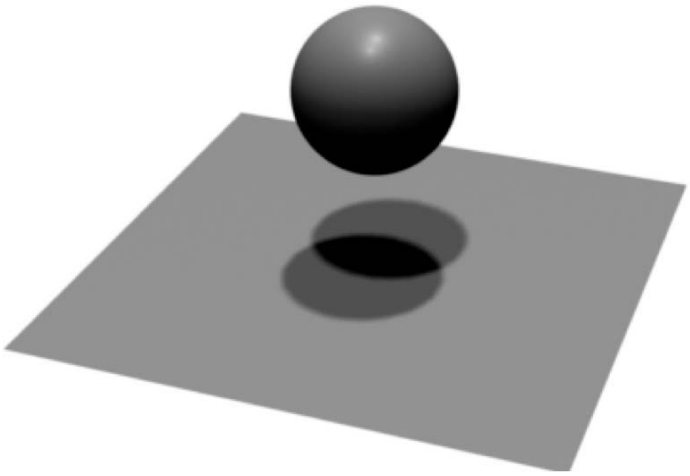


Figure 6

Rendering a Shadow AOV

The shd_distant shader provides an opportunity to explore how data calculated by a light source shader can be rendered into a secondary image. Although light shaders can have parameters declared as output variables they cannot be used **directly** as a source of data for secondary images. The data assigned to a light's output variables can, however, be read by a surface shader and then assigned to its output variables. Such outputs are known as AOVs - arbitrary output variables. The reader should refer to the tutorial "RenderMan: AOVs - Secondary Images" for background information on this topic.

Listing 8 is a slightly modified version of shd_distant. It has an extra parameter to enable the color of the shadow to be adjusted. It also has an output parameter that is assigned a float indicating the extent to which each micropolygon is in shadow.

Listing 8

```
light
shd_distant(     float  intensity = 1;
                 color  lightcolor = color(1,1,1);
                 color  shadowcolor = color(0,0,0);
                string  shadowname = "";
                 float  width = 1;
                 float  samples = 16;
 output varying float  __inshadow = 0)
{
vector direction = vector "shader" (0,0,1);
solar(direction, 0.0) {
    Cl = intensity * lightcolor;
    if(shadowname != "") {
        __inshadow = shadow(shadowname, Ps, "samples", samples, "width", width);
        Cl = mix(Cl, shadowcolor, __inshadow);
        }
    }
}
```

Because the `shadow()` function returns values in the range 0.0 to 1.0 it offers a convenient way of using the `mix()` function to set the output color of the light source.

Listing 9 gives the source of a surface shader that queries the light source output variable `__inshadow` and assigns the value to its own (AOV) output variable. Unlike the output variables of the light source, those of the surface shader can be used as sources of data (AOVs) for secondary images.

Listing 9 (diffuseAOV.sl)

```
surface
diffuseAOV(float Kd = 0.5;
 output varying float  _inshadow = 0)
{
normal  n = normalize(N);
color   diffusecolor = 0;
float   accum_inshadow = 0, inshadow;
color   accum_raw_Cl = 0, raw_Cl;

illuminance( P, n, PI/2 ) {
    if(lightsource("__inshadow", inshadow) == 1)
        accum_inshadow += inshadow;
    diffusecolor += Cl * normalize(L).n;
    }
// Clamp the _inshadow output
_inshadow = mix(0.0, 1.0, accum_inshadow);
Oi = Os;
Ci = Oi * Cs * diffusecolor * Kd;
}
```

The illuminance() loop

Although the `diffuseAOV` surface shader calculates the diffuse (Lambertian) response of a surface to illumination, for simplicity, it ignores the ambient and specular components. Unlike the the sample surface shaders in the tutorial, "RSL: Writing Surface Shaders", it does not use the `diffuse()` function. Instead, it makes use of the `illuminance()` function to accumulate its shading effect.

The `illuminance()` function executes its block of code for each light source in a scene, if and only if, it determines the light is capable of contributing an illumination value. That determination is based on the surface position (`P`), surface orientation (`N`) and a cone within which it samples in-coming light - like a spot light but in "reverse". A cone angle of `PI/2` ensures that each micropolygon samples a 180 degree hemisphere.

Figure 7 (shadow_test._inshadow.tif)

Within the illuminance block, the `lightsource()` function is used to query each light. If a light has a `__inshadow` variable its value is acculumated and finally assigned to the surface shaders own output variable. Listing 10 will render the secondary image shown in figure 7. The image contains an inverted mask of the overlapping shadows.

Listing 10

```
Option "searchpath" "shader" "@:../shaders"
Option "searchpath" "texture" "../textures"
Option "searchpath" "archive" "../archives:Cutter_Help/templates/Rib"
PixelSamples 8 8
DisplayChannel "float _inshadow" "quantize" [0 255 0 255] "dither" [0.5]

Display "./shadow_test.tif" "framebuffer" "rgba"
Display "+./shadow_test._inshadow.tif" "tiff" "_inshadow"
Format 400 250 1
Projection "perspective" "fov" 17
ShadingRate 1

Translate  0 -0.05 3
Rotate -30  1 0 0
Rotate  20  0 1 0
Scale 1 1 -1
WorldBegin
    TransformBegin
        Rotate -80 1 0 0
        Scale 1 1 -1
        LightSource "shd_distant" 1 "string shadowname" ["./shd_map_80.tex"]
                    "float intensity" 0.5 "float width" 4 "float samples" 32
    TransformEnd
    TransformBegin
        Rotate -100 1 0 0
        Scale 1 1 -1
        LightSource "shd_distant" 2 "string shadowname" ["./shd_map_100.tex"]
                    "float intensity" 0.5 "float width" 4 "float samples" 32
    TransformEnd

    Surface "diffuseAOV"
    AttributeBegin
```

```
        Polygon "P" [-0.5 0 -0.5  -0.5 0 0.5  0.5 0 0.5  0.5 0 -0.5]
                "st" [0 0  0 1  1 1  1 0]
    AttributeEnd
    AttributeBegin
        Translate 0 0.35 0
        Sphere 0.15 -0.15 0.15 360
    AttributeEnd
WorldEnd
```

If the reader has Pixar's RenderMan Pro-Server and RenderMan Studio installed on their computer they can render both the beauty pass and the secondary images directly to an "it" (Image Tool) catalog. For example,

```
Display "./shadow_test.tif" "it" "rgba"
Display "+./shadow_test._inshadow" "it" "_inshadow"
```

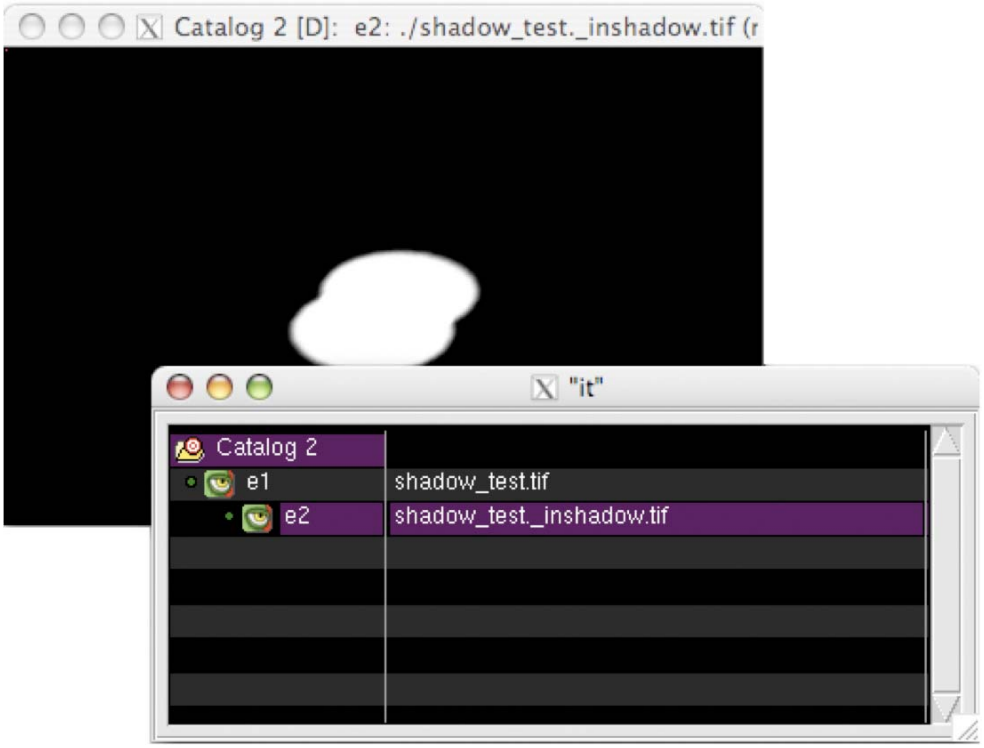


Figure 8

Otherwise, it is necessary to render the secondary image as a image file, for example,

```
Display "./shadow_test" "framebuffer" "rgba"
Display "+./shadow_test._inshadow.tif" "tiff" "_inshadow"
```

Occlusion Directional Light

The next illustration shows the "soft shadow" effect of a directional light that outputs a color based on the use of the `occlusion()` shadeop. The QuickTime movie shown below consists of a 60 frame animation of an occlusion light source, inclined at an angle of 60 degrees, rotating 360 degrees around the Y-axis. The flickering of the occlusion is caused by the compression of the QuickTime movie.

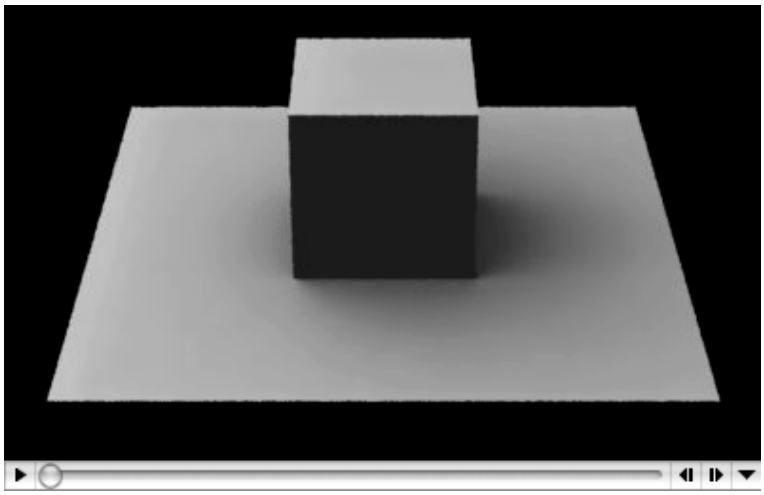


Figure 9

The source code of the light source shader, and a rib file suitable for testing it, is shown in listings 11 and 12.

Listing 11 (occlusionlight.sl)

```
light
occlusionlight(float   intensity = 1,
                       samples = 64,
                       multiplier = 2,
                       coneangle = 90;
               color   lightcolor = 1)
{
vector direction = vector "shader"(0,0,1);
float  occ;
solar(direction, 0.0) {
    occ = 1 - occlusion(Ps, -direction, samples,
                                "coneangle", radians(coneangle));
    occ = pow(occ, multiplier);
    Cl = occ * intensity * lightcolor;
    }
}
```

The `multiplier` and `coneangle` parameters control the density and distribution of the occlusion. However, to avoid artifacts it may be necessary to adjust the trace "bias" in the rib file.

Listing 12 (occlusionlight.rib)

```
Option "searchpath" "shader" "@:../shaders"
Option "searchpath" "texture" "../textures"
Option "searchpath" "archive" "../archives:Cutter_Help/templates/Rib"

Display "occlusionlight" "it" "rgba"
Format 400 240 1
Projection "perspective" "fov" 15
ShadingRate 1

Translate  0 0 3
Rotate -30 1 0 0
```

```
Rotate   0 0 1 0
Scale 1 1 -1
WorldBegin
    Attribute "visibility" "trace" [1]
    LightSource "ambientlight" 1 "intensity" 0.15
    TransformBegin
        Rotate 0 0 1 0
        Rotate -60 1 0 0
        Scale 1 1 -1
        LightSource "occlusionlight" 2 "intensity" 0.8
                    "multiplier" 3 "samples" 1024 "coneangle" 90
    TransformEnd
    Surface "plastic" "Kd" 0.9 "Ks" 0
    AttributeBegin
        Attribute "trace" "float bias" [0.001]
        Scale 0.3 0.3 0.3
        ReadArchive "pCube.rib"
    AttributeEnd
    AttributeBegin
        Polygon "P" [-0.5 0 -0.5  -0.5 0 0.5  0.5 0 0.5  0.5 0 -0.5]
                "st" [0 0  0 1  1 1  1 0]
    AttributeEnd
WorldEnd
```

Rim Lighting

The tutorial "RSL:Edge Effects" demonstrated the use of the dot product to create fake rim lighting. Incidently, the same vector arithmetic is used by Maya's (HyperGraph) "facing ratio" node. Listing 9 (diffuseAOV.sl) also uses the dot product to scale the apparent intensity of the (Lambertan or diffuse) illumination received by a surface. However, the two vectors it uses are not those of the (reversed) view vector and the surface normal but the (reversed) light direction and the surface normal. In other words, what is referred to as Lambertian or diffuse illumination is the result of mulitplying the facing ratio of the light source by its color. Perhaps the illuminance loop used by the diffuseAOV shader should be re-written to make the use of the dot product more apparent ie.

```
illuminance( P, n, PI/2 ) {
    float facing_ratio = normalize(L).n;
    diffusecolor += Cl * facing_ratio;
    }
```

The next shader (listing 13) modifies the output color of a light by the dot product (aka facing ratio) of the surface normal and the view vector. However, because this is implemented by a directional light source the rim effect can be controlled by a lighting artist rather than a shading artist!

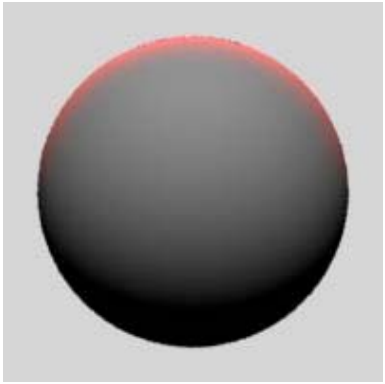


Figure 10

Listing 13 (distantrimlight.sl)

```
light
distantrimlight(float   intensity = 1,
                        width = 0.6;
                color   lightcolor = 1)
{
vector   direction = vector "shader"(0,0,1);
vector   i = normalize(-I);
normal   n = normalize(N);
normal   nf = faceforward(n, I, n);
float    dot = nf.i;
float    rim = smoothstep(1.0 - width, 1.0, 1 - dot);

solar(direction, 0.0) {
    Cl = intensity * lightcolor * rim;
    }
}
```

Light Wrapping

This section is based on the Siggraph 2002 paper, "Renderman on Film" by Rob Bredow (Sony Pictures Imageworks). In his paper the author demonstrates the principles of a shading technique they used in the production of the movie "Stuart Little 2". Their system gave the illusion that illumination from a light source appeared to wrap around the curved surfaces of their characters as if they were lit by an area light source.

