

Shader Writing

Jan Walter

February 9, 2004

Contents

1	Which Renderers Support Custom Shaders?	4
2	How Many Shader Types Are There?	5
2.1	RenderMan	6
2.2	Mental Ray	6
2.3	Mantra	8
2.4	Exercise	9
3	Quick Start	9
3.1	A Constant Surface Shader	10
3.1.1	RenderMan	11
3.1.2	Mental Ray	12
3.1.3	Mantra	15
3.2	How Do Several Shaders Works Together?	16
3.2.1	RenderMan	16
3.2.2	Mental Ray	21
3.2.3	A Simple Scene With Five Different Shaders	28
4	Get Your Hands Dirty	40
4.1	Default Surface Shader	40
4.2	Stairs	42
4.3	Simple Room	46
4.4	Caustics	54
4.5	Baking	61
5	Light Source Shaders	67
5.1	Ambient Light Sources	69
5.2	Distant Light Sources	70
5.3	Point Light Sources	70
5.4	Spot Light Sources	71
5.5	Area Light Sources	71
5.6	Light Sources And Shadows	72
6	Surface Shaders	73
7	Displacement Shaders	73
8	Imager Shaders	73
9	Volume Shaders	73

10 Tables

List of Figures

1	Vshade	12
2	Using the constant shader	13
3	The VEX Builder	15
4	The hair primitive	28
5	Simple test scene	29
6	RenderMan's torus primitive	30
7	A canvas imager shader for BMRT	32
8	Test scene with imager shader and surface shader	35
9	Test scene with imager, surface, and light shaders	38
10	Test scene with bump mapping	39
11	Test scene with true displacement	40
12	Rendering the stairs with real geometry	43
13	Faking the stairs within the shader	43
14	Real geometry from different perspectives	44
15	Faked stairs from different perspectives	44
16	A simple scene in a room with a window	46
17	Reflections in a glass sphere (Radiance)	48
18	Reflections in a glass sphere (mental ray)	48
19	Caustics rendered with mental ray	54
20	Chrysler Building rendered with Lucille	63
21	Chrysler Building rendered with 3Delight	64
22	Baked normals and positions of a head	65
23	Use a procedural shader to grow hair	66

List of Tables

1	A few predefined surface shader variables	11
2	Color spaces	68
3	Predefined surface shader variables	73
4	Predefined imager shader variables	74
5	Predefined light source shader variables	74

1 Which Renderers Support Custom Shaders?

For film production and special effects for TV and movies there are basically two widespread renderers but I will add another renderer to the list because this renderer was originally designed to be very close to **PRMan**¹ and I expect it to be more and more important to the industry in the future:

- **RenderMan** compatible renderers. RenderMan® is a registered trademark of *Pixar*. The RenderMan Interface is a standard interface between modelling programs and rendering programs capable of producing photo-realistic quality images. It's defined in a specification published by Pixar under the title *The RenderMan Interface*. There are a lot of RenderMan compatible renderers but I will focus only on a few of them:
 - **PRMan**² This is Pixar's implementation of the RenderMan Interface. In the past it was a pure scanline renderer based on the **REYES**³ architecture. Today it's a hybrid approach which includes ray tracing and global illumination features. Pixar published the basics about shader writing and RenderMan in the book [1] which was for a very long time the only book available on this subject.
 - **AIR** is another commercial implementation of the RenderMan Interface but because of its low price it's an alternative to PRMan and it has a few very interesting features⁴.
 - **Pixie** is a free implementation and one of the RenderMan compatible renderers which come with the full source code. I think this is interesting enough to mention here for the few people who would like to know more about: How to implement a RenderMan compatible renderer?
 - **BMRT**⁵ was for a long time the only freely available RenderMan compatible renderer. Albeit it did not come with the source code. The reason why I mention it here is that you still can find versions of BMRT on the Internet and that the book [2] is about PRMan and BMRT. There are many examples in that book which will work for both renderers without changing the shaders.

¹See below for reference

²PhotoRealistic RenderMan

³Renders Everything You Ever Saw

⁴E.g. its variant called **BakeAIR** which is used for baking all kind of information available during rendering time.

⁵Blue Moon Rendering Tools

- **Entropy** was a commercial implementation of the RenderMan Interface by a company called *Exluna* but it disappeared from the market after SIGGRAPH 2002 together with the official web page for BMRT. Both renderers were implemented by Larry Gritz the coauthor of the book [2] and you will still find a lot of useful comments by him on the news group about RenderMan⁶.
- **Angel** was written by the author of the third book about RenderMan. See [3] for reference. The renderer is freely available for Linux, free-BSD, SGI, and Windows and there was also a port to the Playstation.
- **Mental Ray** is a commercial general-purpose renderer from a company called *mental images*. See [4] and [5] for references. You can't buy the software directly from mental images but it's integrated in a lot of commercial modelling and animation systems and you can get the software from dealers for this packages. With the latest edition of the second book about mental ray you will get a demo version of the renderer which should enable you to use all shader writing techniques described in both books without the need to buy a full license of the renderer.
- **Mantra** is a renderer which comes with **Houdini** from *Side Effects*. Houdini's VEX language is loosely based on the C language but takes pieces from C++ as well as the RenderManTM shading language. The reason why I include this renderer is that Side Effects offers a non-commercial version called **Houdini Apprentice** and you can render with Mantra for free. The commercial version allows you to render with both PRMan⁷ and mental ray. Houdini's **VEX Builder** can be used to write shaders for Mantra by simply connecting operators. This is a concept which is found in other packages as well and in theory you can use it to write even shaders for RenderMan and mental ray. Pixar offers a program called **SLIM** with their **RenderMan Artist Tools** (RAT) and AIR comes with a similar program called **Vshade**⁸.

2 How Many Shader Types Are There?

Dependent on the renderer you are going to use there are several shader types supported.

⁶comp.graphics.rendering.renderman

⁷or any other RenderMan compatible renderer

⁸available only for Windows

2.1 RenderMan

For most RenderMan compatible renderers you will find following shader types:

- **Surface Shaders:** Surface shaders are attached to all geometric primitives and are used to model the optical properties of materials from which the primitive was constructed. A surface shader computes the light reflected in a particular direction by summing over the incoming light and considering the properties of the surface.
- **Light Source Shaders:** A light source shader calculates the color of the light emitted from a point on the light source towards a point on the surface being illuminated. A light will typically have a color or spectrum, an intensity, a directional dependency and a fall-off with distance.
- **Volume Shaders:** Volume shaders modulate the color of a light ray as it travels through a volume. Volumes are defined as the insides of solid objects. The atmosphere is the initial volume defined before any objects are created.
- **Displacement Shaders:** Displacement shaders change the normals and/or position of points on the surface and can be used to place bumps on surfaces.
- **Imager Shaders:** An imager shader manipulates a final pixel color after all of the geometric and shading processing has concluded.

A good source for RenderMan shaders is the following web page:

<http://www.renderman.org>

2.2 Mental Ray

- **Material Shaders** are maybe the most important shader type in *mental ray*. They are called whenever a visible ray (eye ray, reflected ray, refracted ray, or transparency ray) hits an object.
- **Volume Shaders** may be attached to the camera or to a material. They accept an input color that they are expected to modify to account for the distance the ray travelled through a volume.
- **Light Shaders** are called from other shaders by sampling a light or directly if a ray hits a source.

- **Shadow Shaders** are called instead of material shaders when a shadow ray intersects with an object. Shadow rays are cast by light sources to determine visibility of an illuminated object.
- **Environment Shaders** provide a color for rays that leave the scene entirely, and for rays that would exceed the trace depth limit.
- **Photon Shaders** are used in the photon tracing phase to compute the photon maps that are used to simulate caustics and global illumination.
- **Photon Volume Shaders** are similar to photon shaders in the same way that volume shaders are similar to material shaders: they compute indirect light interactions in volumes, such as volume scattering.
- **Photon Emitter Shaders** are used in the photon tracing phase to control the emission of photons from the light sources.
- **Texture Shaders** are called exclusively by other shaders to relieve other shaders, such as material or environment shaders, from performing color and other computations.
- **Displacement Shaders** are called during tessellation of polygonal or free-form surface geometry. Whenever the tessellator introduces or copies a vertex, the displacement shader is called and expected to return a scalar value that tells the tessellator to move the vertex by this distance along its normal vector.
- **Geometry Shaders** are functions that procedurally create geometric objects. It allows mental ray to store small and simple placeholders in the scene, and generate the objects only when they are needed, and remove them from the cache when they are no longer needed.
- **Contour Shaders** compute contour colors and widths. Any information about the geometry, illumination, and materials of the scene can be used.
- **Lens Shaders** are called when a primary ray is cast by the camera. They may modify the eye ray's origin and direction to implement cameras other than the standard pinhole camera, and may modify the result of the primary ray to implement effects such as lens flares.
- **Output Shaders** are called when the entire scene has been completely rendered. They modify the resulting image or images to implement special filtering or compositing operations.
- **Lightmap Shaders** sample the object that the material is attached to, and compute a map that contains information about the object. This can be used to bake illumination solutions or store vertex colors for a hardware game engine.

The source code for a lot of mental ray shaders can be downloaded from mental images' FTP site:

```
ftp://ftp.mentalimages.com/pub/shaders/
```

2.3 Mantra

Don't worry about the details given here. Important is at the moment only that there are several shader types. See Houdini's documentation:

```
$HH/vex/html/shading.html
```

- **Displacement Shaders:** Displacement shading can be used to move the position of the surface before the surface gets rendered. It is intended as a mechanism to add fine detail to a surface, not as a modelling technique.
- **Fog Shaders:** A fog shader is responsible for modifying the `Cf`, `Of` or `Af` variables after the surface shader has completed its shading. It is possible to use `illuminate()` statements inside of fog shaders.
- **Image3D Shaders:** Create 3D textures.
- **Light Shaders** will get called from surface or fog shaders to compute the illumination from a given light source. The light shader can be invoked using the `illuminate()` loop or using the standard `diffuse()`, `specular()`, etc. functions.
- **Photon Shaders:** When Mantra is generating photon maps, photon shaders are used instead of surface shaders. Thus, the photon context is very similar to the surface context. However, because of the behavioural differences between the contexts, most of the surface context specific functions are not valid in the photon context.
- **Shadow Shaders:** Shadow shaders will get called from surface or fog shaders to occlude the illumination from a given light source. The light will already have been computed by calling the light shader. The function of a shadow shader is to modify the `Cl` variable. Typically, the light will be occluded, causing the `Cl` variable to decrease in intensity. However, it is possible to create "negative" shadows, and increase the illumination due to occlusion.

- **Surface Shaders:** The surface shading context's purpose is to set the final color, opacity and alpha of the surface being rendered. If the `Of` and `Af` variables are not set, they will default to 1. If the `Af` variable is not set, it will resolve to `avg (Of)`. It is possible to set the `Af` variable to any arbitrary value, making it possible to build matte/cutout shaders.

For help about Houdini or Mantra please visit the following web site where you can find tutorials and discussion forums:

<http://www.sidefx.com/community>

2.4 Exercise

Try to find the relationship between shaders for different renderers.

3 Quick Start

As you can see there are so many options and it's hard to get started. I would consider starting with a **RenderMan** compatible renderer first to be easier but once you got the basics there is a lot more to do and the concepts behind **mental ray** are quite powerful.

The most frustrating experience when you start to work with renderers and try to write custom shaders is to end up with an empty (black) picture. I would encourage you to try to understand at least the human readable versions of scene descriptions for several renderers. You might never have to write them yourself in a text editor but it certainly helps if you know about **RIB**⁹ or **MI**¹⁰ files. A lot of production pipelines still export scene description files in ASCII format once and then modify them with little shell (Perl, Python, etc.) scripts to render for example in several passes (beauty, shadow, light, ...). So it's good to know where the shaders are specified (or called) in the resulting scene description files.

Most of the RenderMan compatible renderers allow you to render without any lights defined in the scene. The default behaviour is "faking" a light source by rendering the scene as there would be a light source defined. To be more specific: If you do not specify any surface shader then a RenderMan compatible renderer uses a surface shader called `defaultsurface`. This

⁹RenderMan Interface Bytestream Protocol

¹⁰This is mental ray's scene description file format.

surface shader is able to render the scene without any lights defined because it does take only the normal and the incident vector into account without asking the lights for their contribution. Mental ray will render without any lights defined but the result will be a “black” image. First thing you should learn is to use the alpha channel for the resulting images. You could at least check if something is in “front” of your camera by looking at the alpha channel of the resulting image. In a RIB file you will find a similar line to the one below:

```
Display "imageFilename.tiff" "file" "rgba"
```

You can check the syntax and possible parameters in Pixar’s document *The RenderMan Interface*. Here just a quick reminder what this line specifies: The **name** `imageFilename.tiff` could be the name of a frame-buffer or (in this case) the filename of the resulting image. The **type** `file` has to be supported by all RenderMan compatible renderers and stands for the default file format. The **mode** `rgba` specifies **red**, **green**, **blue**, and **alpha** channels in the resulting output picture. For mental ray you should look for a line like this in the MI file:

```
output ["type"] "format" [opt] "filename"
```

See chapter 11 in the book [4] for an explanation. If you choose for example `rgb` for the **format** and do not specify the optional parameters **type** and **opt** the resulting image with the name `filename` will have an alpha channel as well because the not specified **type** will default to `rgba`.

3.1 A Constant Surface Shader

We will start with a very simple surface shader which shows you how the results of your shader calculations are passed back to the renderer. We don’t worry about input parameters yet but you will learn that RenderMan uses a graphics state which includes shading attributes like the current color and current opacity. These attributes are known inside the shader as if they would have been defined globally (outside the shader). Mental ray uses another mechanism. It passes the state explicitly to the shaders and they grab the information needed by calling functions to extract the needed bits.

3.1.1 RenderMan

Before we will look into light shaders we will start with a simple surface shader. One of the standard shaders shipping with all RenderMan compatible renderers is the `constant` surface shader:

```
surface constant()
{
    Oi = Os;
    Ci = Os * Cs;
}
```

The source code specifies the shader type (`surface`), the shader name (`constant`), and uses only predefined surface shader variables¹¹. See table 1 for an explanation of the predefined surface shader variables we used so far and table 3 on page 73 for a complete table.

Name	Type	Description
Cs	color	Surface color
Os	color	Surface opacity
Ci	color	Incident ray color
Oi	color	Incident ray opacity

Table 1: A few predefined surface shader variables

In a RenderMan surface shader you will set the variables `Oi` and `Ci` to new values. In this case you simply copy the value for the incoming opacity and you multiply the already existing color by the opacity value. This allows colors from behind the surface to show through without overflowing the bounds of a color.

With AIR there comes a program called `Vshade` for the Windows platform. You can use that program to write the shader visually, compile it with an user interface, and render directly to see the effect. The `constant` shader would look like in figure 1.

But where do the values for `Os` and `Cs` come from? They might be the default values (if not specified) or you will find a line starting with `Opacity` or `Color` in the RIB file (just before the surface shader is called and attached to the following geometry with `Surface "constant"`).

¹¹See Pixar's document *The RenderMan Interface* for a complete list of predefined surface shader variables.

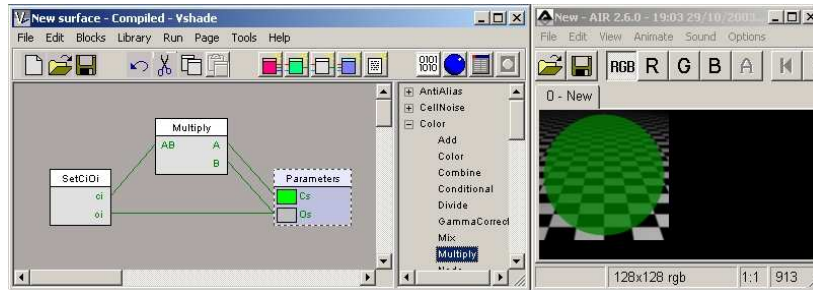


Figure 1: Vshade

You can download the RIB file¹² which was used to render the picture in figure 2 from the same web page where you got this document.

Basically there is a plane in front of a more complicated mesh which looks like a monkey. The plane has the constant surface shader attached and defines a green color and a half transparent material:

```
...
# Plane
Color [ 0 1 0 ]
Opacity [ 0.5 0.5 0.5 ]
Surface "constant"
PointsPolygons ...
...
```

3.1.2 Mental Ray

The shader which can be used to create a similar effect in mental ray is called `mib_opacity` and seems to be significantly more complicated. You can download the MI file¹³, render it with the standard shaders shipping with mental ray, and have a look into the MI file to see how the shader is attached to the geometry:

```
...
camera "Camera"
...
    output          "rgb" "constant.rgb"
...
end camera
```

¹²It's called **constant.rib**.

¹³It's called **constant.mi**.



Figure 2: Using the constant shader

```

...
material "mtl2"
  opaque
  "mib_opacity" (
    "input" 0.0 1.0 0.0,
    "opacity" 0.5 0.5 0.5
  )
end material
...
object "Plane"
  visible trace shadow
  tag 1
  group
    1.0 1.0 0.0
    1.0 -1.0 0.0
    -1.0 -1.0 0.0
    -1.0 1.0 0.0
    v 0
    v 1
    v 2
    v 3
    p "mtl2" 0 3 2 1
  end group
end object
...

```

I will not explain how a mental ray file is structured but here is in short how the shader is attached to the geometry: The shader is defined in the `material` block and the two necessary parameters `input` and `opacity` are set to the same values as in the RenderMan example. The shader is attached to the geometry in the `object` definition by mentioning the name of the material.

The shader `mib_opacity` is not really meant to be used as a standalone material shader but you can use it as one. Normally you would use it as an building block of a so-called PhenomenonTM. But that's something I will explain later.

Have a look at the source code for `mib_opacity` below. It should either come with your mental ray distribution or can be downloaded from mental images' FTP site¹⁴.

```
...
DLLEXPORT miBoolean mib_opacity(
    miColor      *result,
    miState      *state,
    struct mo     *paras)
{
    register miColor *opacity = mi_eval_color(&paras->opacity);
    miColor    inp;

    if (opacity->r == 1.0 && opacity->g == 1.0 &&
        opacity->b == 1.0 && opacity->a == 1.0)
        *result = *mi_eval_color(&paras->input);
    else {
        mi_trace_transparent(result, state);

        if (opacity->r != 0.0 || opacity->g != 0.0 ||
            opacity->b != 0.0 || opacity->a != 0.0) {
            inp = *mi_eval_color(&paras->input);
            result->r = result->r * (1.0 - opacity->r) +
                inp.r * opacity->r;
            result->g = result->g * (1.0 - opacity->g) +
                inp.g * opacity->g;
            result->b = result->b * (1.0 - opacity->b) +
                inp.b * opacity->b;
            result->a = result->a * (1.0 - opacity->a) +
                inp.a * opacity->a;
        }
    }
    return(miTRUE);
}
```

The result of the `mib_opacity` function (and most mental ray shaders) is stored in the variable `result`. In contrast to the shader language of RenderMan you are not forced to use that name but most of the example shaders will. The three calls of the function `mi_eval_color` are used to retrieve the shader parameters from the MI file¹⁵.

If the opacity is “opaque” just the input color is returned. If the material is at least a bit transparent you have to elongate the incoming ray by calling

¹⁴See section 2.2.

¹⁵They are stored in the struct **paras** but that's the function you should use to extract the parameters of type **miColor**. The reason for this is that shaders can operate in the contexts of shader assignments and phenomena without needing knowledge of the context, which is automatically handled.

`mi_trace_transparent`¹⁶. The call might modify the result already before it get mixed with the current color based on the `opacity` value.

3.1.3 Mantra

In Houdini you can use the so-called VEX Builder to create VEX code. Houdini's VEX language is loosely based on the C language but takes pieces from C++ as well as the RenderMan™ shading language. Inside Houdini's VEX Builder you can write VEX code with visual programming. You simply create nodes and link them together. The VEX code is automatically created, compiled, and executed.

In figure 3 you see the constant shader in Houdini. On the right side there is a Viewer Pane which shows you the result of the surface shader applied to a sphere. You can select several shapes to test your shader or create a whole scene with other shapes. In the latter case you have to attach the shader explicitly to the geometry. You can select several backgrounds¹⁷ which is quite useful in this case to see the transparency. You have to check the button to the right of the Background button to turn on the alpha display.

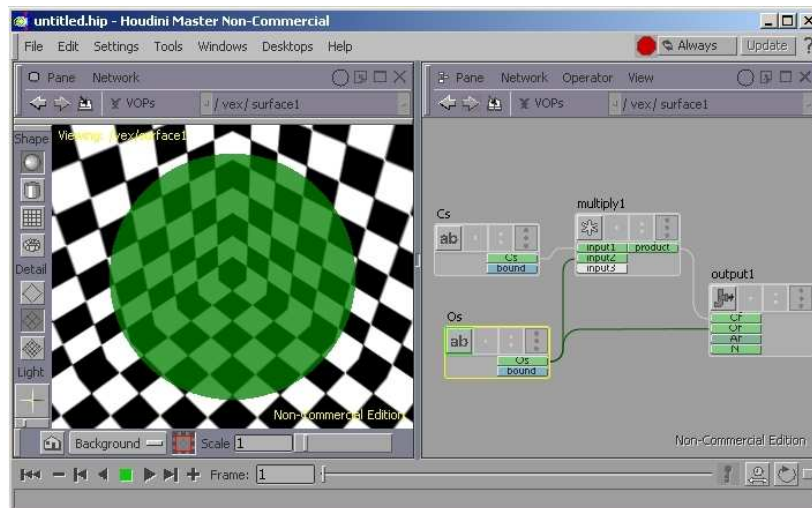


Figure 3: The VEX Builder

¹⁶This does not change the direction of the ray and is therefore more efficient than other calls.

¹⁷The button with the label *Background* at the bottom of the *Viewer Pane* gives you several options.

3.2 How Do Several Shaders Works Together?

3.2.1 RenderMan

Let's start with the interaction between light and surface shaders. I explain the details while we are doing a few tests. Please read section 5 for more details about light shaders.

We create a very simple RIB file:

```
# test.rib
Exposure 1.0 2.2
Display "test.tiff" "file" "rgba"
Format 508 380 1.0
Projection "perspective" "fov" [ 90 ]
WorldBegin
    LightSource "pointlight" 0
    AttributeBegin
        Surface "plastic"
        Translate 0 0 1
        Patch "bilinear"
        "P" [ -1.0 -1.0 0.0 1.0 -1.0 0.0 -1.0 1.0 0.0 1.0 1.0 0.0 ]
    AttributeEnd
WorldEnd
```

You could render this RIB file e.g. with `render test.rib` and check that there is something in front of your camera before you start doing your own tests. It's a simple bilinear patch with 4 control points in front of the camera with a field of view of 90 degrees and translated along the positive z-axis¹⁸.