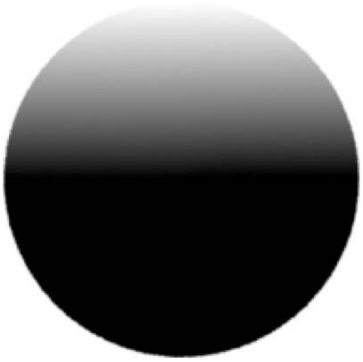


Figure 11
Standard Illumination

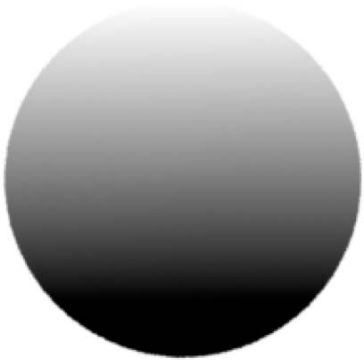


Figure 12
Wrapped Illumination

Although the Siggraph paper explained their light-wrapping technique it did not provide the source code of their custom shaders. Consequently, the shaders given in listings 14 and 15 are, almost certainly, substantially different to those developed at Sony.

The Maths

Figure 13 shows four micropolygons, labeled **a, b, c** and **d**, on a sphere that is illuminated by a downward pointing directional light. The red lines represent vectors pointing **toward** the light source while the green lines represent the surface normals of each micropolygon. The dot product of the vectors and the normals at each of the sampled points is also given along with a "reminder" that the dot product represents the cosine of the angle between the vectors. For convenience, the angles are shown in degrees rather than in radians.

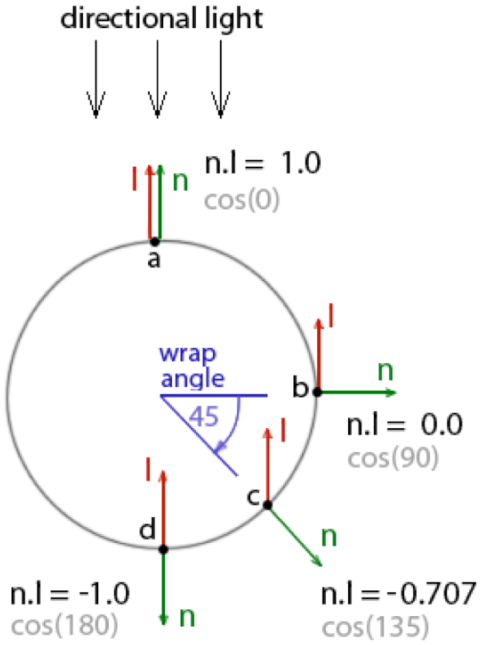


Figure 13

A "regular" surface shader ie. no light wrapping, would assign maximum illumination to the micropolygon at point **a** and minimum illumination to point **b**. The terminator between light and dark would occur along the "equator" of the sphere. However, a surface shader that wraps by, say, 45 degrees would "push" the terminator to point **c**.

The Role of the Surface Shader

Diffuse illumination is calculated by a surface shader either indirectly using the `diffuse()` shadeop or directly within an illuminance loop. For example,

```
color diffusecolor = 0;
illuminance( P, n, PI/2 ) {
    diffusecolor += Cl * normalize(L).n;
    }
```

In figure 13 the "light" vector (`L`) is represented by the normalized `l` vector. An illuminance loop usually samples a hemisphere (`PI/2`) and, as such, the loop skips

those micropolygons that "face away" from a light source. Consequently, negative values of the dot product are avoided. To implement light wrapping, the illuminance loop must perform full spherical sampling (`PI` not `PI/2`). Figure 13 shows that for a wrap angle of 45 degrees the illumination must drop to 0.0 not at **'b'** but instead at **'c'**. The values, derived from the dot product, in the range 1.0 to -0.707, must be remapped to the range 1.0 to 0.0. The illuminance loop shown below performs the required remapping.

```
illuminance("wrapper", P, n, PI ) {
        lightsource("wrapAngle", wrap_angle);
        l = normalize(L);
        float  dot = n.l;
        float  minDot = cos(radians(90 + wrap_angle));
        float  clamped = clamp(dot, minDot , 1.0);
        float  illum = (clamped - minDot)/(1.0 - minDot);
        diffuseColor += Cl * illum;
        }
```

The Role of the Light Source Shader

Although light wrapping must be applied by a surface shader, it makes sense in a studio for the wrap_angle to be determined by the lighting artists ie. controlled by the light sources in a scene. Note that in listing 14 the illuminance loop queries the `__category` name of each light. Only those lights tagged as "wrapper" will be processed within the loop. The light shader in listing 15 has a "dummy" parameter named "wrapAngle". While it is not used by the shader it is present in the parameter list of the light so that it can be queried within the illuminance loop of the surface shader. This enables multiple instances of the `wrapperlight` to apply different "wrappings".

Listing 14 (wrappersurf.sl)

```
surface
wrappersurf(float Ka = 1,
                  Kd = 0.6,
                  Ks = 0.8,
                  roughness = 0.1)
{
normal  n = normalize(N);
vector  l, i = normalize(-I);
color   ambientcolor = ambient() * Ka;
color   diffuseColor = 0, specColor = 0;
float   wrap_angle, dot, minDot, clampedDot, illum;

illuminance("wrapper", P, n, PI ) {
        lightsource("wrapAngle", wrap_angle);
        l = normalize(L);
        dot = n.l;
        minDot = cos(radians(90 + wrap_angle));
        clampedDot = clamp(dot, minDot , 1.0);
        illum = (clampedDot - minDot)/(1.0 - minDot);
        diffuseColor += Cl * illum;
        }

// Handle the non-wrapping lights
illuminance("-wrapper", P, n, PI/2 ) {
```

```
        vector l = normalize(L);
        diffuseColor += Cl * l.n;
        specColor += Cl * specularbrdf(l, n, i, roughness);
        }
diffuseColor *= Kd;
specColor *= Ks;

Oi = Os;
Ci = Oi * Cs * (ambientcolor + diffuseColor + specColor);
}
```

Listing 15 (wrapperlight.sl)

```
light
wrapperlight(float  intensity = 1;
             color  lightcolor = 1;
             string  __category = "wrapper";
             float  wrapAngle = 0)
{
vector direction = vector "shader"(0,0,1);
solar(direction, 0.0) {
   Cl = intensity * lightcolor;
   }
}
```

Listing 16 (wrapper.rib)

```
Option "searchpath" "shader" "@:../shaders"
Option "searchpath" "texture" "../textures"
Option "searchpath" "archive" "../archives:Cutter_Help/templates/Rib"

Display "wrap_test" "framebuffer" "rgba"
Format 400 250 1
Projection "perspective" "fov" 17
ShadingRate 1

Translate  0 0 7
Rotate 0  1 0 0
Rotate 0  0 1 0
Scale 1 1 -1
Imager "background" "background" [1 1 1]
WorldBegin
    TransformBegin
        Rotate -90 1 0 0
        Scale 1 1 -1
        LightSource "wrapperlight" 1 "wrapAngle" 45
    TransformEnd
    TransformBegin
        LightSource "spotlight" 2 "intensity" 5 "from" [1 5 2] "to" [0 0 0]
    TransformEnd
    AttributeBegin
        Surface "wrappersurf"
        Sphere 1 -1 1 360
```

```
    AttributeEnd
WorldEnd
```

Volumetric Fog Effects

This section introduces the use of a very simple volume shader that works in conjunction with a "tagged" custom directional light source. Strickly speaking, the volumetric fog effects shown in figure 14 do not require a custom light. However, assigning a `__category` parameter to the custom light enables instances of the shader to control the effects created by the volume shader. Some what like the techniques used in the previous section, the light source could pass (noise) values to the volume shader for the purpose, say, of making the fog look more interesting.

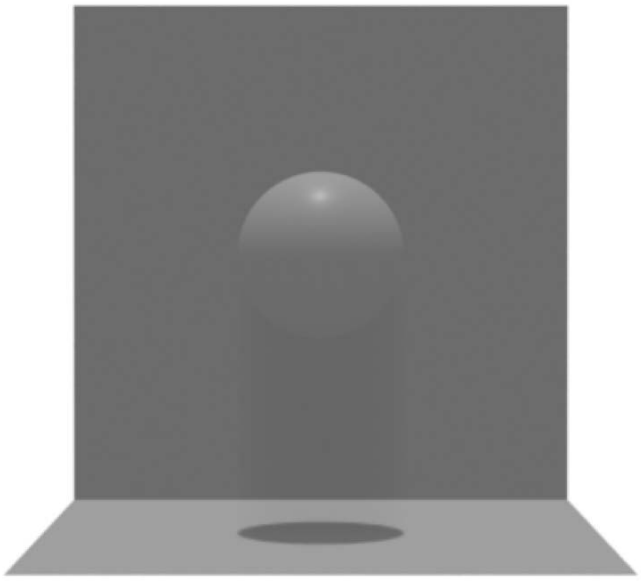


Figure 14

Immediately after a surface shader has determined the apparent color and opacity of a micropolygon a volume shader can modify the micropolygon's color in order to create the illusion that a scene is partially or fully filled with a medium such as water, fog or smoke. For example, the volume shader shown in the next listing steps along the viewing vector (`I`) from point `P` ie. the location of the micropolygon currently being shaded, towards the camera. At each step along the vector it calculates its current position (`currP`) so that the illuminance function can sample the light arriving at that location. An average of the total illumination accumulated along the `I` vector is then added to Ci.

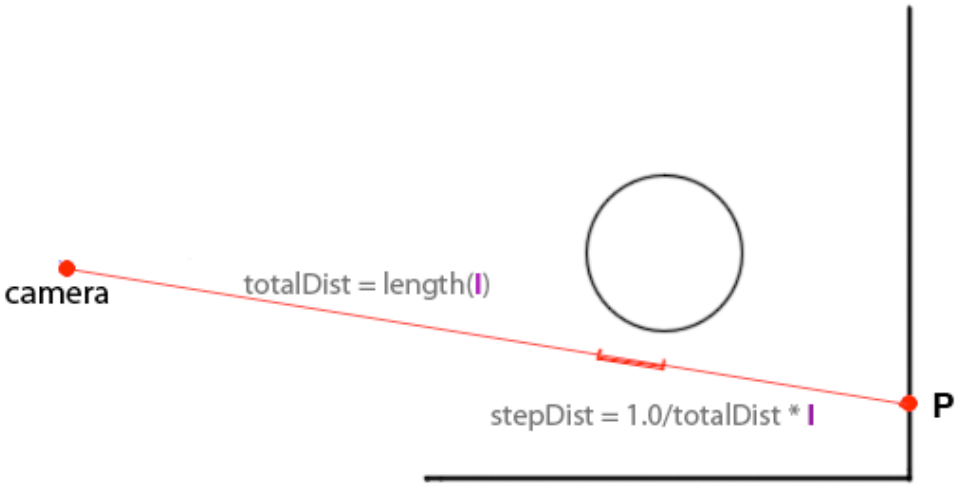


Figure 15

Listing 17 (mkfog.sl)

```
volume mkfog (float step = 0.025)
{
float    totalDist = length(I);
point    currP;
float    stepCount = 1;
color    totalC = 0;

// Jitter the starting point
float     currDist = random() * step;
vector    deltaI = 1.0/totalDist * I;

while(currDist <= totalDist) {
    currP = P - currDist * deltaI;
    illuminance ("foglight", currP) {
        totalC += Cl;
        }
    currDist += step;
    stepCount += 1;
    }
Ci = Ci + totalC/stepCount;
}
```

The code for a light source that works in conjunction with the volume shader is shown next.

Listing 18 (distantFoglight.sl)

```
light distantFoglight(
    string   shadowname = "";
    float    intensity = 1;
    color    lightcolor = 1;
    uniform string __category = "foglight")
{
vector direction = vector "shader" (0,0,1);
```

```
solar(direction, 0.0) {
    Cl = intensity * lightcolor;
    if (shadowname != "")
        Cl *= (1 - shadow(shadowname, Ps, "samples", 64,
                                        "swidth", 1, "twidth", 1));
    }
}
```

The two frame rib file that can be used to test the shaders is given next.

Listing 19

```
FrameBegin 1
    PixelSamples 1 1
    Hider "hidden" "jitter" [0]
    Display "null" "null" "z"
    Display "+./distantshadow2.tex" "deepshad" "deepopacity" "string filter" ["box"]
                                    "float[2] filterwidth" [1 1]
    Format 512 512 1
    Projection "orthographic"
    ShadingRate 1

    Translate  0 0 5
    Rotate -90 1 0 0
    Scale 1 1 -1
    WorldBegin
        AttributeBegin
            Translate 0 0.5 -0.5
            Rotate 90 1 0 0
            Polygon "P" [-0.5 0 -0.5  -0.5 0 0.5  0.5 0 0.5  0.5 0 -0.5]
        AttributeEnd
        AttributeBegin
            Translate 0 0.5 0
            Sphere 0.15 -0.15 0.15 360
        AttributeEnd
        AttributeBegin
            Polygon "P" [-0.5 0 -0.5  -0.5 0 0.5  0.5 0 0.5  0.5 0 -0.5]
        AttributeEnd
    WorldEnd
FrameEnd
FrameBegin 2
    Option "shadow" "float bias" [0.1]

    Display "./volume_test" "it" "rgba"
    Format 350 350 1
    Projection "perspective" "fov" 17
    ShadingRate 1

    Translate  0 -0.5 4
    Rotate -0.5  1 0 0
    Rotate  0  0 1 0
    Scale 1 1 -1
    WorldBegin
        Atmosphere "mkfog" "float step" 0.025
        TransformBegin
```

```
            Translate  0 0 0
            Rotate -90 1 0 0
            Scale 1 1 -1
            LightSource "distantFoglight" 2 "intensity" 0.35
                        "string shadowname" ["./distantshadow2.tex"]
        TransformEnd
        Surface "plastic" "Kd" 0.6
        AttributeBegin
            Translate 0 0.5 -0.5
            Rotate 90 1 0 0
            Polygon "P" [-0.5 0 -0.5  -0.5 0 0.5  0.5 0 0.5  0.5 0 -0.5]
        AttributeEnd
        AttributeBegin
            Translate 0 0.5 0
            Sphere 0.15 -0.15 0.15 360
        AttributeEnd
        AttributeBegin
            Polygon "P" [-0.5 0 -0.5  -0.5 0 0.5  0.5 0 0.5  0.5 0 -0.5]
        AttributeEnd
    WorldEnd
FrameEnd
```

RSL
Writing Displacement Shaders

Overview

Displacement shaders alter the smoothness of a surface, however, unlike bump mapping which mimics the appearance of bumpiness by reorientating surface normals, displacement shading genuinly effects the geometry of a surface. In the case of Pixars prman renderer, each object in a 3D scene is sub-divided into a fine mesh of micro-polygons after which, if a displacement shader has been assigned to an object, each micro-polygon is "pushed" or "pulled" in a direction that is parallel to the original surface normal of the micro-polygon. After displacing the micro-polygon the orientation of the local surface normal (`N`) is recalculated.