Now we start to replace the surface shader and the light shader with our own shaders. But first change the resolution of the resulting image. Change the line `Format 508 380 1.0` to `Format 5 4 1.0`. This will reduce the resolution to only a few pixels in each direction. You will soon see why. The next thing which might help compiling your shaders every time you change a line is to create a `Makefile` and automate the rendering and shader compiling:

```
# Makefile

SHADER = shader
RENDER = render
EXT = slo

all: test.tiff

myLight.$(EXT): myLight.sl
```

¹⁸The camera is looking down the positive z-axis by default.

```

$(SHADER) myLight.sl

mySurface.$(EXT): mySurface.sl
$(SHADER) mySurface.sl

test.tiff: test.rib myLight.$(EXT) mySurface.$(EXT)
$(RENDER) test.rib

clean:
    -rm -f *~ *.$(EXT) test.tiff

```

Now you just type `make` to compile the shaders and render a picture called `test.tiff`. Go back to the RIB file and change the lines where the shaders are called to use your own light and surface shader:

```

...
    LightSource "myLight" 0
    AttributeBegin
        Surface "mySurface"
    ...

```

We are ready to create the two shaders. For a start we just print some information that the shader was called. That's why we reduced the image resolution to create only a few lines of output. The light shader should look like this:

```

/* myLight */

light
myLight()
{
    printf("myLight\n");
}

```

The surface shader looks very similar:

```

/* mySurface */

surface
mySurface()
{
    printf("mySurface\n");
}

```

But if you render the image only output from the surface shader is printed. That means the light shader never gets called. OK, maybe we should use the output variables. Leave the `printf` line as it is but add other lines to make the surface shader act like the standard `constant` shader:

```

/* mySurface */

surface
mySurface()
{
    printf("mySurface\n");
    Oi = Os;
    Ci = Os * Cs;
}

```

The light shader should be similar to the standard `ambientlight`:

```

/* myLight */

light
myLight()
{
    printf("myLight\n");
    Cl = color (1.0, 1.0, 1.0);
}

```

We don't use shader parameters at the moment. Most of the time you will start with fixed values in your shaders and make them step by step available as shader parameters later to give the users of your shaders the possibility to change things from outside your shader without changing the shader itself anymore.

If you run the test again you will notice that the light shader still is not called. Why? Well, the surface shader has to call some functions to invoke the light shader. Let's replace the last line in the **surface** shader:

```

Ci = Os * Cs * ambient();

```

Our ambient light source shader will now be called. The light shader acts like an ambient light source because it does **not** use a `illuminate` or `solar` statement. See section 5.1.

The output of the shaders might look different for several renderers. For **PRMan** first only calls to the surface shader are made; after that only calls to the light shader are made¹⁹. For **BMRT** or **AIR** the calls are intermixed; one call to the surface shader, one call to the light shader, etc. This shouldn't concern you too much as long as both shaders create the same output **image** for all RenderMan compliant renderers. But it shows for the first time that things might be handled differently for all this RenderMan compliant renderers.

¹⁹A possible reason for this might be the intention to use SIMD — Single-Instruction Stream Multiple-Data Stream — architectures for rendering.

For a complete list of **Shading and Lighting Functions** I refer to the *The RenderMan Interface* document from Pixar. Here just a list of functions without further details²⁰:

- **ambient()**: See for example the standard surface shaders `matte`, `metal`, `shinymetal`, `plastic`, or `paintedplastic`.
- **diffuse(normal N)**: See for example the following standard surface shaders `matte`, `plastic`, or `paintedplastic`.
- **specular(normal N; vector V; float roughness)**: See for example the standard surface shaders `metal`, `shinymetal`, `plastic`, or `paintedplastic`.
- **specularbrdf(vector L; normal N; vector V; float roughness)** allows users to write an illuminance loop that reproduces the functionality of the `specular()` function, even if the renderer has an implementation-specific formula for built-in specular reflection.
- **phong(normal N; vector V; float size)** implements the Phong specular lighting model.
- **trace(point P, point R)**: See for example Pixar's `glassrefr` shader.

Now let's have a look at some of this functions and talk about something else which is important in a real production pipeline. It's very convenient to render several passes of a scene and do adjustments afterwards with a compositing software.

Set the resolution back to the values it had before, get rid of the `printf()` statements, and change the surface shader to:

```
/* mySurface */

surface
mySurface(float roughness = 0.1)
{
    normal Nf = faceforward(normalize(N), I);

    Oi = color 1;
    Ci = (color(1, 0, 0) * ambient() +
          color(0, 1, 0) * diffuse(Nf) +
          color(0, 0, 1) * specular(Nf, normalize(-I), roughness));
}
```

The result is not very exciting because the light shader still returns only the ambient lighting. Let's change that²¹:

²⁰All this functions do return a color.

²¹That's the standard **pointlight** shader by the way.

```

/* myLight */

light
myLight(
    float intensity = 1;
    color lightcolor = 1;
    point from = point "shader" (0,0,0);
)
{
    illuminate(from)
    Cl = intensity * lightcolor / L.L;
}

```

Basically you separate and color encode the ambient, diffuse, and specular contribution. The ambient term is in the red component, the diffuse term in the green component, and the specular in the blue component. In the compositing program you can separate the three components into three grey scale pictures, multiply it with any color you want, apply filters e.g. blur the highlight from the specular contribution more in x-direction than in the y-direction etc. This can save a lot of rendering time for a complicated geometry to match the lighting and coloring conditions in life action. You could do that in theory for every light. I use it here only to show you that you should keep the post-production in mind and that you might use basically the same RIB file for several passes and modify certain parts to create several pictures which are useful for compositors.

The example scene is not very interesting. So replace the linear patch through a sphere and add an ambient light source to get an extra light contribution:

```

# test.rib
Exposure 1.0 2.2
Display "test.tiff" "file" "rgba"
Format 508 380 1.0
Projection "perspective" "fov" [ 90 ]
WorldBegin
    LightSource "ambientlight" 0
    LightSource "myLight" 1
    AttributeBegin
        Surface "mySurface"
        Translate 0 0 2
        Sphere 1 -1 1 360
    AttributeEnd
WorldEnd

```

Exercises:

1. Try to use the separating into passes with real production scenes.
2. Try to match the lighting of a photograph with real actors or buildings by adding virtual components.

3.2.2 Mental Ray

Now let's make some experiments with **mental ray**. For light shaders please read chapter 3.13 of the book [5], for material shaders please read chapter 3.8 of the same book. Here is a simple scene similar to the RenderMan example in mental ray's MI file format:

```

verbose off
link "base.so"
$include <base.mi>
link "contour.so"
$include <contour.mi>
link "physics.so"
$include <physics.mi>

options "opt"
    samples          -1 2
    contrast         0.1 0.1 0.1
    object space
end options

camera "Camera"
    frame            1
    output            "rgb" "test.rgb"
    focal             12.0
    aperture          32.0
    aspect            1.33333333333
    resolution        508 380
end camera

instance "Camera_inst" "Camera"
    transform
        1.0 0.0 0.0 0.0
        0.0 1.0 0.0 0.0
        0.0 0.0 1.0 0.0
        0.0 0.0 0.0 1.0
end instance

light "Lamp"
    "mib_light_point" (
        "color" 1.0 1.0 1.0,
        "shadow" false,
        "factor" 1,
        "atten" false,
        "start" 0,
        "stop" 1
    )
    origin 0.0 0.0 0.0
end light

instance "Lamp_inst" "Lamp"
    transform
        1.0 0.0 0.0 0.0
        0.0 1.0 0.0 0.0
        0.0 0.0 1.0 0.0
        0.0 0.0 0.0 1.0
end instance

material "mtl"
    opaque
    "mib_illum_lambert" (
        "ambient" 0.0 0.0 0.0,
        "diffuse" 0.8 0.8 0.8,
        "ambience" 0.0 0.0 0.0,
        "mode" 1,
        "lights" [ "Lamp_inst" ]
    )
end material

object "Plane"
    visible trace shadow
    tag 1
    group
        1.0 1.0 0.0
        1.0 -1.0 0.0
        -1.0 -1.0 0.0
        -1.0 1.0 0.0

```

```

        v 0
        v 1
        v 2
        v 3
        p "mtl" 0 3 2 1
    end group
end object

instance "Plane_inst" "Plane"
    transform
        1.0 0.0 0.0 0.0
        0.0 1.0 0.0 0.0
        0.0 0.0 1.0 0.0
        0.0 0.0 1.0 1.0
end instance

instgroup "rootgrp"
    "Camera_inst"
    "Lamp_inst"
    "Plane_inst"
end instgroup

render "rootgrp" "Camera_inst" "opt"

```

You should check that this MI file renders fine with your local mental ray installation before you proceed. Here is a **Makefile** which can be used to render the scene by simply typing `make` in your Unix shell. Later we will add commands to compile the shaders etc. but for now the `Makefile` looks like this:

```

# Makefile

MENTALRAY      = /usr/local/mentalray
RENDER         = $(MENTALRAY)/linux-x86/bin/ray
INCLUDE        = $(MENTALRAY)/common/include
LDPATH         = $(MENTALRAY)/linux-x86/shaders
EXT            = so

all: test.rgb

test.rgb: test.mi
    $(RENDER) -include_path $(INCLUDE) -ld_path $(LDPATH) test.mi

clean:
    -rm -f *~ *.$(EXT) test.rgb

```

Let's change the light shader first. This way we are sure that the material shader does call the light shader and later we can replace the material shader as well. We don't need shadows at the moment. The light shader looks like this:

```

/* myLight.c */

#include <shader.h>

struct myLight
{
    miColor color; /* color of the light source */
};

```

```

DLLEXPORT miBoolean
myLight(
    register miColor* result,
    register miState* state,
    register struct myLight *paras
    )
{
    *result = *mi_eval_color(&paras->color);
    return(miTRUE);
}

```

To compile the shader I changed the Makefile:

```

# Makefile

MENTALRAY      = /usr/local/mentalray
RENDER         = $(MENTALRAY)/linux-x86/bin/ray
INCLUDE        = $(MENTALRAY)/common/include
LDPATH         = $(MENTALRAY)/linux-x86/shaders
OBJEXT         = o
SOEXT          = so
CC             = gcc
LINK           = ld
DEFINES        = -DLINUX -DLINUX_X86 -DX86 -DEVIL_ENDIAN -D_GNU_SOURCE \
-D_REENTRANT -DSYSV -DSVR4 -Dinline=__inline__
CFLAGS         = -ansi -fPIC -O3 -mpentiumpro -fexpensive-optimizations \
-finline-functions -funroll-loops -fomit-frame-pointer -frerun-cse-after-loop \
-fstrength-reduce -fforce-mem -fforce-addr $(DEFINES)
INCPATH        = -I. -I$(INCLUDE)

all: test.rgb

myLight.$(OBJEXT): myLight.mi myLight.c
    $(CC) -c $(CFLAGS) $(INCPATH) -o myLight.$(OBJEXT) myLight.c

myLight.$(SOEXT): myLight.$(OBJEXT)
    $(LINK) -shared -export-dynamic -o myLight.$(SOEXT) myLight.$(OBJEXT)

test.rgb: test.mi myLight.$(SOEXT)
    $(RENDER) -include_path $(INCLUDE) -ld_path $(LDPATH) test.mi

clean:
    -rm -f *~ *.$(SOEXT) *.$(OBJEXT) test.rgb

```

One thing I forgot to mention is that you need a header file for your shader as well. In this case I named it `myLight.mi` and it looks (so far) like this:

```

# myLight.mi

declare shader
    color "myLight" (color "color")
    version 1
    apply light
end declare

```

We are ready to use the light shader now. So please edit the original MI file and add respectively change the following lines:

```

...
link "myLight.so"
#include <myLight.mi>
...

```



```

light "Lamp"
    "myLight" (
        "color" 1.0 1.0 1.0
    )
    origin 0.0 0.0 0.0
end light
...

```

Compile the shader with `make myLight.so` and copy (or softlink) the files `myLight.so` and `myLight.mi` to a place where mental ray can find it before you render. For my local installation the compiled files go to `/usr/local/mentalray/linux-x86/shaders` and the header files can be found in `/usr/local/mentalray/common/include`. You could also specify the full path to your files in the MI file.

It's time to change the material shader. We add a few more lines to the Makefile:

```

...
mySurface.$(OBJEXT): mySurface.mi mySurface.c
    $(CC) -c $(CFLAGS) $(INCPATH) -o mySurface.$(OBJEXT) mySurface.c

mySurface.$(SOEXT): mySurface.$(OBJEXT)
    $(LINK) -shared -export-dynamic -o mySurface.$(SOEXT) mySurface.$(OBJEXT)

test.rgb: test.mi myLight.$(SOEXT) mySurface.$(SOEXT)
    $(RENDER) -include_path $(INCLUDE) -ld_path $(LDPATH) test.mi
...

```

The header file for the new material shader looks like this:

```

# mySurface.mi

declare shader
    color "mySurface" (
        color          "ambient",
        color          "diffuse",
        color          "specular",
        scalar         "spec_exp",
        integer        "mode",
        array light     "lights"
    )
    version 1
    apply material
end declare

```

The material shader takes three colors as parameters for the ambient, diffuse, and specular color contributions. In the **RenderMan** shader I just set them in the latest version of the shader without giving the user the choice to change the values from outside (in the RIB file). Go back and make the modifications needed, if you want to. The **mental ray** material shader itself needs a bit of explanation:

```

/* mySurface.c */

#include <shader.h>

struct mySurface
{
    miColor ambient; /* ambient color */
    miColor diffuse; /* diffuse color */
    miColor specular; /* specular color */
    miScalar spec_exp; /* Phong exponent */
    int mode; /* light mode: 0..2 */
    int i_light; /* index of first light */
    int n_light; /* number of lights */
    miTag light[1]; /* list of lights */
};

DLLEXPORT miBoolean
mySurface(
    miColor* result,
    miState* state,
    struct mySurface* paras
)
{
    int i;
    int n_light; /* number of light sources */
    int i_light; /* offset of light sources */
    int samples;
    miColor* diffuse;
    miColor* specular;
    miColor color;
    miColor sum;
    miScalar spec_exp;
    miScalar dot_nl;
    miScalar phong_specular;
    miTag* light;
    miVector dir;

    /* ambient */
    *result = *mi_eval_color(&paras->ambient);
    diffuse = mi_eval_color(&paras->diffuse);
    specular = mi_eval_color(&paras->specular);
    spec_exp = *mi_eval_scalar(&paras->spec_exp);
    n_light = *mi_eval_integer(&paras->n_light);
    i_light = *mi_eval_integer(&paras->i_light);
    light = mi_eval_tag(paras->light) + i_light;
    for (i = 0; i < n_light; i++)
    {
        sum.r = sum.g = sum.b = 0;
        samples = 0;
        while (mi_sample_light(&color, &dir, &dot_nl, state, *light, &samples))
        {
            /* diffuse */
            sum.r += dot_nl * diffuse->r * color.r;
            sum.g += dot_nl * diffuse->g * color.g;
            sum.b += dot_nl * diffuse->b * color.b;
            /* specular */
            phong_specular = mi_phong_specular(spec_exp, state, &dir);
            if (phong_specular > 0.0)
            {
                sum.r += phong_specular * specular->r * color.r;
                sum.g += phong_specular * specular->g * color.g;
                sum.b += phong_specular * specular->b * color.b;
            }
        } /* while (mi_sample_light(...)) */
        if (samples)
        {
            result->r += sum.r / samples;
            result->g += sum.g / samples;
            result->b += sum.b / samples;
        } /* if (samples) */
    }
    return(miTRUE);
}

```

First of all you will notice a difference between the header file and the structure used in the C file. Every parameter translates easily from the description in the header file to the C equivalent. Except the light array which results in three variables in C.

The ambient contribution does not come from a call to evaluate all ambient light sources like in **RenderMan** but is simply an input parameter. Which means that every material shader which does take ambient contribution into account should have a parameter to let the user set the color etc.

The material shader loops over a light list which comes from the input parameter as an array of lights. There is no such looping mechanism as the `illuminate` statement in **RenderMan**.

Area lights are taken into account by the loop using `mi_sample_light` because that function must be called in a loop until it returns `miFALSE`. Every sample of the light is summing up within the loop. That's why you have to divide by the number of samples. For lights which are not area lights there should be only one sample taken. But be careful not to divide by zero.

The result of the call `mi_sample_light` to the light shader will be in the variable `color`. There are two other return values provided if the pointers are nonzero: `dir` is the direction from the current intersection point in the state to the light, `dot_nl` is the dot product of this direction and the normal in the state.

For the specular contribution I took the Phong factor but there are other choices²²:

- **mi_phong_specular**: Calculate the Phong factor based on the direction of illumination `dir`, the specular exponent `spec_exp`, and the state variables `normal` and `dir`.
- **mi_fresnel_specular**: Calculate the specular factor `ns` based on the illumination direction `dir`, the specular exponent `spec_exp`, the inside and outside indices of refraction `ior_in` and `ior_out`, and the state variables `normal` and `dir`.
- **mi_cooktorr_specular**: Calculate the specular color `result` according to the Cook–Torrance reflection model for the incident direction `dir_in` and the reflection direction `dir_out` at a surface with normal `normal`. The `roughness` is the average slope of surface microfacets. `ior` is the relative index of refraction for three wavelengths.
- **mi_blinn_specular**: Like `mi_cooktorr_specular`, but only for one wavelength.
- **mi_blong_specular**: This is similar to `mi_blinn_specular`, but implements a hybrid of Blinn and Phong shading instead of true Blinn

²²See **mental ray** documentation of this function calls for the types of the parameters needed.

shading. It is included separately to support the Softimage Blinn shading model.

Finally we make some adjustments to the MI file to render a sphere:

```
...
link "myLight.so"
$include <myLight.mi>
link "mySurface.so"
$include <mySurface.mi>
...
light "Lamp"
    "myLight" (
        "color" 1.0 1.0 1.0
    )
    origin 0.0 0.0 0.0
end light
...
material "mtl"
    opaque
    "mySurface" (
        "ambient" 1.0 0.0 0.0,
        "diffuse" 0.0 1.0 0.0,
        "specular" 0.0 0.0 1.0,
        "spec_exp" 40,
        "mode" 1,
        "lights" ["Lamp_inst"]
    )
end material
...
$include "sphere.mi"

instance "sphere_inst" "sphere"
    transform
        0.2 0.0 0.0 0.0
        0.0 0.2 0.0 0.0
        0.0 0.0 0.2 0.0
        0.0 0.0 2.0 0.2
    material "mtl"
end instance

instgroup "rootgrp"
    "Camera_inst"
    "Lamp_inst"
    "sphere_inst"
end instgroup

render "rootgrp" "Camera_inst" "opt"
```

The MI file `sphere.mi` comes with the examples of the book [4] and you can download the examples and all shaders from mental images' FTP site. I made a softlink to this file at a location where **mental ray** can find it. In contrast to **RenderMan** and other renderers **mental ray** does not support other primitives beside the usual polygons, NURBS surfaces, and subdivision surfaces. There is one exception as you can see in figure 4.



Figure 4: The hair primitive

3.2.3 A Simple Scene With Five Different Shaders

Let's create a very simple test scene and write all kind of shaders for it. In figure 5 you see a torus in front of the camera. The right half of the background leaves space for our imager shader, the left half is covered by a bilinear patch. We will use the same scene for **RenderMan** and **mental ray**.

Let's start with **RenderMan**. The RIB file is so simple that I will explain it a bit and tell you where to make modifications to integrate the shaders step by step. The original RIB file looks like this:

```
# shadertest.rib
Exposure 1.0 2.2
Display "shadertest.tiff" "file" "rgba"
Format 508 380 1.0
Projection "perspective" "fov" [ 90 ]
LightSource "pointlight" 0 "intensity" [ 1 ]
WorldBegin
  Surface "plastic"
# Patch
  AttributeBegin
    Translate -10 0 10
    Scale 10 10 10
    Patch "bilinear" "p"
      [ -1.0 -1.0 0.0 1.0 -1.0 0.0 -1.0 1.0 0.0 1.0 1.0 0.0 ]
  AttributeEnd
# Torus
  AttributeBegin
    Translate 0 1 5
    Rotate 45 1 0 0
    Torus 3.0 1.0 0.0 360.0 360.0
  AttributeEnd
WorldEnd
```

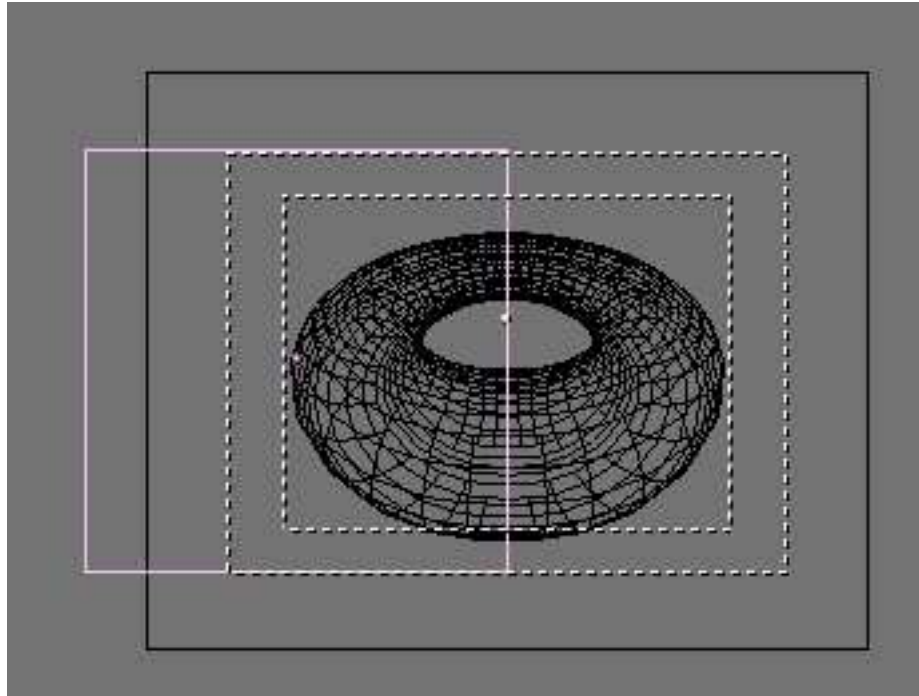


Figure 5: Simple test scene

The scene defines a light source outside the `WorldBegin` and `WorldEnd` block. At the moment we use one of the standard shaders `pointlight` for the light source shader. Within the `WorldBegin` and `WorldEnd` block we define the bilinear patch for the background and the torus for the foreground. The geometry is defined within an `AttributeBegin` and `AttributeEnd` block. The transformations are defined within this blocks because we don't want them to affect the other geometry. The effect is limited to this block and restored outside. The geometry itself is more or less defined in a single line. The bilinear patch uses 4 points with x-, y-, and z-coordinates. The torus is defined in a line like this:

```
Torus rmajor rminor phimin phimax thetamax ...parameterlist...
```

The major radius `rmajor` defines the distance of a circular arc which could be used to define the torus by rotating this cross section around the z-axis. The minor radius `rminor` is the radius of the circular arc and the angles are used in a way that we define a whole torus instead of the more general definition.