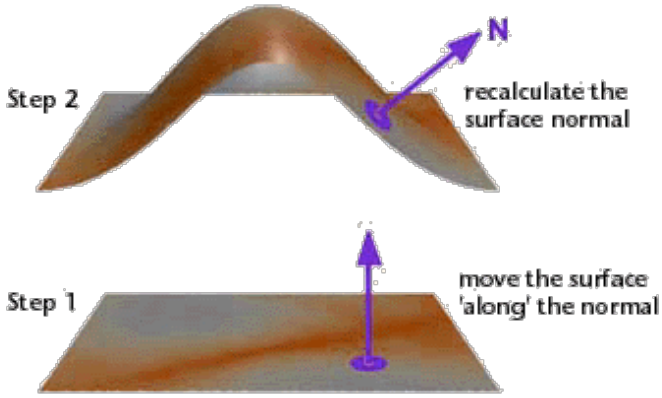


Figure 1

The following algorithm lists the four basic steps that a displacement shader generally follows in order to set the position (`P`) and normal (`N`) of the micro-polygon being shaded.

1	Make a copy of the surface normal (`N`) ensuring it is one unit in length.
2	Calculate an appropriate value for the displacement - what will be referred to in these notes as the `hump` factor!
3	Calculate a new position of the surface point "`P`" by moving it "along" the copy of the surface normal by an amount equal to `hump` scaled by the value of the instance variable `Km`.
4	Recalculate the surface normal (`N`).

To make a meaningful decision about the distance, if any, a micro-polygon should be displaced, a shader may make reference to the micro-polygon's,

- 2D surface position `s`, `t`, `u`, `v`,

- 3D xyz position `P`,
- orientation `N`,
- camera distance `L`.

plus other less obvious attributes of a micro-polygon. Such information is either directly or indirectly available in data the renderer makes available to a shader through the use of global variables.

Displacement Shaders & Global Variables

The following table lists the global variables accessible to a displacement shader. For the corresponding list of global variables available to a surface shader refer to the tutorial "RSL: What is a Surface Shader".

Global variable	Meaning
P	surface position
N	surface geometric normal
s, t	surface texture coordinates
Ng	surface geometric normal
u,v	surface parameters
du, dv	change in u, v across the surface
dPdu,dPdv	change in position with u and v
I	camera viewing direction
E	position of the camera

Using Cutter for Shader Writing

It is highly recommended the reader use Cutter for their shader writing. It has many very useful time saving features. Refer to the tutorial "Cutter: Shader Writing" for information about Cutter and how it should be set up.

Basic Code

The experiments on displacement shading in this tutorial are based the shader shown in listing 1.

Listing 1

```
displacement
test1(float Km = 0.1)
{
float   hump = 0;
normal  n;

/* STEP 1 - make a copy of the surface normal */
n = normalize(N);

/* STEP 2 - calculate the displacement */
hump = 0;

/* STEP 3 - assign the displacement to P */
```

```
P = P - n * hump * Km;

/* STEP 4 - recalculate the surface normal */
N = calculatenormal(P);
}
```

Texture Coordinates

Although micro-polygons have 3D xyz positions, given by the global variable `P`, they also have a 2D position in '`st`' texture space. Irrespective of their actual size, nurbs and quadric surfaces cover exactly 1 unit in 's' and 't'.

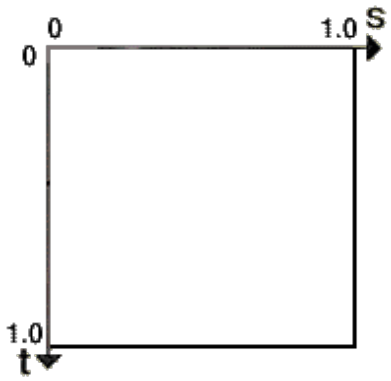


Figure 2

Use Cutter's Rman Tool palette to generate a rib file to test your shaders - figure 3. The poly-plane is set up to also cover one unit in texture space ie.

```
Polygon "P"  [-0.5 0 -0.5  -0.5 0 0.5  0.5 0 0.5  0.5 0 -0.5]
        "st" [0 0  0 1  1 1  1 0]
```

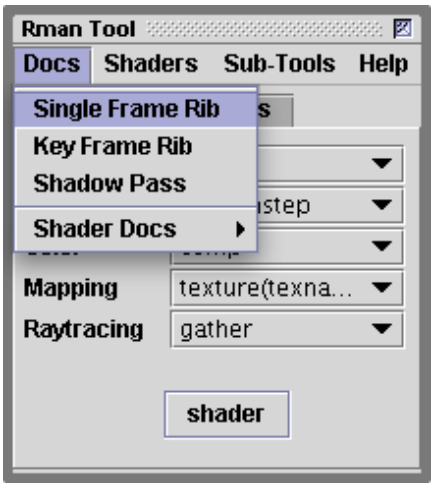


Figure 3

The first shader, listing 2, uses a simple "`if`" test to decide whether a micro-polygon is within a narrow band.

Listing 2

```
displacement
test2(float Km = 0.1)
{
float   hump = 0;
normal  n = normalize(N);

if(t >= 0.4 && t <= 0.6)
    hump = 1;

P = P - n * hump * Km;
N = calculatenormal(P);
}
```

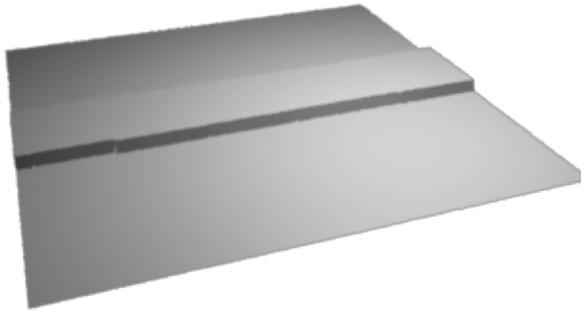


Figure 4

The edge of the raised band is aliased. For the moment we will ignore the defect. The next shader uses the RSL `distance()` function to determine if a micro-polygon is within a distance defined by the shader parameter `radius`.

Listing 3

```
displacement
test3(float Km = 0.1,
            radius = 0.3)
{
float   hump = 0;
normal  n = normalize(N);
float   d = distance(point(0.5,0.5,0), point(s, t, 0));

if(d <= radius)
    hump = 1;

P = P - n * hump * Km;
N = calculatenormal(P);
}
```

Again, the aliased rim of the circle will be ignored.

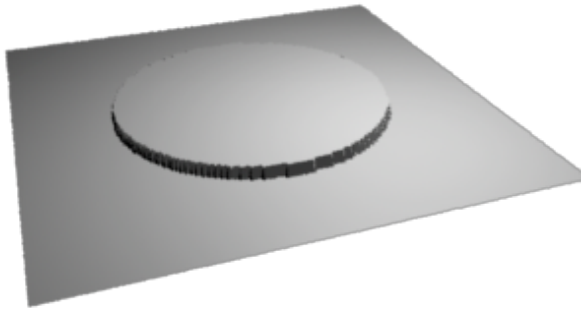


Figure 5

Smoothstep

The shader in listing 4 uses the RSL `smoothstep()` function to soften the edge of the circular displacement. It also provides an extra shader parameter to control the width of the softening. For more information about the use of the `smoothstep()` function refer to the tutorial "RSL: Using Smoothstep".

Listing 4

```
displacement
test4(float Km = 0.1,
            radius = 0.3,
            blur = 0.04)
{
float   hump = 0;
normal  n = normalize(N);
float    d = distance(point(0.5,0.5,0), point(s, t, 0));

hump = 1 - smoothstep(radius - blur, radius + blur, d);

P = P - n * hump * Km;
N = calculatenormal(P);
}
```

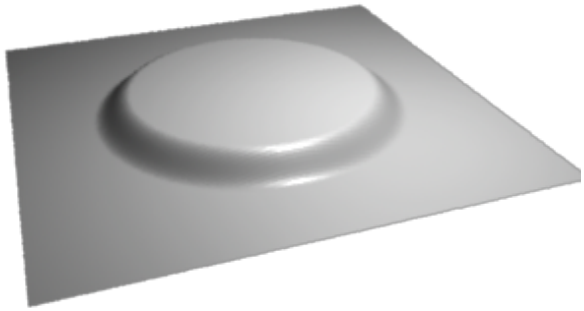


Figure 6

Listing 5 provides additional parameters, s_center and t_center, to control the placement of the circle.

Listing 5

```
displacement
test5(float Km = 0.1,
            radius = 0.1,
            blur = 0.04,
            s_center = 0.25,
            t_center = 0.25)
{
float   hump = 0;
normal  n = normalize(N);
float   d = distance(point(s_center,t_center,0),point(s,t,0));

hump = 1 - smoothstep(radius - blur, radius + blur, d);

P = P - n * hump * Km;
N = calculatenormal(P);
}
```

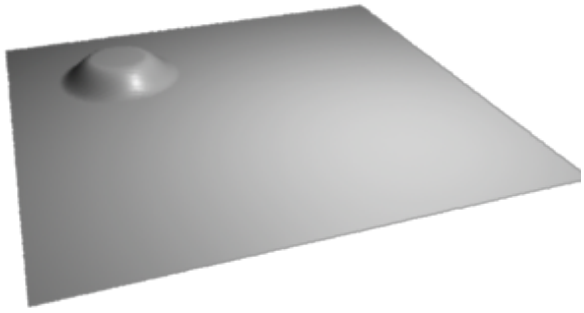


Figure 7

Displacement Mapping

The shader in listing 6 implements simple image embossing. Although the `texture`() can return a float the value represents only the red channel of the image. Unfortunately, it is not an average of the "rgb" channels. For this reason the shader calculates the average "rgb" value. Strickly speaking, the shader should calculate the grayscale value but taking a simple average is good enough.

Listing 6

```
displacement
test6(float Km = 0.1;
      string texname = "")
{
float   hump = 0;
normal  n = normalize(N);

if(texname != "") {
    color c = texture(texname);
```

```
        hump = (comp(c, 0) + comp(c, 1) + comp(c, 2))/3;
        }

    P = P - n * hump * Km;
    N = calculatenormal(P);
    }
```

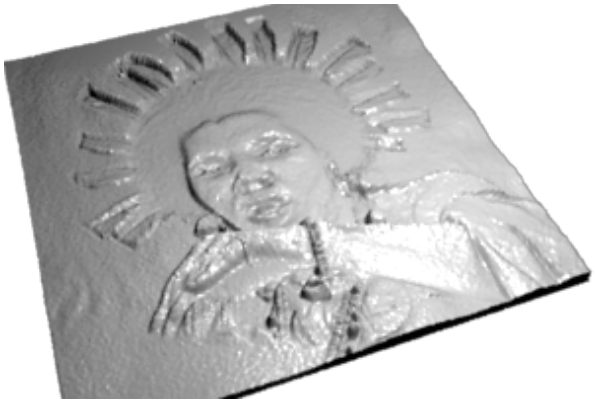


Figure 8

The shader was used in the rib file in the following way.

```
    Displacement "test6" "texname" ["swazi.tx"] "Km" -0.20
```

The texture file "swazi.tex" was converted from the image shown in figure 9. For more information about converting tif files to textures refer to the tutorial "Writing Surface Shaders".

Figure 9

Noise I

The next shader uses the RSL `noise()` function to create a bumpy surface. For more information about this function refer to the tutorials "Using Noise" and "Writing Surface Shaders".

Listing 7

```
displacement
test7(float Km = 0.1,
            s_freq = 6,
            t_freq = 8)
{
float   hump = 0;
normal  n = normalize(N);

hump = noise(s * s_freq, t * t_freq);

P = P - n * hump * Km;
N = calculatenormal(P);
}
```

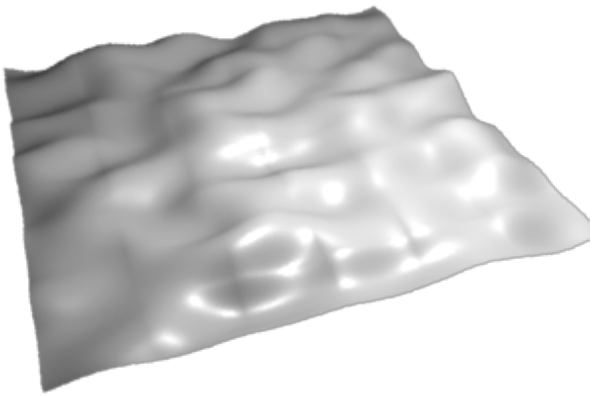


Figure 10

Because the inputs to the `noise()` function are '`s`' and '`t`' the bumps are "parented" to the texture space of the surface and as such will move with the object. In other words the bumps will not appear to slide or move over the surface. If movement of the bumps is required it can be done in two ways. Listings 8 and 9 address this issue.

Animated 'st' Noise

The shader in listing 8 applies an offset to the '`s`' and '`t`' values before they are scaled by their respective frequency parameters. The bumps can be animated incrementely increasing the `s_offset` and/or `t_offset` on a frame-by-frame basis. For information about Cutter's keyframing capabilities refer to the tutorial "KeyFraming"

Listing 8

```
displacement
test8(float Km = 0.1,
            s_freq = 6,
            s_offset = 0,
            t_freq = 8,
            t_offset = 0)
{
float   hump = 0;
normal  n = normalize(N);
```

```
hump = noise((s - s_offset) * s_freq,
             (t - t_offset) * t_freq);

P = P - n * hump * Km;
N = calculatenormal(P);
}
```

The banding seen in figures 11 and 12 are caused by a defect in the (Perlin) noise function. In theory the displacements should be smooth in all directions but there are discontinuities at the integer lattice. The defect is particularly noticable with large displacements.

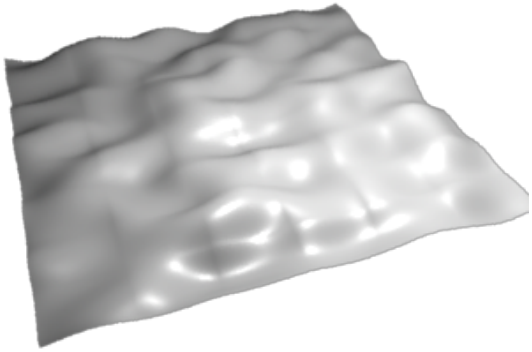


Figure 11
`s_offset` = 0.0

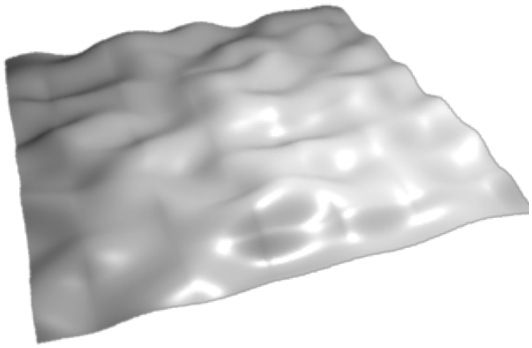


Figure 12
`s_offset` = 0.1

Animated 3D Noise

The shader in listing 9 uses a micro-polygons xyz position (`P`) as an input to `noise`(). Unlike the previous shader in listing 8 that supplied two inputs, and hence produced 2D noise, the current shader generates true 3D noise. The shader was applied to a cubic stack of poly-planes from which a spherical hole "gouged out" with a special purpose surface shader. The variations in displacement caused by the 3D noise can clearly be seen.

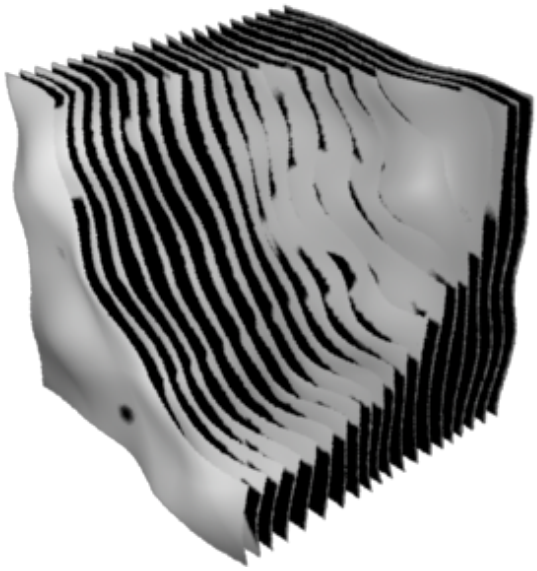


Figure 13

Listing 9

```
displacement
test9(float Km = 0.1,
            freq = 1)
{
float   hump = 0;
normal  n = normalize(N);

hump = noise(P * freq);

P = P - n * hump * Km;
N = calculatenormal(P);
}
```

The principle issue with the shader is that point '`P`' is defined in "camera space". Consequently, the noise is "parented" to the camera - movements of the camera will move the noise! Refer to the tutorial "Writing Surface Shaders" for a more information about coordinate systems.

To ensure an artist has control over 3D noise the next shader enables point '`P`' to be transformed into either a pre-existing or a user-defined coordinate system. The pre-existing coordinate systems are "camera", "world", "object" and "shader". However, as shown next a user-defined coordinate system can be established with the `CoordinateSystem` rib statement.

Listing 10a

```
displacement
test10(float Km = 0.1,
            freq = 1;
    string   space = "object")
{
float   hump = 0;
normal  n = normalize(N);

point    p = transform(space, P);
hump = noise(p * freq);

P = P - n * hump * Km;
N = calculatenormal(P);
}
```

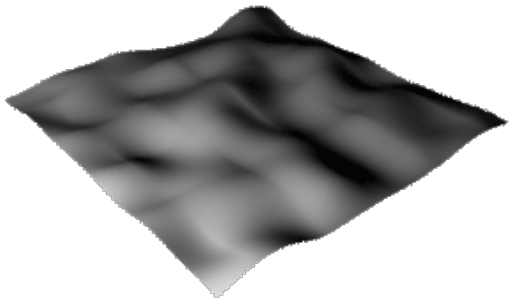


Figure 14

```
Displacement "test10" "space" ["object"]
```

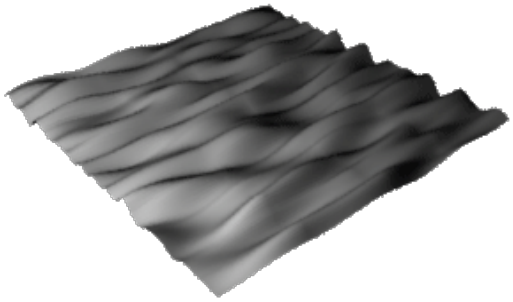


Figure 15

```
Displacement "test10" "space" ["myspace"]
```

The rib file used to render figure 15 defined a user-defined coordinate system as follows.

Listing 10b

```
TransformBegin
    Translate 0 0 0
    Rotate  0 1 0 0
    Rotate  0 0 1 0
    Rotate  0 0 0 1
    Scale   0.25 1 1
    CoordinateSystem "myspace"
TransformEnd
Displacement "test10" "space" ["myspace"] "Km" -0.50 "freq" 1
```

Turbulance

A simulation of turbulance or fractal noise can be achieved by using the noise() within a loop. On each iteration of the loop the value returned from noise() is added to the result of the previous iteration. Successfully higher frequencies but smaller amplitudes are used for iteration. The visual result is richer because the shading can appear to mimic natural surfaces ie. large bumps have small bumps which in

turn have enen smaller

Listing 11a

```
displacement
test11a(float Km = 0.1,
              freq = 1,
              layers = 3;
    string   space = "object")
{
float   hump = 0;
normal  n = normalize(N);
point   p = transform(space, P);
float   j, f = freq, amplitude = 1;

for(j = 0; j < layers; j += 1) {
    hump += noise(p * f) * amplitude;
    f *= 2;
    amplitude *= 0.5;
    }

P = P - n * hump * Km;
N = calculatenormal(P);
}
```

The problem with applying a displacement directly with the value returned from `noise`() is that the displaced surface moves away from its original position. This "side-effect" is made worse when a number of displacements are summed. For example, in figure 16 the lower poly-plane marks the starting position for the displaced polygon. In figure 17 an adjustment has been made to the shader so that on average the displaced surface is 50% above and below its original location.

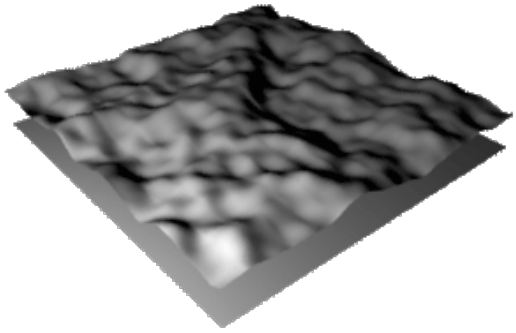


Figure 16

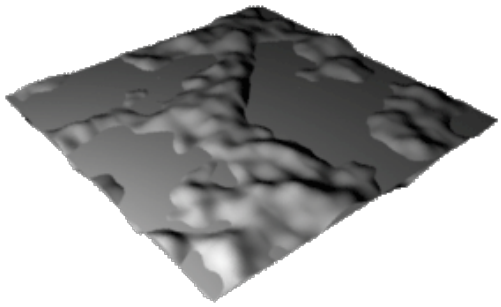


Figure 17

In listing 11b a constant value of 0.5 is substracted from the noise value. In general it a good idea to always subtract 0.5 from noise.

Listing 11b

```
displacement
test11b(float Km = 0.1,
```

```
                freq = 1,
                layers = 3;
    string      space = "object")
{
float   hump = 0;
normal  n = normalize(N);
point   p = transform(space, P);
float   j, f = freq, amplitude = 1;

for(j = 0; j < layers; j += 1) {
    hump += (noise(p * f) - 0.5) * amplitude;
    f *= 2;
    amplitude *= 0.5;
    }

P = P - n * hump * Km;
N = calculatenormal(P);
}
```

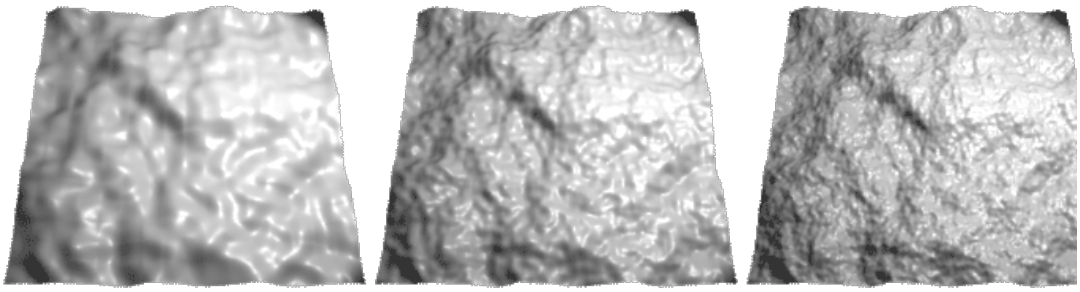


Figure 18
From left to right - "layers" 3, "layers" 4 and "layers" 5

Some interesting visual effects can also be created by ensuring the value returned fr
noise() is always positive. Listing 11c uses the `abs`() function to create the effect see
in figure 19.

Listing 11c

```
displacement
test11c(float Km = 0.1,
                freq = 1,
                layers = 3;
    string      space = "object")
{
float   hump = 0;
normal  n = normalize(N);
point   p = transform(space, P);
float   j, f = freq, amplitude = 1;

for(j = 0; j < layers; j += 1) {
    hump += abs(noise(p * f) - 0.5) * amplitude;
    f *= 2;
    amplitude *= 0.5;
```

```
    }

P = P - n * hump * Km;
N = calculatenormal(P);
}
```

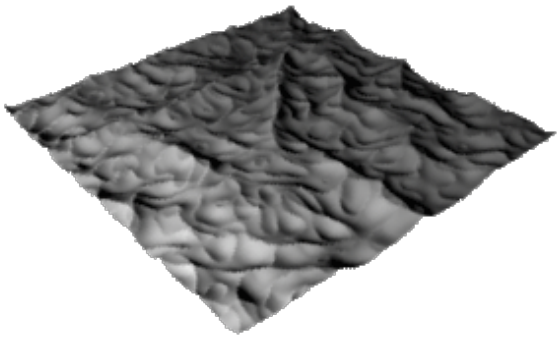


Figure 19

Ripples

Listing 3 demonstrated the use of the RSL `distance()` function to calculate the distance between to points. Figure 20 shows a method that uses the theorem of Pythagoras to also calculate the straight line distance between two points. One point is defined by the coordinates `a,b` and the other by `s,t`. In listing 12 the first coordinates will define the center of a ripple while the second coordinates are those for the micro-polygon that is being shaded.

Listing 12a

```
displacement ripple1(float Km = 0.03,
                             numripples = 8,
                             a = 0.3,
                             b = 0.25)
{
float  sdist = s - a,
       tdist = t - b,
       dist = sqrt(sdist * sdist + tdist * tdist),
       hump = sin(dist  * 2 * PI * numripples);

normal n = normalize(N);

P = P - n * hump * Km;
N = calculatenormal(P);
}
```

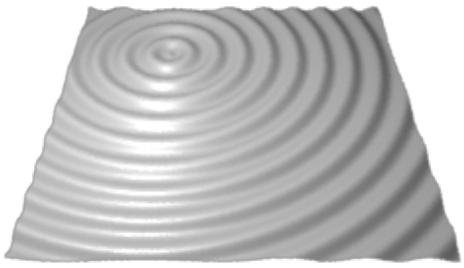


Figure 20

Ripples in a pool of water, say as the result of a drop of rain, normally propogate outward as 2 or 3 concentric waves. Listing 12b applies a constraint on the ripples seen in figure 20 in order to mimic the rain-drop effect. The constraint is based on a double use of the smoothstep() function. For more information about this RSL function refer to the tutorial "RSL: Using smoothstep".

Listing 12b

```
displacement ripple1(float Km = 0.03,
                             numWaves = 12,
                             a = 0.3,
                             b = 0.25,
                             rippleRad = .5,
                             rippleWidth = 0,
                             rippleFade = 0.13)
{
float  sdist = s - a,
       tdist = t - b,
       dist = sqrt(sdist * sdist + tdist * tdist),
       hump = sin(dist  * 2 * PI * numWaves);

float  w = rippleWidth/2;
float  inner = rippleRad - w;
float  outer = rippleRad + w;
hump = hump * smoothstep(inner - rippleFade, inner, dist) *
            (1 - smoothstep(outer, outer + rippleFade, dist));
normal n = normalize(N);
P = P - n * hump * Km;
N = calculatenormal(P);
}
```

Figure 21

RSL
Using smoothstep

Introduction

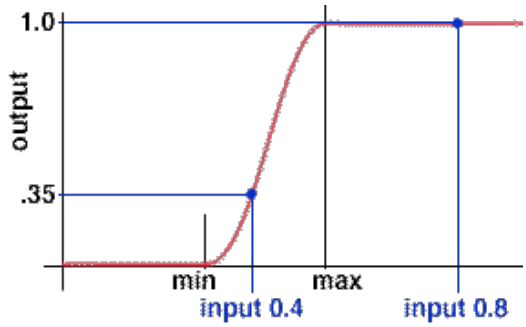
The function `smoothstep()` is part of the maths library of shading language functions. Given three values, `min`, `max` and `input`, the function will return a number between 0 and 1 that represents the relationship of the `input` value to the `min` and `max` values.

If **input** is less than `min`, smoothstep() will return 0.
If **input** is equal to, or larger than `max`, smoothstep() will return 1.
If **input** is between `min` and `max`, smoothstep() will return a value (proportionately) between 0 and 1.0.

For example, suppose the min and max values are 0.3 and 0.6. Using the smoothstep function with input values of 0.4 and 0.8 we get output values of 0.35 and 1.0.

```
smoothstep(0.3, 0.8, 0.4);
```

```
smoothstep(0.3, 0.8, 0.8);
```



Applying Smoothstep

As shown on the right the smoothstep function can be used to control the blending colors or a displacement. The code for the displacement shader is shown in listing 1.

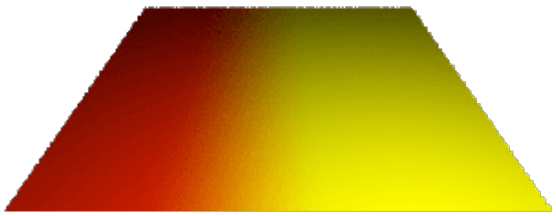


Figure 1 smoothcolor.sl

Figure 2 smoothbump.sl

Listing 1

```
displacement
smoothbump(float  Km = 0.1,
                  min = 0.3,
                  max = 0.8)
{
float    hump = smoothstep(min, max, t);
normal   n = normalize(N);

P = P - n * hump * Km;
N = calculatenormal(P);
}
```

Combining Smoothsteps - part 1

Often a shader must control a blending factor by smoothly **increasing** and **decreasing** an effect. For example, in the case of the displaced cylinder the **min** and **max** values might be used to define locations where a displacement is ramped-up then ramped-down.