If you render the RIB file with a RenderMan compliant renderer the alpha

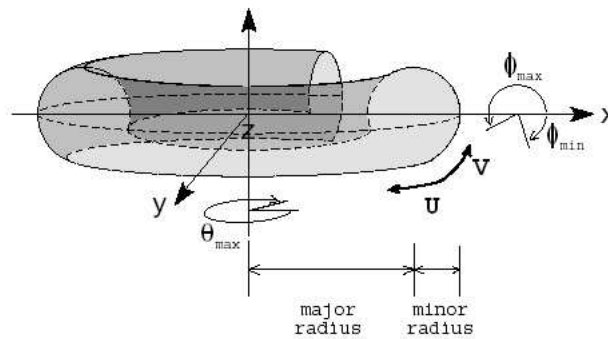


Figure 6: RenderMan's torus primitive

channel will “cut out” the part of the background which is **not** covered by the bilinear patch. Now let's add the following line before `WorldBegin`:

```
...
Imager "background" "background" [ 1 0 0 ]
...
WorldBegin
...
```

This will fill the background which was “cut out” before with a red color. We will write our own **imager shader** now. So what information do we have within an imager shader? Let's try to visualize some of this information. Table 4 on page 74 shows a full list of predefined imager shader variables.

Please modify the RIB file to use our own shader. You should know by now how to do this. The shader looks at the moment like this:

```
/* myImager */

imager
myImager()
{
    Ci = color(0.0, 1.0, 0.0);
    Oi = 1;
    alpha = 1;
}
```

Add the proper lines to your Makefile and render the scene. Well, the result is not what we want — the whole picture is green now. If we take the alpha value into account we could create a simple “contour shader”:

```
/* myImager */
```

```

imager
myImager()
{
    if (alpha == 0.0) Ci = color(0.0, 1.0, 0.0);
    else if (alpha != 1.0) Ci = color(1.0, 0.0, 0.0);
    Oi = 1;
    alpha = 1;
}

```

Let's not talk about if a shader like this would be useful or not. What I wanted to show is that shaders are kind of hard to debug. In theory you can create output lines within a shader, analyze these lines with another program, visualize them etc. but the simplest way is to create a test scene where a certain color like red does not occur and use that color to indicate problem areas.

The next version of the imager shader will show how to access some information which does **not** come from predefined surface shader variables²³:

```

/* myImager */

imager
myImager()
{
    float xyp[3] = { 1.0, 1.0, 1.0 };
    option("Format", xyp);
    Ci = color(xcomp(P) / xyp[0], ycomp(P) / xyp[1], 0);
    Oi = 1;
    alpha = 1;
}

```

We get access to the `Format` line in the RIB file and extract the `x`- and `y`-resolution of the rendered picture from there. We use that information to scale the pixel coordinates stored in `P` so that the result is between zero and one. We use that to color encode the coordinates in the background image.

It's hard to find a real application for an imager shader. I leave it for you as an exercise to find one. In Pixar's document you will find the standard background shader and an example how to write an imager shader to implement the exposure and quantization process:

```

imager
exposure(float gain=1.0, gamma=1.0, one = 255, min = 0, max = 255)
{
    Ci = pow(gain * Ci, 1/gamma);
    Ci = clamp(round(one * Ci), min, max);
    Oi = clamp(round(one * Oi), min, max);
}

```

²³See chapter 15.8 "Message Passing and Information Functions" of the Pixar's document *The RenderMan Interface*.

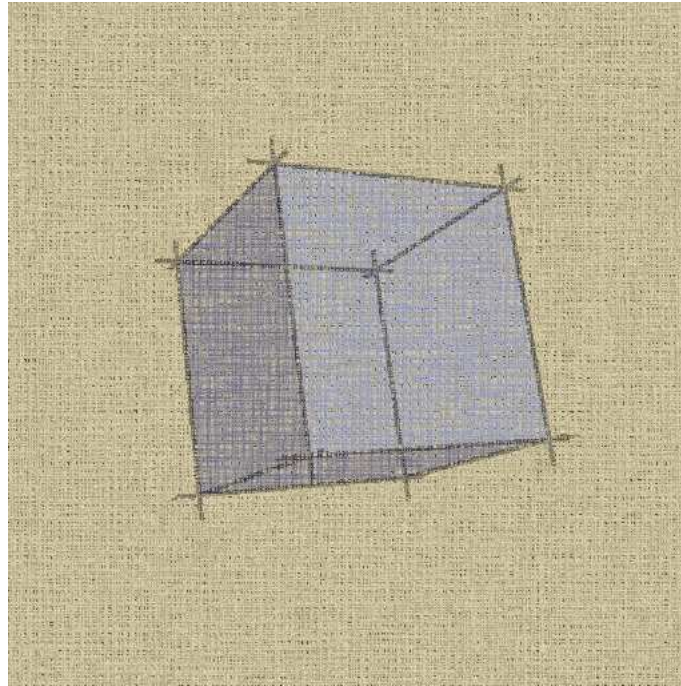


Figure 7: A canvas imager shader for BMRT

Beside the fact that this shader will not compile it's still an example that an imager shader could be used to do color correction. I think there are not too many applications for an imager shader but one interesting experiment can be found on the following web page from Katsuaki Hiramitsu:

http://www.edit.ne.jp/~katsu/img_index.htm

Unfortunately the shader does not compile for other renderers beside BMRT and it has the “knowledge” about the scene within the imager shader. Anyway, maybe worth to look at it in figure 7.

At least an imager shader can be used to develop patterns or to visualize procedural textures without having any geometry or lights in the RIB file. Once you are happy with your pattern you can apply a similar line in a surface shader and get rid of the imager shader.

Let's just create a simple pattern — a disk — as described in the book [3]. In chapters 20 to 22 you find more information about patterns but let's just develop one pattern in the imager shader and apply it later for a surface shader. Here is the source code of the imager shader:

```

/* myImager */

imager
myImager()
{
    /* some of this variables should be shader parameters */
    float xyp[3];
    float xcoord, ycoord;
    float dist;
    float inDisk;
    float radius = 0.4;
    point centre = point(0.5, 0.5, 0.0);
    point here;
    color inside = color(0, 0, 1);
    color outside = color(1, 0, 0);
    /* get x- and y-resolution from global options */
    option("Format", xyp);
    /* make sure the coordinates are between 0 and 1 */
    xcoord = xcomp(P) / xyp[0];
    ycoord = ycomp(P) / xyp[1];
    here = point(xcoord, ycoord, 0.0);
    /* how far are we away from the centre? */
    dist = distance(centre, here);
    /* are we inside the disk? */
    if (dist <= radius) inDisk = 1.0;
    else inDisk = 0.0;
    /* use inside or outside color */
    Ci = mix(outside, inside, inDisk);
    /* we don't need the alpha channel for this test */
    Oi = 1;
    alpha = 1;
}

```

Most of the shader is straight forward and I put some comments inside the shader. There are two functions we haven't talked about yet. The function `distance` returns the Euclidean distance between two point. You could have defined a vector from `centre` to `here` and call the function `length` for this vector or you could have calculated the Euclidean distance yourself with help of the function `sqrt`²⁴. The other function is called `mix` and takes two colors and a float value to mix both colors with the following formula:

$$(1 - a) * color_1 + a * color_2$$

Before we apply this to a surface shader we make some modifications to mix the two colors slowly. The following output comes from a Unix program called `diff` which shows only the difference between two file versions:

```

11a12
>   float fuzz = 0.1;
14c15
<   color inside = color(0, 0, 1);
---
```

²⁴This is the square root.

```

> color inside = color(1, 1, 0);
25,26c26
< if (dist <= radius) inDisk = 1.0;
< else inDisk = 0.0;
---
> inDisk = 1 - smoothstep(radius - fuzz, radius + fuzz, dist);

```

I defined a new variable called `fuzz` and changed the `outside` color from blue to yellow. It looks nicer to mix the colors yellow and red instead of blue and red. Normally you would use the color set by the user in the RIB file as outside color and the inside color would be a shader parameter but we will do that in a minute. The last change I made is that I use a function called `smoothstep` which defines a range with its first two parameters and ramps smoothly in between. So if our current position here is less than the `radius` minus our `fuzz` distance then we are completely inside the disk and use the `inside` color only. If the current position is greater than our `radius` plus the `fuzz` distance we use the outside color only. Every position in between mixes both colors in a way that it slowly fades from one color to the other.

Now let's apply this technique to a **surface shader**. We have to change a few things to create a surface shader from the imager shader we got so far:

```

1c1
< /* myImager */
---
> /* mySurface */
3,4c3,4
< imager
< myImager()
---
> surface
> mySurface()
7,8d6
< float xyp[3];
< float xcoord, ycoord;
17,18d14
< /* get x- and y-resolution from global options */
< option("Format", xyp);
20,22c16
< xcoord = xcomp(P) / xyp[0];
< ycoord = ycomp(P) / xyp[1];
< here = point(xcoord, ycoord, 0.0);
---
> here = point(s, t, 0.0);
28,31c22,23
< Ci = mix(outside, inside, inDisk);
< /* we don't need the alpha channel for this test */
< Oi = 1;
< alpha = 1;
---
> Oi = Os;
> Ci = Os * mix(outside, inside, inDisk);

```

First of all we rename the shader to `mySurface`, we change the type of the shader to `surface`, and we get rid of a few variables we don't need anymore because we replace the `x`- and `y`-coordinates of the image shader by the texture coordinates `s` and `t`. We leave the opacity as it is and apply the color without taking any lighting into account²⁵.

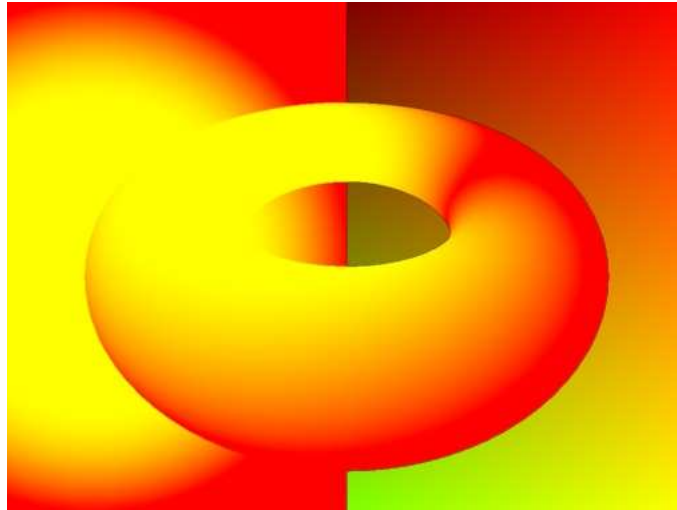


Figure 8: Test scene with imager shader and surface shader

Before we can render the new scene we change the RIB file to use the new surface shader:

```
9c9
< Surface "plastic"
---
> Surface "mySurface"
```

See figure 8 for the resulting image. I want to change a few things before I continue with other shader types. Read chapter 22 of the book [3] to understand the concept of **Tiling and Repeating Patterns**. I want the disk pattern to be repeated several times over the surfaces and I want to take the lighting into account:

```
4c4,5
< mySurface()
---
> mySurface(float Ka = 1, Kd = 0.5, Ks = 0.5, roughness = 0.1;
>           color specularcolor = 1)
10a12
```

²⁵You can test this by removing the light source shader from the RIB file.

```

> float ss, tt;
15,16c17,23
< /* make sure the coordinates are between 0 and 1 */
< here = point(s, t, 0.0);
---
> color myCs;
> normal Nf = faceforward(normalize(N), I);
> /* repeat the pattern */
> ss = mod(s*10, 1);
> tt = mod(t*10, 1);
> /* use ss and tt instead of s and t */
> here = point(ss, tt, 0.0);
23c30,32
< Ci = Os * mix(outside, inside, inDisk);
---
> myCs = Os * mix(outside, inside, inDisk);
> Ci = Os * (myCs * (Ka * ambient() + Kd * diffuse(Nf)) +
>           specularcolor * Ks * specular(Nf, normalize(-I), roughness));

```

The easiest thing to do for the lighting is to take parts from the source code of an already existing shader like the `plastic` shader and change a few lines in the old version. The shader takes now parameters which are identical to the `plastic` shader. Instead of assigning the color directly to the `Ci` output variable I use a new variable named `myCs` for my surface color.