The trick here is to notice that subtracting the values returned from the smoothstep() function from 1.0 has the effect of inverting its effect ie. the output values decrease from 1.0 to 0. A combined blending effect can be obtained by,

```
blend = smoothstep(0.2, 0.3, t) * (1 - smoothstep(0.6, 0.7, t);
```

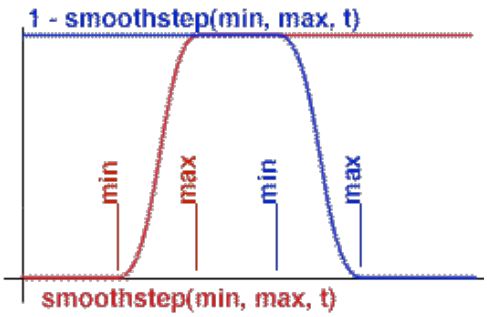


Figure 3

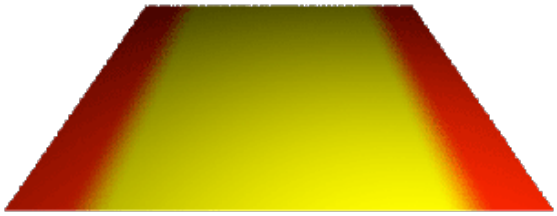


Figure 4 smoothcolor2.sl

Figure 5 smoothbump2.sl

Combining Smoothsteps - part 2

The double ramping seen in the previous section can also be achieved in two directions, say, in '`s`' and the '`t`'.

```
blend = smoothstep(0.2, 0.3, s) * (1 - smoothstep(0.6, 0.7, s) *
        smoothstep(0.2, 0.3, t) * (1 - smoothstep(0.6, 0.7, t);
```

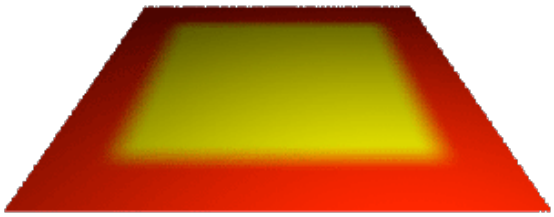


Figure 6 smoothcolor3.sl

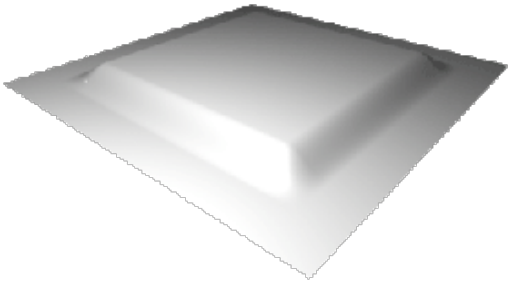


Figure 7 smoothbump3.sl

Variety

The shading language incorporates many useful maths functions that can be used to generate visual effects such as bumpiness and variations in color. Maths functions, however, produce visual effects that look unnaturally smooth and regular. The `noise`() function can be used to add variations to a visual effect. The noise function can be considered to be a black-box number generator. Irrespective of the magnitude of its inputs, the output values from the `noise`() function are, in theory, always in the range 0 to 1. In practice the output values are generally in the range 0.27 to 0.7.

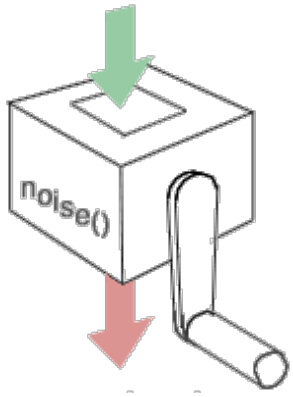


Figure 1 - the "noise" machine!

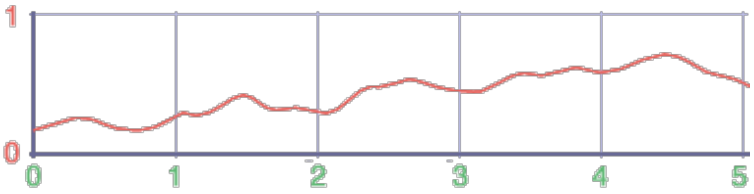


Figure 2 - a plot of noise values

Frequency & Amplitude

In a displacement shader noise might be used to effect the bumpiness of a surface in say the '`s`' direction, for example,

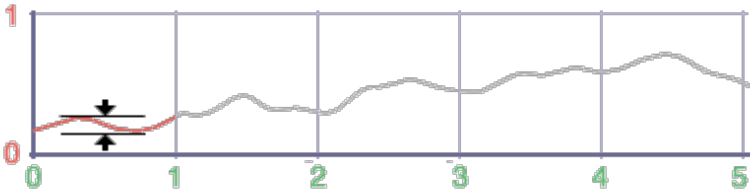


Figure 3

Because the default values of 's' range from 0 to 1 the resulting frequency of the noise is low. This can been seen in figure 3 where there is only a couple of peaks and valleys along the first part of the line segment. A poly plane displaced by,

```
hump = noise(s);
```

is shown in figure 4.

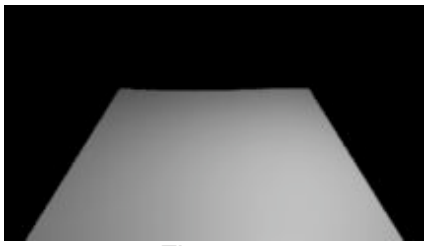


Figure 4

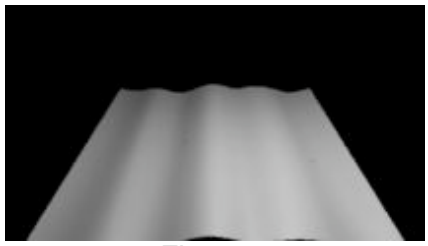
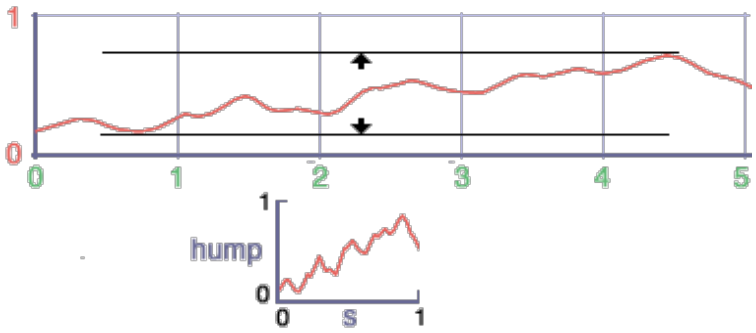


Figure 5

To increase the frequency of the output values (from the noise function) the input values are scaled, for example,



A poly plane displaced by,

```
hump = noise(s * 5);
```

is shown in figure 5. To control the **amplitude** of the noise its output value can be scaled. In the line of code shown below the difference in height between the valleys and peaks is reduced by approximately a third ie.

```
hump = noise(s * 5) * 0.3;
```

Figures 4 and 5 are examples of **one-dimensional noise.**

2D Noise

The `noise()` function also accepts two input values, for example,

```
hump = noise(s * 5, t * 5) * 0.3;
```

Using two values generates **two-dimensional noise**.

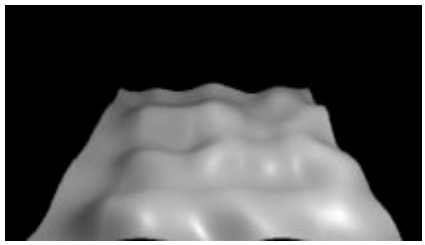


Figure 6

Listing 1 shows that replacing the constant values (5 and 0.3) with the instance variables, `freq` and `amp`, makes the shader more flexible because those parameters can be set in the rib file.

Listing 1

```
displacement
basic_2d_noise(float Km = 0.1,
                     freq = 5,
                     amp = 1)
{
float  hump = 0;
point  n = normalize(N);

hump = noise(s * freq, t * freq) * amp;

P = P - n * hump * Km;
N = calculatenormal(P);
}
```

3D Noise

The noise function also accepts a three-dimensional input. For example, we could use the global variable `P` that references the xyz coordinates of the point that is currently being displaced by a shader.

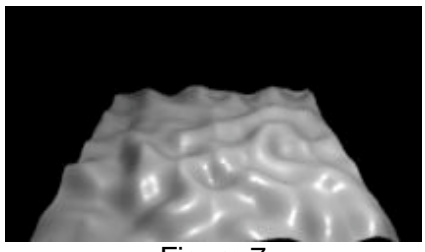


Figure 7

Listing 2

```
displacement
basic_3d_noise(float Km = 0.1,
                     freq = 5,
                     amp = 1)
{
float  hump = 0;
```

```
point  n = normalize(N);

hump = noise(P * freq) * amp;

P = P - n * hump * Km;
N = calculatenormal(P);
}
```

Sticky 3D Noise

A problem arises when using the global variable `P` as a source of data because its xyz coordinates are, by default, measured relative to the camera. In other words, `P` is in the **camera coordinate system**. Therefore, changing the distance or orientation of the world relative to camera causes the surface to appear to slide over stationary bumps. Three frames of an animation are shown in figure 8. Note that despite the rotation of the polyplane the "feature" shown in red remains fixed in the same position relative to the picture frame. In other words the noise is, in effect, parented to the camera!

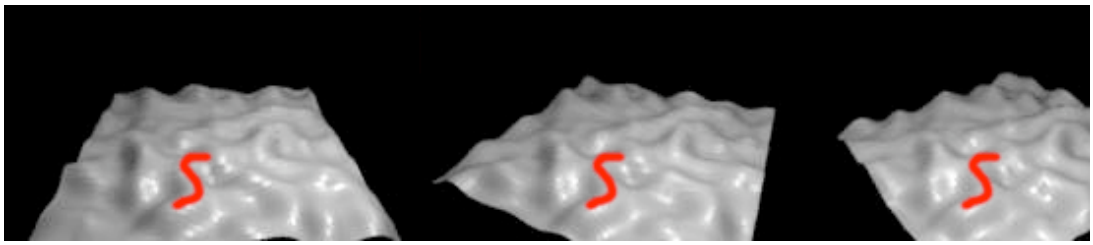


Figure 8

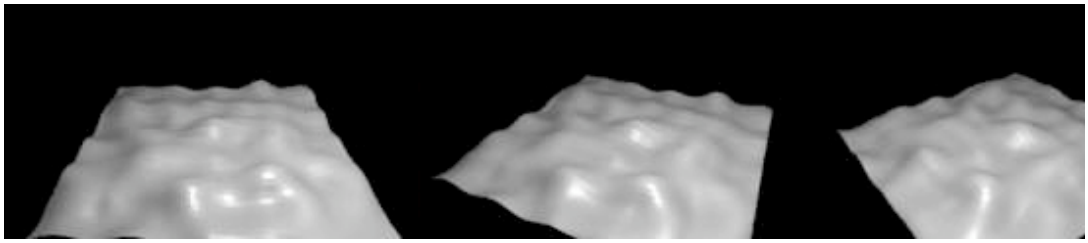


Figure 9

To make the displacements "stick" to the surface of the polyplane, point `P` should **not** be used directly but instead a copy of its data, transformed into "object" or "shader" (coordinate system) space, is used instead. Listing 3 gives an example of how this is accomplished.

Listing 3

```
displacement
parented_3d_noise(float Km = 0.1,
                        freq = 5,
                        amp = 1;
                  string space = "object")
{
float  hump = 0;
```

```
point  n = normalize(N),
pp = transform(space, P);

hump = noise(pp * freq) * amp;

P = P - n * hump * Km;
N = calculatenormal(P);
}
```

Figure 9 shows three frames of an animation in which the displacement shader is using "object" space. In effect the displacements are parented to the polyplane.

RSL
Using Cellnoise

Introduction

Pages 255 to 261 of "Advanced RenderMan" by Gritz and Apodaca provide an introduction to the RSL function **cellnoise()**. This note attempts to explain how Gritz and Apodaca are using cellnoise to create solid textures.

Basic Code

The sample code used for this tutorial consists of the function and surface shader given in listing 1. The function `dist2cell`() is almost identical to the code on pages 257 and 258 of the "Advanced RenderMan" book. The function shown in listing 1 differs in that it provides a parameter that enables a user-specified coordinate system to control `cellnoise`(). The `dist2cell`() function first transforms the xyz position (`p`) of the micro-polygon being shaded into what ever `spacename` is passed to the function from the shader. A nested for-loop finds the distance to the centers of cubes that form a 3x3x3 lattice of imaginary "cells" in the neighborhood of point `p`. The function returns the distance to the center of the nearest cell. Note that in listing 1 because the call to `cellnoise`() is commented the nearest cell is always the central cube in the lattice ie. the "cell" (or cube) in which point `p` is located.

The `cell_test` shader determines if the value returned from the function is within a user-defined distance called `shape_rad`. If the value exceeds `shape_rad` the micro-polygon being shaded is made fully transparent. Before cellnoise() is activated the shader is little more than a rather uninteresting cookie-cutter. This will not be the case after `cellnoise`() is activated.

Listing 1

```
float dist2cell(point p; string spacename; float freq)
{
point pp = transform(spacename, p) * freq;
point thiscell = point(floor(xcomp(pp)) + 0.5,
                       floor(ycomp(pp)) + 0.5,
                       floor(zcomp(pp)) + 0.5);
float dist2nearest = 1000;
uniform float i,j,k;
for(i = -1; i <= 1; i+= 1)
   for(j = -1; j <= 1; j+= 1)
       for(k = -1; k <= 1; k+= 1)
          {
          point testcell = thiscell + vector(i,j,k);
          point pos = testcell;
```

```
            // + vector cellnoise(testcell) - 0.5;
          float dist = distance(pos,pp);
          if(dist < dist2nearest)
             dist2nearest = dist;
          }
return dist2nearest;
}
//-------------------------------------------------------
surface
cell_test(float Kd = 1,
                cellfreq = 5,
                shape_rad = 0.5,
                shape_freq = 8,
                shape_amp = 1,
                mindist = 10,
                maxdist = 15;
         string spacename = "shader")
{
color surfcolor = 1;
normal n = normalize(N);
normal nf = faceforward(n, I);

float d = dist2cell(P, spacename, cellfreq);
float rad = shape_rad;
if(d <= rad)
    Oi = Os;
else
    Oi = 0;
surfcolor = 1 - smoothstep(mindist, maxdist, length(I));

color diffusecolor = Kd * diffuse(nf);
Ci = Oi * Cs * surfcolor * diffusecolor;
}
```

The function, with the code shown in comments ie.

```
   point pos = testcell;// + vector cellnoise(testcell) - 0.5;
```

was used by the surface shader to assign transparency/opacity to a criss-crossed stack of poly-planes shown in figure 1. As can be seen the opaque spheres are aligned to the centers of the matrix of imaginary cells.

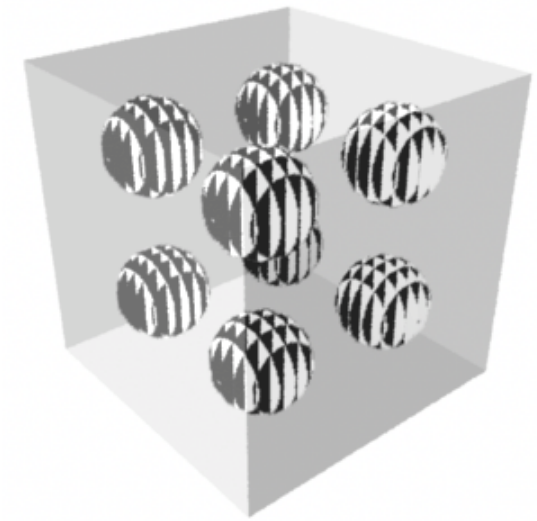


Figure 1

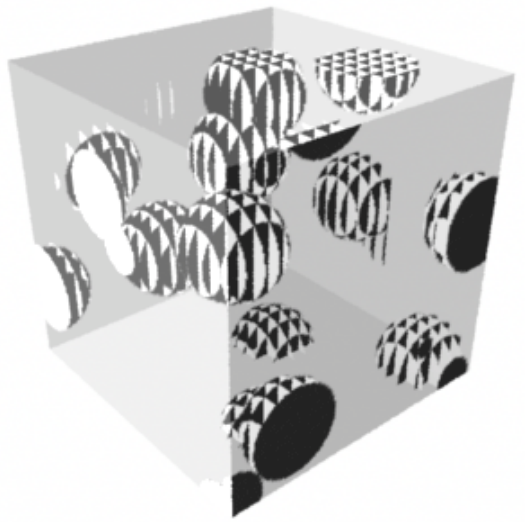


Figure 2

Figure 2 was rendered with cellnoise() activated ie.

```
point pos = testcell + vector cellnoise(testcell) - 0.5;
```

As can be clearly seen in figure 2, `cellnoise`() has had the effect of jittering the centers of the lattice of cells. As a consequence the pattern of spheres has become irregular.

RSL
Introduction to Class-Based Shaders

Introduction

This tutorial provides an introduction to the writing of class-based shaders. The notes were prepared using prman 13.5.2. The primary source of information on this topic is the Pixar document,

`DOCS/prman_technical_rendering/AppNotes/`**`ShaderObjects.html`**

That document refers to enhancements to the shading language that enable shaders to be written in an **object oriented** style of programming (OOP). Because the term "`object`" can be generically applied to (almost) anything stored in the memory of a computer the combination of new and existing RSL terminology can be confusing. This tutorial refers to "traditional shaders" and "class-based shaders" in an attempt to distinguish the older shader programming techniques from the new OOP style in which a shader is "wrapped" or "packaged" within a class.

For the purpose of demonstrating the basics of the object oriented features of the RenderMan Shading Language, two shaders from the tutorial "RSL: Shader to Shader Messaging" are used as "starting points" for the development of a couple of variations of a class-based shader.

Basic Code - Traditional Shaders

The combined effect of the shaders in listings 1 and 2 are shown below. The displacement shader, `hills.sl`, assigns bumpiness to an object. The surface shader, `snow.sl`, based on its use of the Rsl `displacement`() function to querry the value of the `hump` variable of the displacement shader, decides which color to assign to the surface. Both shaders have deliberatrly been kept simple so that the reader can more easily see the correspondence between the original code of listings 1 and 2 and its later use in the class-based shaders.

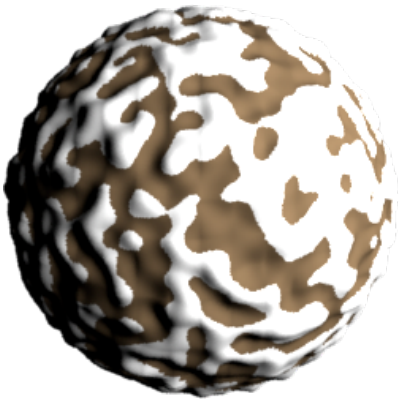


Figure 1

Listings 1 and 2 (hills.sl, snow.sl)

```
displacement hills(float  Km = -0.1,
                          Kf = 8;
    output varying float  hump = 0)
{
normal n = normalize(N);

hump = noise(transform("shader",P) * Kf);
P = P - n * (hump - 0.5) * Km;
N = calculatenormal(P);
}

surface snow(float Kd = 0.8,
                   snow_ht = 0.5)
{
normal  n = normalize(N),
        nf = faceforward(n, I);
float   hump = 0;
color   surfcolor = Cs;

// Querry the displacement shader
if(displacement("hump", hump) == 1) {
    if(hump >= snow_ht)
        surfcolor = 1;
    }
color  diffusecolor = Kd * diffuse(nf);
Oi = Os;
Ci = Oi * surfcolor * diffusecolor;
}
```

The rib file used to render figure 1 is shown in listing 3. It should be noted the surface shader, despite changes to the value of the "`Kf`" parameter of the displacement shader, correcly colorizes (ignore the aliasing) the bumps irrespective of their location on the surface of the sphere. Message passing ensures the coordinated behavior of the shaders.

Listing 3 (snowOnHills.rib)

```
Display "untitled" "it" "rgba"
Format 250 250 1
Projection "perspective" "fov" 40
ShadingRate 1

Translate  0 0 3
Rotate -30 1 0 0
Rotate 0   0 1 0
Scale 1 1 -1
WorldBegin
    LightSource "pointlight" 1 "intensity" 45 "from" [3 3 3]
```

```
    TransformBegin
        Surface "snow" "snow_ht" 0.5
        Displacement "hills" "Kf" 8

        Attribute "bound" "displacement" [0.1]
        Color 0.341 0.266 0.184
        Sphere 1 -1 1 360
    TransformEnd
WorldEnd
```

Basic Code - Class Based Shader

Listing 4 gives the first "cut" of a class-based shader that mimics the behavior of `hills.sl` and `snow.sl`. The first thing to notice is that the use of the reserved word `class` gives no indication of what "kind" of shader is being implemented. In contrast, the source code of a traditional shader immediately "declares" what it is implementing by the use of a reserved word such as `surface`, `displacement`, `light` etc.

Only upon further inspection of `snowOnHills.sl` do we discover that it encapsulates two items of functionality. It can perform displacement shading and surface shading, both of which are implemented by special functions known, in the terminology of Object Oriented Programming (OOP), as **methods**. A class-based shader, such as snowOnHills, is not required to implement both of these methods but in doing so it ensures that data such as `n` and `hump` can be shared its methods.

Listing 4 (snowOnHills.sl)

```
class snowOnHills(float Kd = 1,
                        Km = -0.1,
                        Kf = 8,
                        snow_ht = 0.5)
{
varying float  hump = 0;
varying normal n = 0;

public void displacement(output point P; output normal N) {
    n = normalize(N);
    hump = noise(transform("shader", P) * Kf);
    P = P - n * (hump - 0.5) * Km;
    N = calculatenormal(P);
    }

public void surface(output color Ci, Oi) {
    n = normalize(N);
    normal nf = faceforward(n, I);
    color   surfcolor = Cs;

    if(hump >= snow_ht)
        surfcolor = 1;
    color  diffusecolor = Kd * diffuse(nf);
```

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

Assigning the Shader Object in a Rib File

If the implementation of snowOnHills contained **only** a displacement method it would be obvious that a rib file that referenced the shader (object) should do so as follows,

```
Displacement "snowOnHills" "Km" -0.1 "Kf" 8
```

However, snowOnHills has both displacement **and** surface shading capabilities, so it is less obvious how it should be referenced in a rib file. As shown in figures 2 and 3 using it as a Displacment shader or a Surface shader yields very different results.

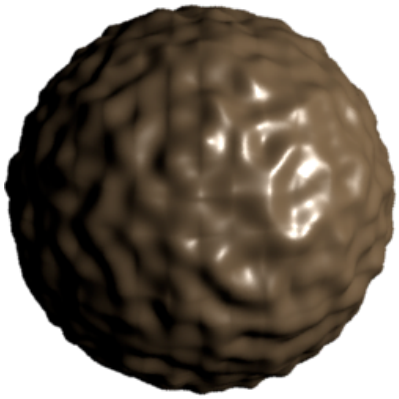


Figure 2
Using the shader object as a Displacement shader ie.

```
Displacement "snowOnHills"
Surface "plastic"
```

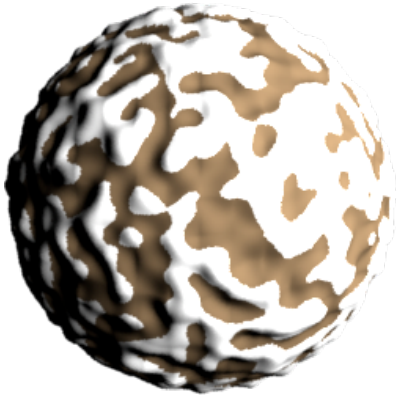


Figure 3
Using the shader object as a Surface shader ie.

```
Surface "snowOnHills"
```

Specularity is absent from figure 3 because the surface method does not perform a specular lighting calculation. The shader can be adapted to take advantage of (re-lighting) rendering efficiencies that might possibly be introduced in future versions of Pixar's software. This is the subject of the next section.

Factored Lighting

Listing 5 replaces the `surface()` method with the following,

```
public void prelighting (output color Ci, Oi)
public void lighting    (output color Ci, Oi)
public void postlighting(output color Ci, Oi)
```

Apart from the declaration of `diffusecolor` and `surfcolor` as member (ie. shared) variables, the functionality of the new shader object is the same as the previous version. Using the new shader object in a rib file is the same as figure 3 ie.

```
TransformBegin
    Surface "snowOnHills"
    Attribute "bound" "displacement" [0.1]
    Color 0.341 0.266 0.184
    Sphere 1 -1 1 360
TransformEnd
```

Listing 5 (factored lighting)

```
class snowOnHills(float Kd = 1,
                        Km = -0.1,
                        Kf = 8,
                        snow_ht = 0.5)
{
varying float  hump = 0;
varying normal n = 0;
varying color  diffusecolor = 0;
varying color  surfcolor = 1;

public void displacement(output point P; output normal N) {
    n = normalize(N);
    hump = noise(transform("shader", P) * Kf);
    P = P - n * (hump - 0.5) * Km;
    N = calculatenormal(P);
    }
public void prelighting(output color Ci, Oi) {
    if(hump >= snow_ht)
        surfcolor = 1;
    }
public void lighting(output color Ci, Oi) {
    diffusecolor = diffuse(n) * Kd;
    }
public void postlighting(output color Ci, Oi) {
    Oi = Os;
    Ci = Oi * Cs * surfcolor * diffusecolor;
    }
}
```

Using Co Shaders

For the purposes of illustrating the use of **co shaders**, the version of snowOnHills in this section does not use factored lighting. Although a "co shader" is part of the shading pipeline it is not a shader as such - at least not in the sense that it can be assigned and used, by itself, to shade an object. Instead, it implements one or more methods that can be called upon to perform calculations on behalf of a class-based shader (ie. shader object). For example, listing 6 provides the code for a co shader that "returns" white only for micro-polygons of a bump that are facing upward.

Listing 6 (Co shader)

```
class hillColor()
{
public void getColor(normal dir;
                     float  KsnowLine, altitude;
               output color c;)
    {
    vector objectDir = transform("world", dir);
    // Facing upward, therefore, show snow!
    if(ycomp(objectDir) >= 0 && altitude >= KsnowLine)
        c = 1;
    else
        c = Cs;
    }
}
```

A co shader friendly version of snowOnHills is shown next.

Listing 7

```
class snowOnHills(float Kd = 1,
                        Km = -0.1,
                        Kf = 8,
                        snow_ht = 0.5;
                  string  co_shader = "")
{
varying float  hump = 0;
varying normal n = 0;

public void displacement(output point P; output normal N) {
    n = normalize(N);
    hump = noise(transform("shader", P) * Kf);
    P = P - n * (hump - 0.5) * Km;
    N = calculatenormal(P);
    }

public void surface(output color Ci, Oi) {
    n = normalize(N);
    normal nf = faceforward(n, I);
    color   surfcolor = Cs;
    if(co_shader != "")
        {
        shader shd = getshader(co_shader);
        shd->getColor(n, snow_ht, hump, surfcolor);
        }
    else
        {
        if(hump >= snow_ht)
        surfcolor = 1;
        }
    color  diffusecolor = Kd * diffuse(nf);
```

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

Using a co-shader in a rib file is relatively straightforward. For example, RmanTools can write the appropriate rib statement - figure 4.

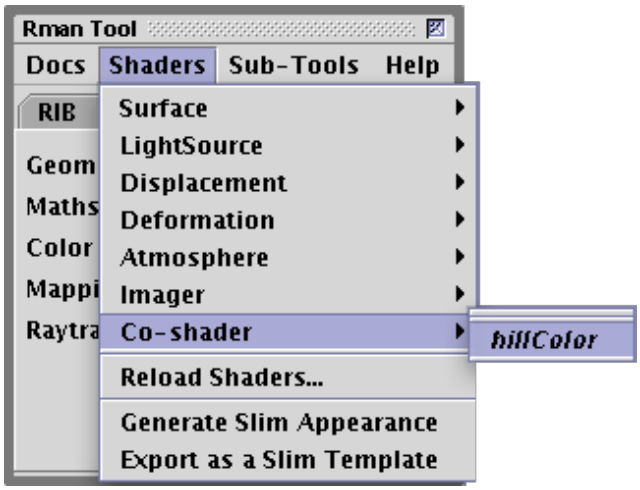


Figure 4

As shown below, Cutter inserts a comment that names of the public method(s) implemented by the co-shader. It also provides a generic name ("`local_name`") by which the co-shader can be referenced by the shader object that will make use of it ie. snowOnHills. It is best to change the generic name to something that is descriptive of the purpose of the co-shader.

```
    TransformBegin
        # Public method: getColor()
        Shader "hillColor" "locale_name"
                "foo" 1.0
        Surface "snowOnHills" "co_shader" ["locale_name"]
        Attribute "bound" "displacement" [0.1]
        Color 0.341 0.266 0.184
        Sphere 1 -1 1 360
    TransformEnd
```

The effect of the co-shader can be seen in figure 5.

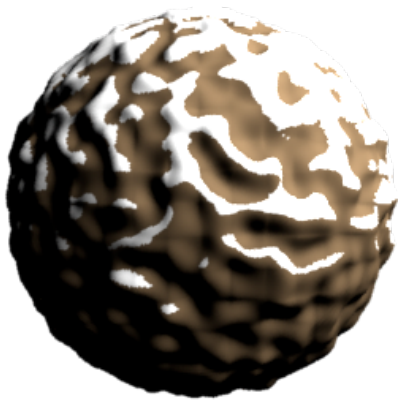


Figure 5

Why Use Co-Shaders?