For the repeating disk pattern I introduce the variables `ss` and `tt`. The most important lines for the repeating pattern are the two lines using the `mod` function. This so-called **modulo** function takes the first argument and divides it by the second one. The remainder of the division is returned. In this case the remainder is the part behind the dot. The return value of `mod(3.7, 1)` is 0.7. By scaling `s` and `t` by 10 before we use the `mod` function we make sure that the disk is repeated ten times in each direction but `ss` and `tt` still vary between zero and one.

The resulting image looks far too dark. Instead of changing the surface shader we are writing our own **light shader**. First we will not take any attenuation over distance into account:

```

/* myLight */

light
myLight()
{
    Cl = color 1;
}

```

Let's render a picture with this three shaders. You have to change the RIB file before your start rendering:

```

7c7
< LightSource "pointlight" 0 "intensity" [ 1 ]
---
> LightSource "myLight" 0

```

Do you remember section 3.2? This is an ambient light and the surface shader calls it because of the `ambient()` function which is multiplied by one because of the default value of the shader parameter `Ka = 1`. Let's try another light source shader which fades from `near` to `far` linearly between full intensity to black:

```
/* myLight */

light
myLight(float intensity = 1, near = 1, far = 10;
        color lightcolor = 1;
        point from = point "shader" (0,0,0))
{
    float len, brightness;

    illuminate (from)
    {
        len = length(L);
        if (len < near) brightness = 1;
        else if (len > far) brightness = 0;
        else brightness = 1 - (len - near) / (far - near);
        Cl = intensity * lightcolor * brightness;
    }
}
```

If we render again the torus will be lit but the linear patch is in the dark. You have to modify the RIB file again to change the default parameter values for the light source shader:

```
7c7,9
< LightSource "myLight" 0
---
> Declare "near" "float"
> Declare "far" "float"
> LightSource "myLight" 0 "near" [ 3 ] "far" [ 15 ]
```

Now a bit of the linear patch is lit but **not** all as you can see in figure 9. We didn't implement a spotlight shader even though it looks a bit like one. Change for a moment the light shader:

```
13c13
<         if (len < near) brightness = 1;
---
>         if (len < near) brightness = 0;
```

The reason for this is that it's hard to find the border to the near value, but simple for the far border. If we make everything outside the near and far range unlit it's easier to find good parameter settings. This leads me to a new exercise for you. Think about a way to find the closest and furthestmost

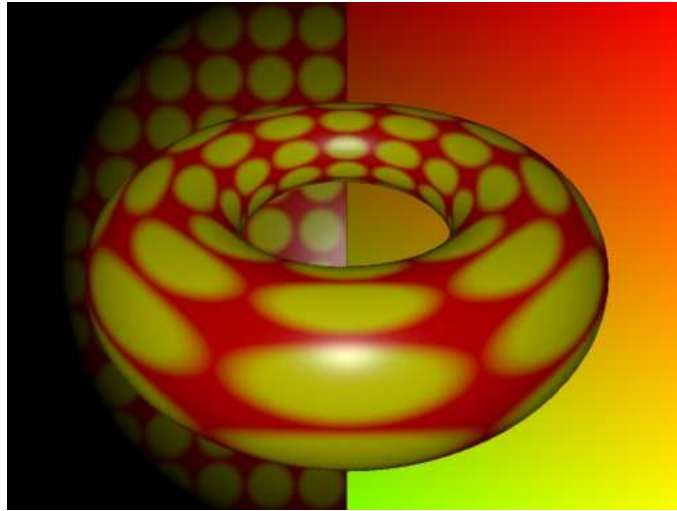


Figure 9: Test scene with imager, surface, and light shaders

point to/from the camera on any surface during rendering. Use this values for the previous light source shader and vary the `near` and `far` values from there.

We will write a **displacement shader** now. A very simple one which just uses the `sin` function along the `s` and `t` texture coordinates to vary the position and the normal before the surface shader gets called:

```
/* myDisplacement */

displacement
myDisplacement ()
{
    float amp = 0.1 * sin(t*20*PI);
    amp += 0.1 * sin(s*20*PI);
    P += amp * normalize(N);
    N = calculatenormal(P);
}
```

Look at figure 10 for the result if you simply change the RIB file to include the displacement shader without specifying additional attributes for the renderer.

```
22a23
> Displacement "myDisplacement"
```

Actually it's dependent on which renderer you are using. BMRT and AIR do use **bump mapping** if you don't tell the renderer to use true displacement. Pixie and Angel use always **true displacement**.

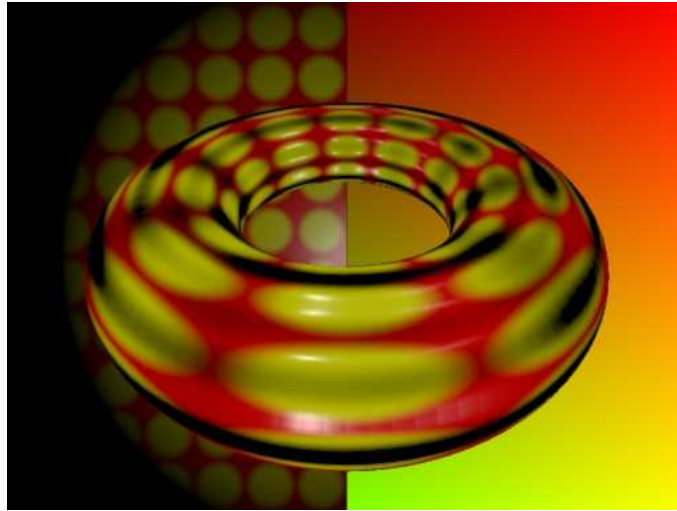


Figure 10: Test scene with bump mapping

Let's compare this to figure 11 which was rendered with the **same** shader but with the following additional attributes:

```
10a11,12
> Attribute "render" "truedisplacement" [1]
> Attribute "displacementbound" "sphere" [1]
```

For bump mapping you get a strange artefact. Some points on the surface are rendered black because the bump mapping bends the normal so much that it's facing away from the light and the camera. The silhouette of the torus is still the original one because the positions on the surface are **not** modified by bump mapping. But for some areas where the surface is far enough away from the camera the bump mapping works quite well. Especially if the displacement is very subtly which is true for a lot of images you will render. For example if you render skin it's nice to have little dents to make the surface look not too perfect but the dents are small in size and they do not perturb the original surface too much.

Exercises:

1. Find a "good" application for an imager shader.
2. Change the last surface shader example so that all parameters which might be important to change from outside are shader parameters with default values.
3. Use the new surface shader parameters to create variations of the same scene by manipulating the RIB file.

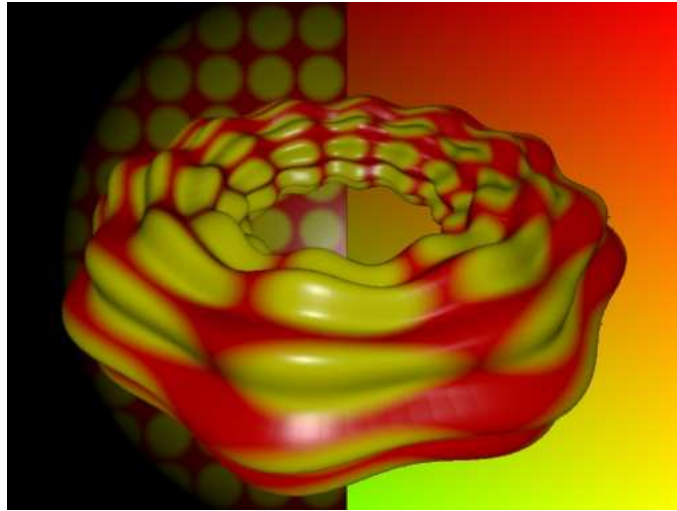


Figure 11: Test scene with true displacement

4. Think about writing a shader to find the closest point to the camera on any surface and the furthestmost point from the camera. This is not only useful for the light shader we talked about above but also for finding good near and far **clipping planes** or to have better values for rendering **shadow maps**.

4 Get Your Hands Dirty

4.1 Default Surface Shader

In section 3 I mentioned a **RenderMan** shader called `defaultsurface` which allows to render an image without any light and surface shaders used in the RIB file. We want to write a similar shader for **mental ray** now. Let's have a look how **AIR**²⁶ implements the `defaultsurface` shader:

```
surface defaultsurface (float Ka=.2, Kd=.8)
{
    Oi=Os;
    Ci=Os*Cs*(Ka+Kd*abs(normalize(I).normalize(N)));
}
```

The implementation of the mental ray shader is straight forward. The only difference is that mental ray has no global state for the current surface color

²⁶Be aware that each implementation might use a different shader.

and opacity. The RenderMan shader takes the values of `Os` and `Cs` into account. This color values can be set in the RIB file by using `Opacity` or `Color`. If you want a similar behaviour within the mental ray shader you should use additional shader parameters for the color and opacity values. Here is the header file for the mental ray shader:

```
# myDefaultSurface.mi

declare shader
    color "myDefaultSurface" (
        scalar          "Ka",
        scalar          "Kd"
    )
    version 1
    apply material
end declare
```

To use this shader in a MI file I changed the example from section 3.1 a bit:

```
...
link "myDefaultSurface.so"
#include <myDefaultSurface.mi>
...
material "mtl"
    opaque
    "myDefaultSurface" (
        "Ka" 0.2,
        "Kd" 0.8
    )
end material
...
object "Monkey"
    visible trace shadow
    tag 1
    group
        0.4375 0.1640625 0.765625
...
        p "mtl" 504 322 320 390
    end group
end object

instance "Monkey_inst" "Monkey"
    transform
        1.0 0.0 0.0 0.0
        0.0 1.0 0.0 0.0
        0.0 0.0 1.0 0.0
        0.0 0.0 5.0 1.0
end instance

instgroup "rootgrp"
    "Camera_inst"
    "Monkey_inst"
end instgroup

render "rootgrp" "Camera_inst" "opt"
```

First of all you have to link your shared library and include the appropriate header file. If you can't copy the compiled shader and the header file to a place where mental ray can find it — e.g. because you don't have root permissions on an Unix system — you can specify the full path in the MI file.