In the context of the relatively simple code for snowOnHills, it makes very little sense to use a co-shader for the calculation of the surface color. However, that is true only because the code has been deliberately kept simple for the sake of the tutorial. A reason for using a co-shader is its "plug-and-playness". For example, without making any changes to the snowOnHills shader, a different effect can be achieved merely by substituting another co-shader. The main point to note is that snowOnHills expects to call a co-shader that implements a public method with this signature,

```
public void getColor(normal; float, float; output color)
```

Therefore, **any** co-shader that has a public method of the "form" expected by snowOnHills can be deployed. Listing 8 gives the code for a different co-shader that can be used by snowOnHills.

Listing 8 (drift.sl)

```
class drift(vector snowDrift = vector(0,1,0))
{
vector snowDir;

public void construct() {
    snowDir = transform("object", normalize(snowDrift));
    }

public void getColor(normal dir;
                     float  KsnowLine, altitude;
               output color c;)
    {
    vector objectDir = transform("object", normalize(dir));

    // Calculate the dot product to decide if we're
    // facing the direction of the snow.
    if(objectDir.snowDir >= 0)
        c = 1;
    else
        c = Cs;
    }
}
```

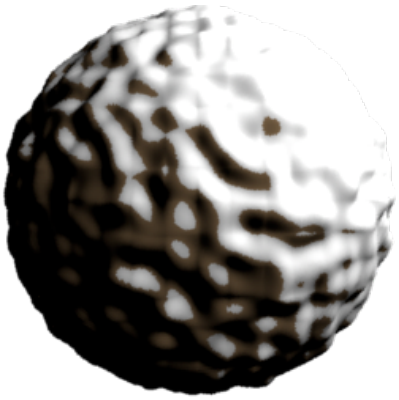


Figure 6
Setting a snow direction.

The rib that produced figure 6 was edited as follows,

```
TransformBegin
    # Public method: getColor()
    Shader "drift" "locale_name" "snowDrift" [1 1 0]
    Surface "snowOnHills" "co_shader" ["locale_name"]
    Attribute "bound" "displacement" [0.1]
    Color 0.341 0.266 0.184
    Sphere 1 -1 1 360
TransformEnd
```

Cutter
Shader Writing

Introduction

Editing rib and rsl files with Cutter offers many advantages compared to using a general purpose text editor. Cutter applies syntax coloration to both types of scripts. Rendering a rib file and compiling a shading language document is conveniently accomplished using the keyboard shortcuts Alt + e, Control + e or Apple + e. If Pixar's documentation is installed on the users computer Alt + double clicking on a keyword in a rib or rsl file will trigger Cutter to display the relevant html document in its internal browser. Being able to quickly refer to Pixar's documentation in an excellent aid to learning about their unique rendering and shading technology. As an added bonus for those who wish to use their custom shaders with either RenderMan Artist Tools (RAT) or RenderMan Studio (RMS), Cutter automatically writes a slim appearance file for each shader it compiles. In addition, for users of RAT or RMS, Cutter can also convert shading language source code into a Pixar Slim template, thus enabling artists to add custom shading nodes to Slim. For detailed information about Cutter and Slim refer to the tutorial "Cutter: Automatic Conversion of Shaders and RSL Functions to Slim Files" For users who wish to use their custom shaders with Houdini, Cutter can automatically invoke "rmands" (a utility application that is part of the Side Effects installation) in order to create and update an artists OTL file.

This tutorial outlines how Cutter should be set up. The tutorial assumes the reader has installed a RenderMan complient renderer.

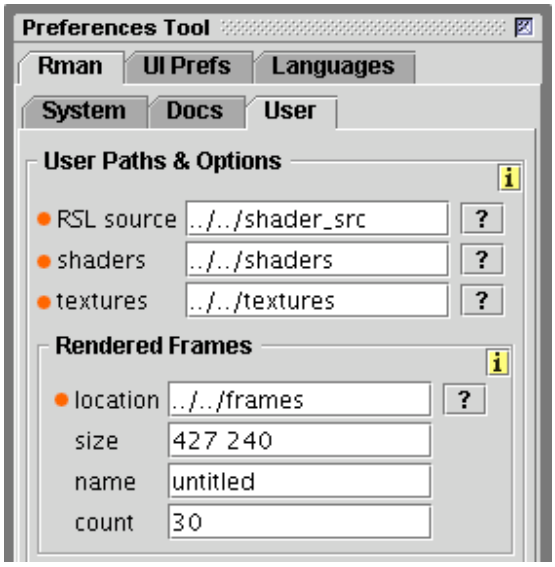
Using Cutter for Shader Writing

First, the reader should check their RenderMan (Rman) preference settings in Cutter. Open the preferences window ie.

```
Edit->Show Preferences->Rman->User
```

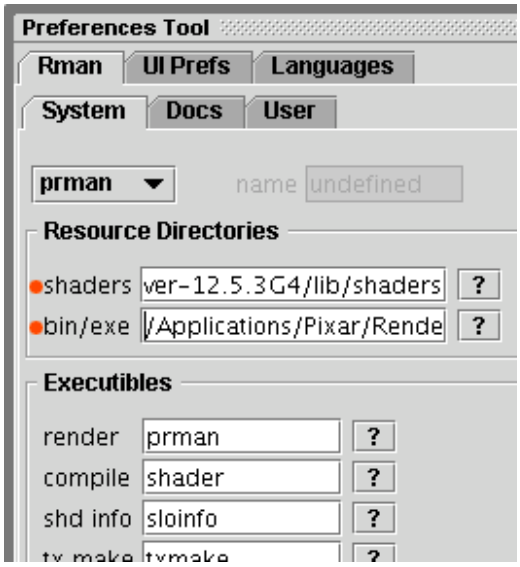
Setting the User Paths

Set these paths to the directories that will store your shader source code, shaders, textures and rendered frames of animation. The paths can be specified as full or relative. Relative paths "begin" at the directory in which the cutter.jar file is located.



Cutter 1

Setting the directories for the shader source files, shaders, textures and rendered frames.



Cutter 2

Setting the location of the shaders and bin directory of the renderer.

Setting up for Houdini

If your shaders will be used with Side Effects Houdini, "Output to Houdini OTL" shou be activated and a path should be set to a shared OTL file.

```
Edit->Show Preferences->Rman->User->Output to Houdini OTL
```

If the path is left empty Cutter will create a OTL file for each shader it compiles. Initia Cutter will indicate the file does not exist ie.

Cutter 3

Activating OTL output.

Ignore the warning. The path will change from red to black once the OTL is created.

Setting the Preferred Renderer and Pixar's Slim Output

By default Cutter expects to compile shaders and render rib files using Pixar's rende environment ie. RenderMan Pro-Server. As shown below if you are using a different system it must be set using the drop-down menu.

Cutters Shader Development Work Flow

The process of developing and testing a shader consists of repeatedly cycling through the following five steps. Once a shader yields visual results that look promising, then and only then, should it be tested in an application such as Maya or Houdini. Confining the developmental shader writing process entirely to Cutter ensures a very fast work flow.

1 Open a copy of a "constant_test" shader ie.

```
Rman Tool->Docs->Shader Docs->Constant
```

2 Save the file as "contant_test.sl" in your "shader_src" directory

3 Compile the shader - keyboard shortcuts Alt+e, Control+e or Apple+e.

Cutter will ensure the compiler will save the shader to the users "shaders" directory.

4 Open a rib file to test the shader ie.

```
Rman Tools->Docs->Single Frame Rib
```

Cutter will generated a rib file that references the compiled shader and lists its default parameter values. It will also add a number of `Option "path"` statements that will ensure your shaders, textures and rib archive directories will be searched by the renderer.

5 Save the rib file and render it - keyboard shortcut Alt+e, Control+e or Apple+e.

Cutters Keyframing Facilities

It is often very useful to animate the parameters of a shader in order to see how surface opacity, color and displacements interact. Importing a shader into Maya or Houdini is a time consuming process although, of course, such applications enable an artist to fully assess a shader. Cutter offers a simple keyframing facility that enables animations to be directly and quickly created. For information about this topic refer to the tutorial "Cutter: KeyFraming".

Cutter
Converting Shaders and RSL Functions to Slim Templates

Introduction

This tutorial provides a detailed description of Cutter's ability to automatically generate Slim™ template and appearance files. Although the tutorial includes a brief description of these files it is assumed the reader has some familiarity with Slim palettes and appearance editors.

To avoid confusion, when this tutorial refers to the Slim application it will capitalize the first letter of the word **S**lim. Slim's text files, on the other hand, will be referred to in all lower case ie. **slim**.

Slim scripts are text files that come in two flavors - appearance slims and template slims. Both are built on top of Tcl and both are identified by their ".slim" file extension. An appearance slim file contains GUI information that defines the interface presented by a Slim editor to an artist for the purpose of adjusting the parameters of a shader. By itself an appearance slim file is useless - it must be accompanied by a pre-compiled shader. A template file, on the other hand, is not associated with a pre-compiled shader so in addition to having GUI information it also contains Tcl code that is used by Slim to write and compile a shader "on-the-fly".

A shader (plus an optional appearance slim file) imported into Slim (RenderMan Artist Tools RAT) or HyperShade (RenderMan Studio RMS) appears as a "non-connectable/static" shading node. A template slim file when read by Slim appears as a "connectable/dynamic" shading node. When using RMS a template cannot be referenced directly by HyperShade. However, once Slim has used the template file to generate and compile a shader it can be **added** to the scene ie. HyperShade loads the newly compiled shader.

A full description of the slim file format can be found at within the Rat documentation at,

```
programmingRAT/customizing_slim/slimfile.html
programmingRAT/customizing_slim/templatesAdvanced.html
```

Refer to the tutorial "Slim Quick Reference" for examples of slim "parameter blocks". Cutter can assist an artist by automatically generating both appearance and template slim files.

Appearance Slim Files

When the renderer in the RmanTools->Options->Environment popup menu (figure 1) is set to Pixar, Cutter automatically generates an appearance slim document for the RSL source code file being compiled.

Figure 1

For example, if the source code for Pixar's classic cloth.sl shader is compiled, Cutter will generate an appearance file called cloth.slim. Cutter saves its appearance slim files in the same directory as the compiled shader file.

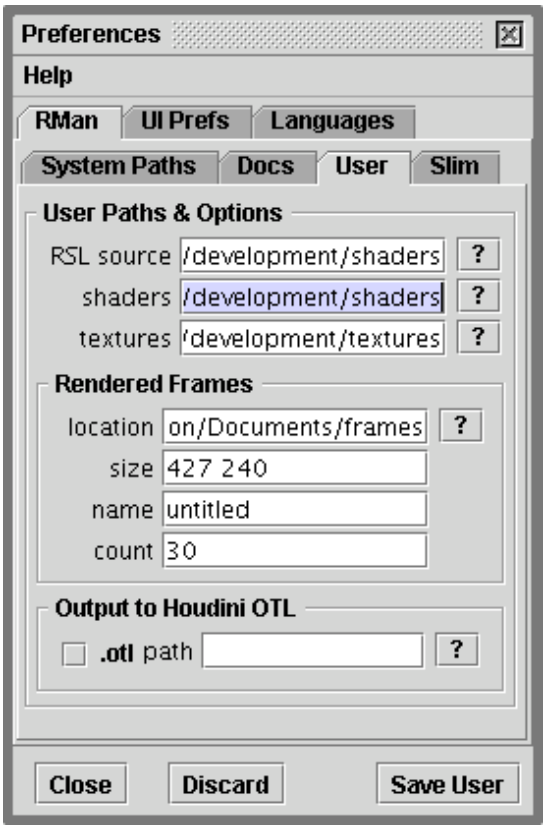


Figure 2 - Setting the shaders directory

A users preferred shader directory, figure 2, can be set using the

```
Edit->Show Preferences->Rman->User->shaders
```

By including user-interface hints within the comments that "accompany" the

declaration of a shaders parameters (instance variables) an artist can easily take advantage of Cutters ability to "tune" the way a shader is presented by HyperShade (RMS) or Slim (RAT).

Template Slim Files

Cutter can generate a template slim document from most types of shaders or RSL functions - but not functions that return void or an array. For example, figure 3 shows the code for an RSL function, taken from the Advanced RenderMan book, being exported as a slim template. After saving the slim document it must be loaded or read by Slim.

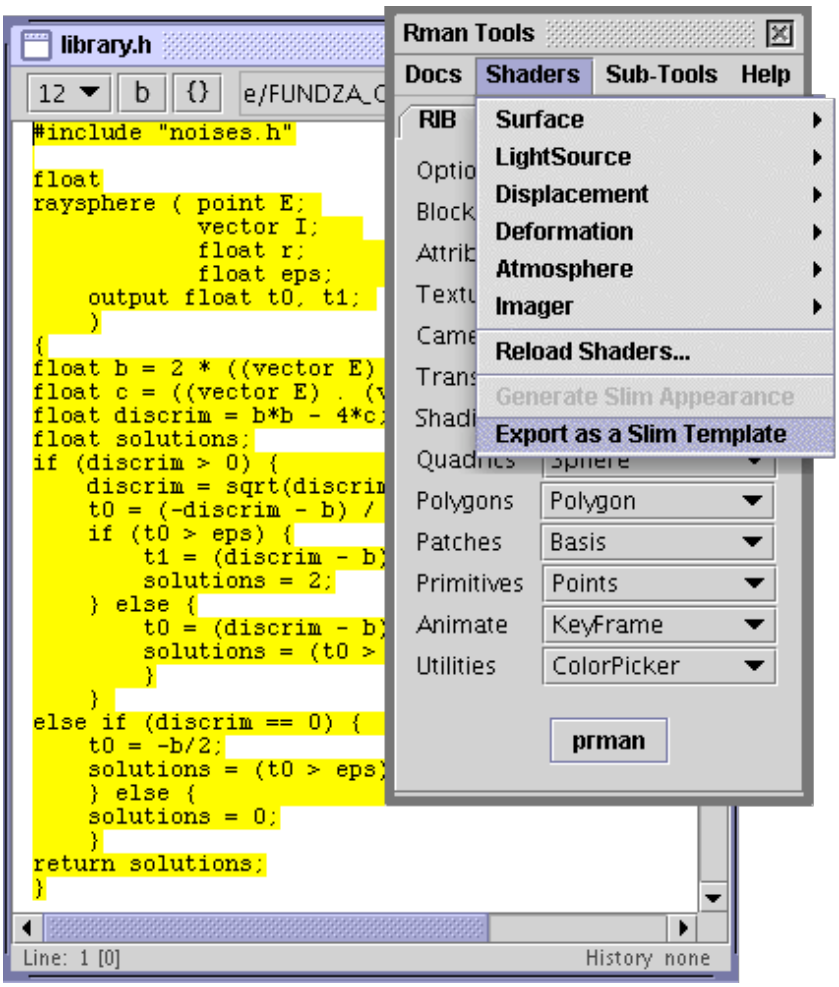


Figure 3

Cutter & Custom Templates

This section focuses on the loading and use of templates produced by Cutter. First, the procedure for RAT will be dealt with followed by RMS.