After having defined the shader and its parameters as a material within the MI file you can use the material in the definition of the `Monkey` object. Notice that there is only a camera and one geometry object defined — no lights. The shader itself looks like this:

```
/* myDefaultSurface.c */

#include <shader.h>

struct myDefaultSurface
{
    miScalar Ka;
    miScalar Kd;
};

DLLEXPORT miBoolean
myDefaultSurface(
    miColor* result,
    miState* state,
    struct myDefaultSurface* paras
)
{
    miScalar Ka, Kd;
    miScalar abs_dot_nd;

    /* get parameters */
    Ka = *mi_eval_scalar(&paras->Ka);
    Kd = *mi_eval_scalar(&paras->Kd);
    /* dot product of the normal and the direction vector */
    abs_dot_nd = fabs(state->dot_nd);
    /* result */
    result->r = Ka + Kd * abs_dot_nd;
    result->g = Ka + Kd * abs_dot_nd;
    result->b = Ka + Kd * abs_dot_nd;
    result->a = 1.0;

    return(miTRUE);
}
```

Notice that you don't have to calculate the dot product of the normal and the direction vector yourself because it's already stored in the state.

4.2 Stairs

Consider the following problem: You have a scene with a lot of staircases or for example a huge pyramid. Most of the time the camera shows the

whole scene from distance which means that you can't see the individual steps very well. Instead of modelling each individual step you consider to create a simplified model without steps and faking the effect of having stairs within the shader:

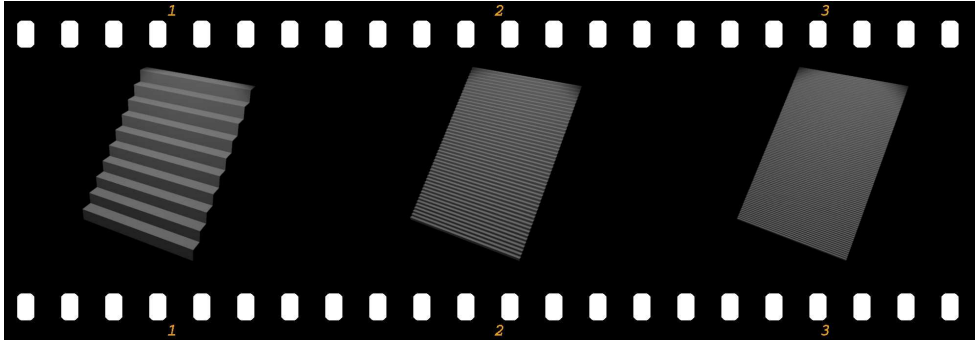


Figure 12: Rendering the stairs with real geometry

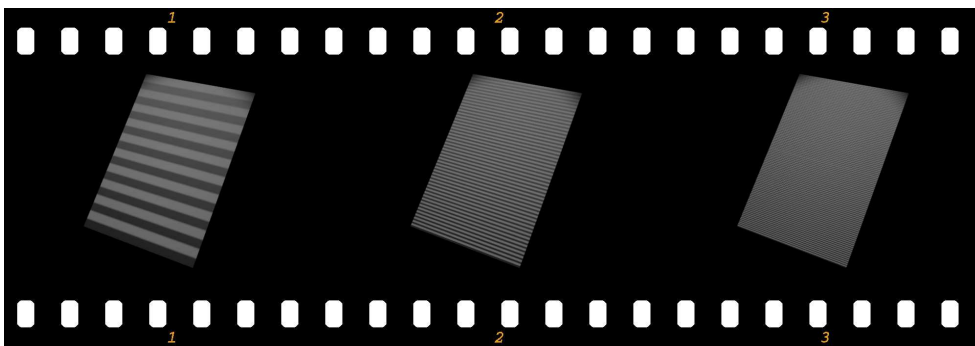


Figure 13: Faking the stairs within the shader

```
/* stairs */

surface
stairs(float Ka = 1, Kd = 0.5, Ks = 0.5, roughness = 0.1;
      color specularcolor = 1;
      float steps = 5;)
{
    normal Nf;
    float sv = mod(v * steps, 1);

    if (sv < 0.5)
        Nf = ntransform("object", "current", normal(0, -1, 0)); /* front */
    else
        Nf = ntransform("object", "current", normal(0, 0, 1)); /* up */

    Oi = Os;
    Ci = Os * (Cs * (Ka * ambient() + Kd * diffuse(Nf)) +
              specularcolor * Ks * specular(Nf, normalize(-I), roughness));
}
```

As you can see in figure 12 I rendered a simple staircase with 10, 50, and 100 steps. I used a Python program to generate the RIB file automatically based on a parameter for the number of steps. On the web page where you downloaded this document or where you read it right now you will find a copy of the Python script I used to generate the staircases. It's called `stairs.py`. You should generate the real geometry and render it with your favourite RenderMan compatible renderer to compare the results with the shader based version using the following RIB file. Change the parameter `steps` for the shader before you render the RIB file again:

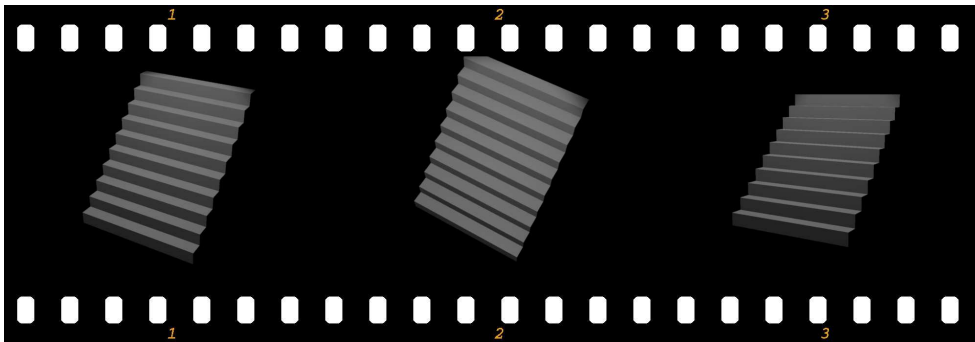


Figure 14: Real geometry from different perspectives

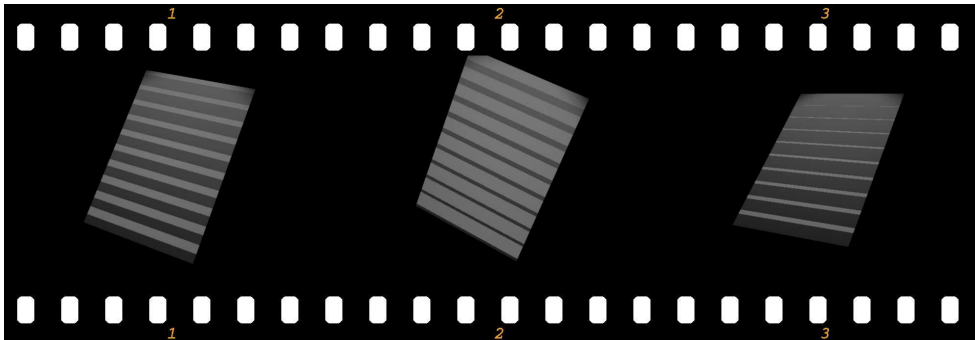


Figure 15: Faked stairs from different perspectives

```
# stairs.shader.rib
Exposure 1.0 2.2
Display "stairs.shader.tiff" "file" "rgba"
Format 640 480 1.0
Projection "perspective" "fov" [ 30 ]
Translate 0 0.25 7
Rotate -30 1 0 0
Rotate 30 0 1 0
Rotate 90 1 0 0
Rotate 180 0 1 0
```

```

WorldBegin
  Surface "plastic"
  LightSource "spotlight" 1
  "color lightcolor" [ 1.0 1.0 1.0 ]
  "point from" [ 0.0 -1.5 5.0 ]
  "point to" [ 0.0 0.0 0.0 ]
  "float intensity" [ 10 ]
  "float coneangle" [ 0.35 ]
# ribPatchMesh
AttributeBegin
  Surface "stairs" "float steps" [ 100 ]
  PatchMesh "bilinear" 2 "nonperiodic" 2 "nonperiodic"
  "P" [ -1.0 -1.0 -1.0
        1.0 -1.0 -1.0
        -1.0 1.0 1.0
        1.0 1.0 1.0 ]
AttributeEnd
WorldEnd

```

You can improve the quality by introducing a fuzz area to get better anti-aliasing:

```

/* stairs */

surface
stairs(float Ka = 1, Kd = 0.5, Ks = 0.5, roughness = 0.1;
      color specularcolor = 1;
      float steps = 4, fuzz = 0.05;)
{
  float p;
  normal Nf;
  normal front = ntransform("object", "current", normal(0, -1, 0));
  normal up = ntransform("object", "current", normal(0, 0, 1));
  float sv = mod(v * steps, 1);

  if (sv < 0.5 - fuzz)
  {
    if (sv > fuzz || (v * steps) <= fuzz)
    {
      Nf = front;
    }
    else
    {
      p = 0.5 + sv / (2 * fuzz);
      Nf = ((1.0 - p) * up + p * front);
    }
  }
  else if (sv > 0.5 + fuzz)
  {
    if (sv < (1.0 - fuzz) || v >= (1.0 - fuzz / steps))
    {
      Nf = up;
    }
    else
    {
      p = (sv - 1.0 + fuzz) / (2 * fuzz);
      Nf = ((1.0 - p) * up + p * front);
    }
  }
}

```

```

else
{
    p = (sv - 0.5 + fuzz) / (2 * fuzz);
    Nf = ((1.0 - p) * front + p * up);
}

Oi = Os;
Ci = Os * (Cs * (Ka * ambient() + Kd * diffuse(Nf)) +
           specularcolor * Ks * specular(Nf, normalize(-I), roughness));
}

```

Exercise:

Try to think about other improvements you could introduce to make this solution available to a wider range of staircases. At the moment I assume that the staircase was modelled a certain way. How can you generalise it? For the case with 100 steps the difference between the real geometry and the shader version is hard to see but this is only true for this perspective from the camera. If you would look more or less straight from the front you should basically see only the front of the steps and nearly nothing from the top. This effect can be seen as well if you look at the top step of the staircase with only 10 steps in figure 12. How can you compensate for that in the shader?

4.3 Simple Room

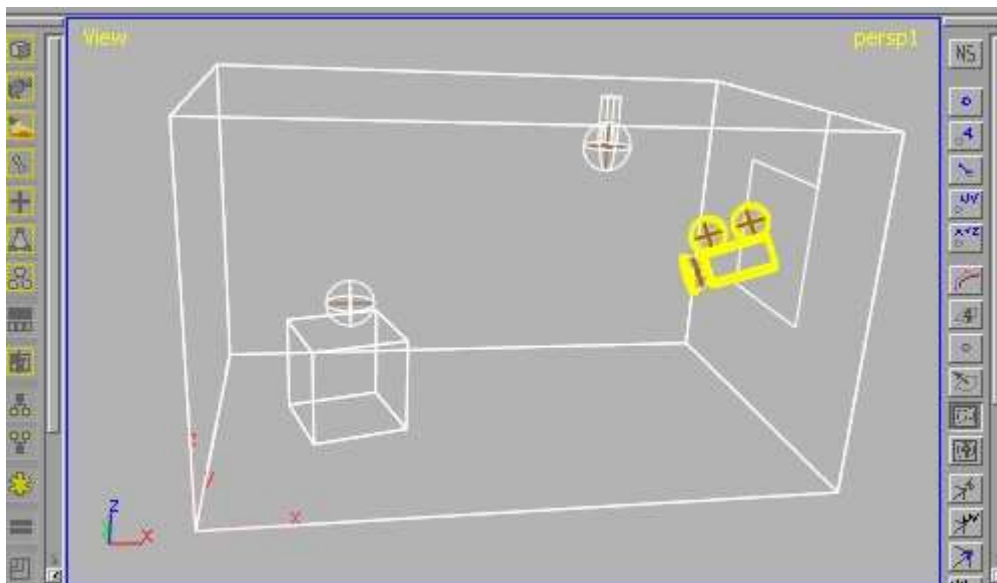


Figure 16: A simple scene in a room with a window

To understand how rendering works it's always a good idea to take an example from one renderer and try to create a similar picture with another renderer. If you can't get away by using standard shaders it might be a good starting point for writing a few shaders.