RenderMan Artist Tools

Make sure you have the "expert menu's" option activated in
`Slim->Preferences` tab.

A custom slim file can be read by Slim via its console - figure 4.

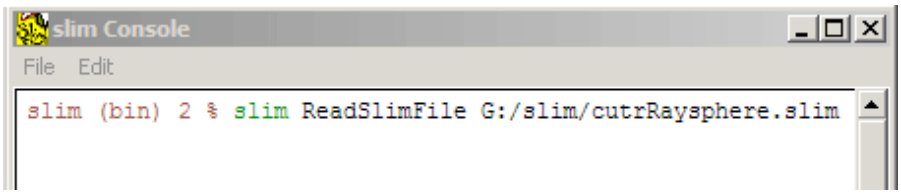


Figure 4

To create a node from the custom template use the "Preloaded" menu item (figure 5).

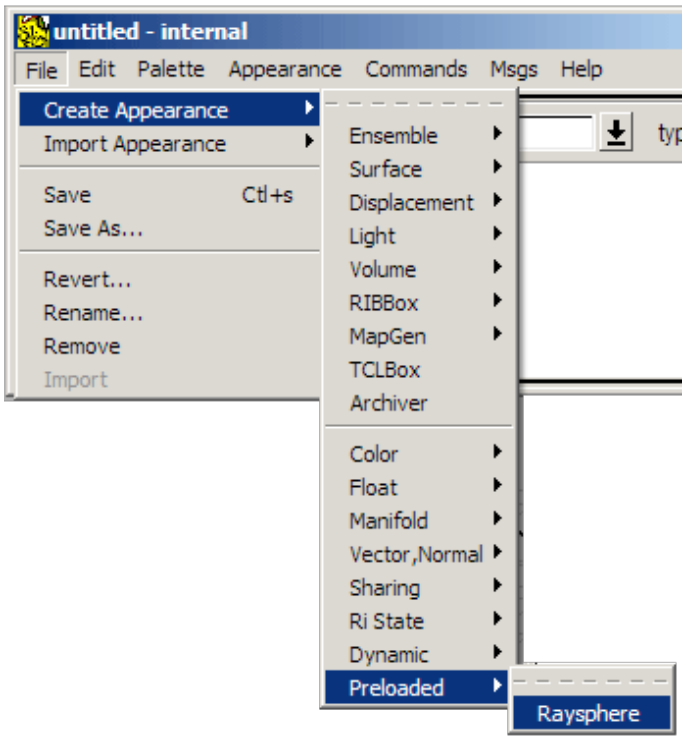


Figure 5

RenderMan Studio

Make sure the "expert menu's" option is activated in `Slim->Preferences->Interface` tab.

A custom slim file can be read by Slim via its console - figure 4. To create a node from the custom template you will need to dig around the "floats" sub-menus (figure 6).

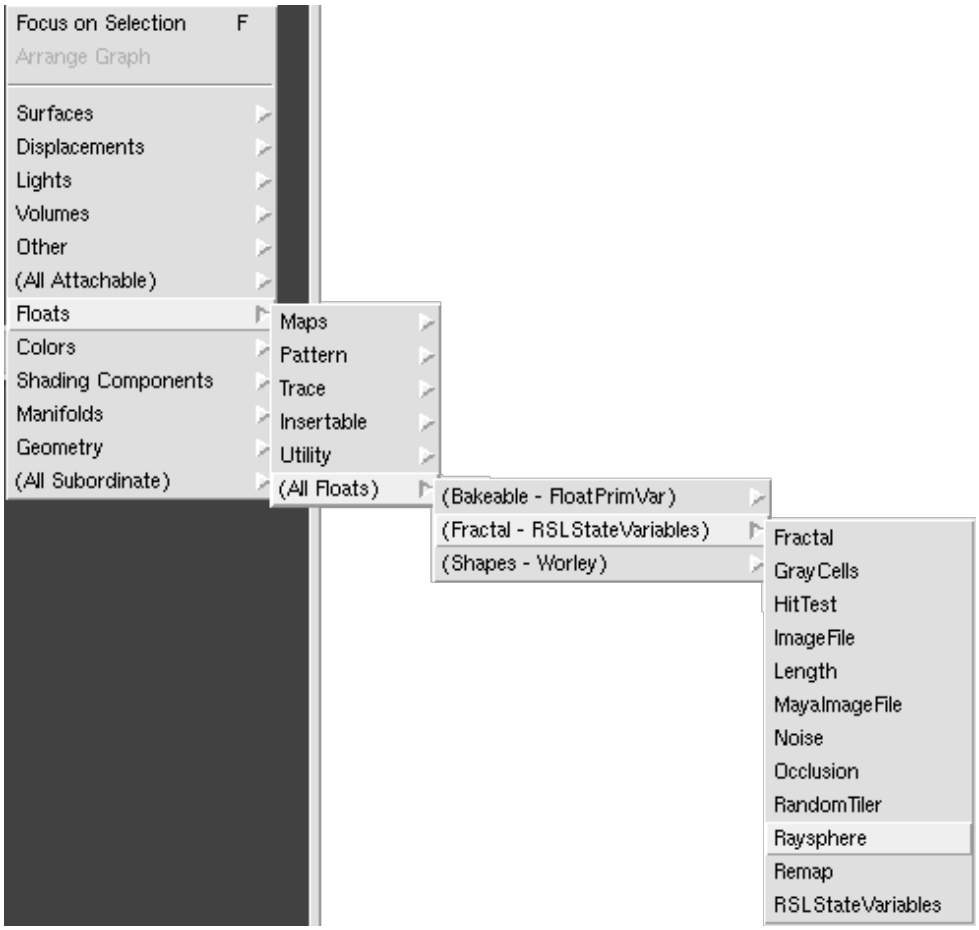


Figure 6

Cutter and Maya's Command Port

Cutter is able to communicate with Maya via a port. In Maya enter the following command,

```
commandPort -n ":2222"
```

There is no particular significance to the port number "2222" but its the one that Cutter uses by default. It can be changed in Preferences - figure 7.

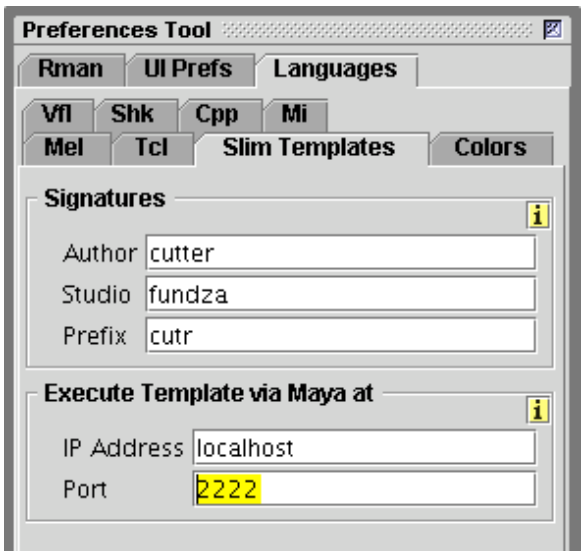


Figure 7

Assuming you have a template slim file open on Cutter's desktop execute the file using the keyboard shortcut, alt+e, control+e or apple+e. Cutter will write a temporary mel script into its own directory and then send a mel `source` command to Maya on the users chosen port. Depending on whether you are using RAT or RMS, Cutter will write the appropriate mel script that will cause Slim to,

- open a palette window
- add a node created from your template slim file, and
- open an editor for the node.

Cutter & UI Hints

When Cutter writes an appearance or template slim file it parses the data for each shader/function parameter. Cutter reads the datatype, name, default value and any **commented text** that accompanies a parameter. Converting a parameter datatype, name and value to the appropriate Slim file format is straight forward. However, Cutter uses certain items of information "embedded" within the comments as a guide to how it should define the GUI that Slim will present to the artist. Cutter determines what information to put into a slim file in a two step process.

Step 1

First, it guesses what each parameter should look like when they are displayed by the Slim editor. The guesses are based on the data type and name of each shader parameter or function argument.

Step 2

Next, Cutter looks for a user-interface hint (ui-hint) within the comments associated with a parameter. Cutter considers any text it finds within "[" and "]" brackets as possible sources of information that it can use to refine the "look" of the GUI.

Table 1 gives a full listing of the rules that it applies when making a guess about the "look" of a GUI. Table 2 lists what Cutter considers to be valid ui-hints.

Table 1 Guessing

float param If param begins with "**K**", for example, float **K**z = 0; or the param name is either, *roughness, blend, mix, smooth or step* The slider will have the range 0 to 1.	If the name of a parameter matches the names shown on the left, or if its initial character is "K", Cutter will write a slim document that will assign a slider. This occurs only if the parameter is of type **float**
string param A guess is made about the type of popup menu that should appear next o the text field based on a partial match of param with the following,	The list of partial names on the left are associated with the following Slim subtype's. Subtypes configure the popup menu that appears next to a text field. Sub-types are,

tex, tra, env, filter, shad, refl, world, shader, camera, current or coord	"texture" "environment" "filter" "shadow" "reflection" "spacename" "spacename"

Table 2 UI-hints

float param /* [range hint] */ **Examples** /* [-2 5] */ /* [-2 5 1] */ /* [0 or 1] */ /* [low med high 1 2 3] */	 "slider" range from -2 to 5 "slider" 1 unit increments "switch" on or off "selector" popup names/values
string param /* [string hint] */ **Examples** /* [texture] */ /* [environment] */ /* [filter] */ /* [shadow] */ /* [reflection] */ /* [spacename] */	A Slim popup will match th ui-hint. In the case of filter a "**selector**" will be assigned with following values, box gaussian disk radial-bspline
ANY param /* [collection hint] */ **Examples** /*[collection foo] */ /*[collection foo 0 or 1] */ /*[collection foo pixar,FNoise]*/	If a hint begins with "**collection**" the parameter is bundled into a Slim collection . The collection is displayed in a "closed" state.
ANY param /* [inline hint] */ **Examples** /* [pixar,FNoise] */	Hints that include a comma will "hard-wire" a connection node to the parameter. The connected node will appear (inline) within a collection.
ANY param /* [expression hint] */ **Examples** /* [exp {lerp(0.0,1,$pct}] */ /* [expr {lerp(0.0,1,$pct}] */ /* [expression {lerp(0.0,1,$pct}] */	A hint beginning with "exp", "expr" or "expression" followed by text bounded by "{" and "}" defines a Tcl expression.

Hybrid

Defining shading nodes with template slim files provides an artist with the flexibility to connect its inputs to a shading network. On the other hand, limiting an artist to using pre-compiled shaders prevents them from using the node as part of a shading network. Between these two "extremes" there is a hybrid approach in which the artist is given a custom node, defined by a slim template, but some of its parameters are hard-wired ie. pre-connected, to specific nodes. Using the "inline hint" shown in table 2 ensures a parameter is pre-connected and that the connection cannot be broken.

Copies of Shaders & their Appearance Slim Files

A slim appearance file generated by Cutter specifies the full path to the shader that "accompanies" the slim file, for example,

```
slim 1 appearance slim {
    instance surface "foo" "//C/shaders/foo" {
```

This is fine when the shader (plus slim appearance file) is imported into Slim and the Maya scene is rendered locally but an absolute path causes errors when a Maya project directory is moved to a render farm for remote rendering.

The path to the shader can be relative if the shader (and its slim document) are stored in Maya's project "rmanshader" or "rib" directory. In addition to specifying a fixed "shaders" directory, users can nominate an additional (temporary) directory in which copies of their ".slo" and ".slim" files can be saved by Cutter - figure 6.

Figure 8

For example, if the user selects the "rmanshader" directory the path to the shader in a slim appearance document would be of the form,

```
slim 1 appearance slim {
    instance surface "foo" "rmanshader/foo" {
```

Limitations

There's probably lots of them, but here are the known ones!

#include's

If the include statement does not specify the full path to a header, Cutter applies a number of rules when determining what path to provide in the output template document

System headers ie. **#include <foo.h>**, appear in the template file prefixed with the full path to the **lib/shaders** directory of the Pixar installation. Cutter does not check

the existence of the file.

Include statements such as **#include "foo.h"** cause Cutter to search for the header file in the following locations.

1. the same directory as the source document
2. the RSL source directory specified in the Preferences (figure 2)
3. the shaders directory specified in the Preferences (figure 2)
4. the system headers directory

The first location in which the file is found will be used as the specification of the full path given to the slim output document. If the include file cannot be found it is specified "as is" in the output slim document - this will almost certainly cause an error later if the template is loaded into slim.

While it is acceptable to use header files that reference other headers when compiling a shader, it appears that slim template documents cannot do the same. Therefore, you will have to edit your include files so that they do not reference "secondary" headers. There may be a work-around but I do not know what it is!

Shader Instance Variables and their Default Values

Cutter can "read" these assignments,

```
float foo = 5,
      koo = radians(5);
```

but not these assignments,

```
float foo = 1/(2 * PI);
float pp  = PI;
```

Cutter can handle fixed length arrays but not the newer kind of variable length arrays. For example, this is acceptable,

```
color nnn[3] = {1, (1,2,3), color(1)};
```

#define's

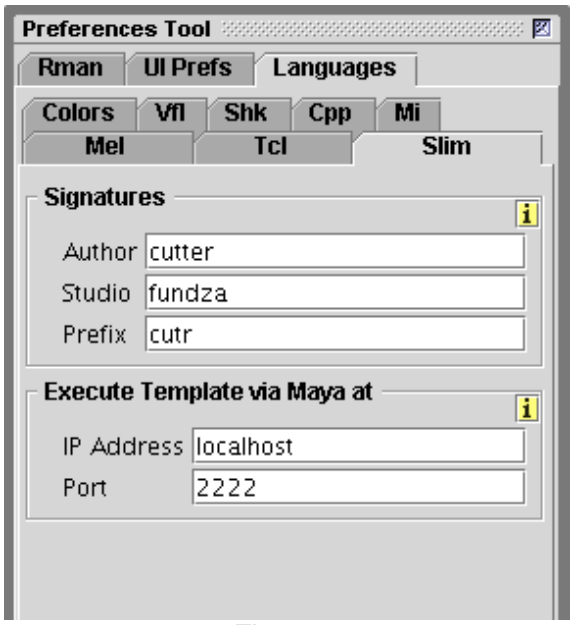
These cannot appear in the document from which a slim template will be generated. They must be moved into an include file. This may change in later versions of Cutter.

custom functions implemented in an SL document

Custom functions must be moved to an include file.

Signatures

The Languages->Slim preferences panel (figure 9) enables a user to set their "studio", "author" and "prefix" signatures. These items are used to identify the slim templates generated by Cutter.


Preferences Tool
Rman
UI Prefs
Languages
Colors
Vfl
Shk
Cpp
Mi
Mel
Tcl
Slim
Signatures
Author
cutter
Studio
fundza
Prefix
cutr
Execute Template via Maya at
IP Address
localhost
Port
2222
Figure 9