I started a few years ago a comparison between different renderers and came across a renderer called **Radiance**²⁷. This simple example is based on a tutorial²⁸ for this renderer.

Have a look at figure 16. You will see a simple room with a floor, a ceiling, and four walls. One of the walls has a hole where a window can be placed. The camera looks at two simple objects in the room: A glass sphere and a blue box. The light is coming from two light sources: There is another sphere in the room which acts as a light source and we can think of daylight coming from outside through the window. Although the scene is quite simple it can give you some headache how to produce certain effects²⁹ with your favourite renderer:

1. First of all a glass sphere reflects and refracts light rays. Which makes it hard to fake it with a simple scanline renderer.
2. None of the light sources is directly visible but the spherical light inside the room is shown several times in the glass sphere. Most of the renderers don't show a light source even if you would look directly into the direction where the light is located. You have to take care yourself that there is a geometry which looks like a light bulb or anything which motivates where the light comes from.
3. The light which comes from outside through the window is seen as a direct reflection of the window itself from the front of the sphere, as a refracted and therefore distorted copy on the back side of the sphere, and as a direct reflection of the light which is cast on the floor and one of the walls³⁰.
4. In theory you could have caustics in the scene created by focusing and dispersion of light by specular reflection or refraction but let's talk about that later³¹.

Let's first try to make life easier by using a renderer with ray tracing and a lot of standard shaders coming with it. I reproduced the scene with mental

²⁷<http://radsite.lbl.gov/radiance/HOME.html>

²⁸<http://radsite.lbl.gov/radiance/refer/tutorial.html>

²⁹Look at figure 17.

³⁰This might be visible as a refracted copy somewhere as well.

³¹See section 4.4.



Figure 17: Reflections in a glass sphere (Radiance)

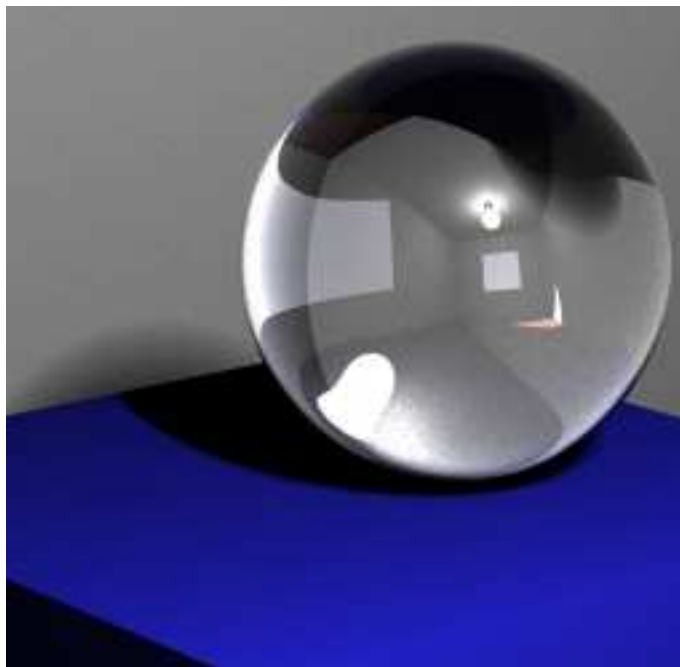


Figure 18: Reflections in a glass sphere (mental ray)

ray and here are some comments about the effects I was talking about before and how they were achieved:

1. The material used in the original Radiance file is called `dielectric` and described in the **Radiance Reference Manual** as: “A *dielectric material is transparent, and it refracts light as well as reflecting it. Its behaviour is determined by the index of refraction and transmission coefficient in each wavelength band per unit length. Common glass has a index of refraction around 1.5, and a transmission coefficient of roughly 0.92 over an inch. An additional number, the Hartmann constant, describes how the index of refraction changes as a function of wavelength. It is usually zero.*” **Mental Ray** comes with a `physics` library and you should be able to get the source code for the shaders from mental images’ FTP site. If you look into the header file³² of the shader called `dielectric_material` you will find all parameters for this material shader. In this case I used the `col` parameter for the light absorption per unit-length and the `ior` parameter for the index of refraction:

```
material "crystal"
  opaque
  "dielectric_material" (
    "ior" 1.5,
    "col" 0.5 0.5 0.5,
    "lights" [ "Lamp_inst", "Sun_inst" ]
  )
end material
```

Ray tracing can be either turned on in the global `options` statement or a shader³³ specifies that it can operate only if ray tracing is enabled. In the later case mental ray will enable ray tracing even if no ray tracing was specified in the global `options` statement.

2. I decided to make the light source inside the room visible by using an area light source:

```
light "Lamp"
  "physical_light" (
    "color" 60 60 60
  )
  visible
  origin 0.0 0.0 0.0
  sphere .125 15 15
end light
```

³²Look either into the MI file called *physics.mi* which should come with mental ray anyway or get the source code and look into the file called *dielecshade.h*.

³³See for example the *physical_lens_dof* lens shader in the *physics* library.

For mental ray the type of a light source is determined by the absence or presence of some parameters. If you define only an `origin` you will have a **point light**. An **infinite light** requires only a `direction` whereas a **spot light** must be given an `origin`, a `direction`, and a `spread` value. Please be aware that the `spread` value is already the **cosine** value of a given angle. Most of the standard shaders for spot lights will have a `cone` parameter which is also given as a **cosine** value instead of specifying the angle in degrees or radians. For RenderMan compatible renderers the `spotlight` shader requires angles in radians and calculates the **cosine** value inside the shader.

There are four different shapes of area light sources: rectangles, flat discs, spheres, and cylinders. Although mental ray 3.1 adds arbitrary geometric objects, and a user-defined shape we can simply use a `sphere` with an radius of 0.125 and 15 samples in `u`- and `v`-direction. To make the light source sphere visible in the reflection and refraction of the glass sphere on the blue box we use the keyword `visible`. Have a look in the source code of light source shaders to see how to deal with rays which hit the geometry of the light source directly. Another aspect of area light sources are **soft shadows**. You don't get soft shadows for the light source inside the room by using Radiance but you do get soft shadows for mental ray. The difference is that Radiance uses always visible light sources but this does **not** necessarily mean that Radiance uses area lights.

3. One thing we can't easily translate into other renderers is the usage of the `illum` material used by Radiance for the window. Have a look again at figure 17 or 18. There are two aspects of the light coming through the window. There is a very bright spot of light cast on the ground and one of the walls, and there is the visibility of the window itself. If you compare figure 17 and 18 more carefully you might notice that you get a reflection of the light source inside the room even in the reflections of the window itself in the image created by Radiance. This additional reflection is missing in the image I created with mental ray.

Let's first read what the `illum` material in Radiance does: *"Illum is used for secondary light sources with broad distributions. A secondary light source is treated like any other light source, except when viewed directly. It then acts like it is made of a different material (indicated by the string argument), or becomes invisible (if no string argument is given, or the argument is "void"). Secondary sources are useful when modelling windows or brightly illuminated surfaces."*

I decided to split the two aspects into two different solutions for mental ray. The very bright spot of light cast on the ground and one of the walls is simulated by adding another light source to the scene which acts like the sun shining from outside through the window on the floor. The sky simulation in Radiance creates the following file called `sky.rad`:

```
# gensky 3 20 10 -a 40 -o 98 -m 105
# Local solar time: 10.34
# Solar altitude and azimuth: 43.3 -35.4
# Ground ambient level: 17.7

void light solar
0
0
3 6.73e+06 6.73e+06 6.73e+06

solar source sun
0
0
4 0.421409 -0.593560 0.685639 0.5

void brightfunc skyfunc
2 skybr skybright.cal
0
7 1 1.10e+01 2.12e+01 5.45e-01 0.421409 -0.593560 0.685639
```

I don't want to explain exactly how this works but the important information from this file is the direction where the sun is. There is a vector $v = (0.421409, -0.593560, 0.685639)^\top$ used twice in this file. This is the direction vector to the sun. Instead of using a very distant light with a very high color value representing the energy I decided to calculate a matrix which will position the light source for the sun 100 units away from the middle of the window³⁴ along the vector mentioned above and find some values for the light energy which do look good:

```
light "Sun"
    "physical_light" (
        "color"      1e7 1e7 1e7
    )
    origin 0 0 0
end light

instance "Sun_inst" "Sun"
    transform
        1.0 0.0 0.0 0.0
        0.0 1.0 0.0 0.0
        0.0 0.0 1.0 0.0
        -45.1409 58.356 -69.5639 1.0
end instance
```

³⁴The position is $(3, 1, 1)^\top$.

The reason why I didn't use a distant light source is that in mental ray a distant light source has no origin. But how can I specify that the light source representing the sun is outside the room and does not illuminate other objects directly beside that little portion of the ground and one of the walls? I think 100 units is far away enough to make the light rays which are emitted from a point light and go through the window nearly parallel and the values for the light color a purely found by "try and error".

The second aspect is the visibility of the window itself. Because the window is never seen directly in the final image I decided to represent the "window" just by the absence of geometry at the hole in the back wall. This hole means that the reflected rays will leave the scene without hitting any geometry. In this case an environment shader is called by mental ray if the MI file representing the scene defines one. I first tried to use a simple phenomenon:

```
declare phenomenon
  color "env_sh" (
  )
  shader "tex" "mib_opacity" (
    "input" 9.0 9.0 10.0,
    "opacity" 1.0 1.0 1.0
  )
  root = "tex"
end declare

camera "Camera"
  frame          1
  output         "rgb" "room.rgb"
  focal          35.0
  aperture       32.0
  aspect         1.0
  resolution     510 510
  environment "env_sh" ()
end camera
```

This phenomenon creates an unwanted artefact. The rim of the sphere gets the same slightly blue color as the reflections of the window. Why is that? Well, at the rim of the sphere the refracted and reflected rays hitting the front and back of the sphere might bounce back and forth. This effect can happen a lot with material similar to glass especially if it's attached to geometry with a lot of curvature or local displacements. The solution to this problem is to write a very simple environment shader:

```
/* myEnvironment.c */

#include <shader.h>

struct myEnvironment
{
  miColor color;
```

```

};

DLLEXPORT miBoolean
myEnvironment (
    miColor* result,
    miState* state,
    struct myEnvironment* paras
)
{
    miColor* color;

    /* get parameters */
    color = mi_eval_color(&paras->color);
    /* result */
    if (state->reflection_level >= (state->options->reflection_depth - 1) ||
        state->refraction_level >= (state->options->refraction_depth - 1) ||
        (state->reflection_level + state->refraction_level) >=
        (state->options->trace_depth - 1))
    {
        result->r = 0.0;
        result->g = 0.0;
        result->b = 0.0;
    }
    else
    {
        *result = *color;
    }

    return(miTRUE);
}

```

Basically environment shaders get called for rays that leave the scene entirely, and for rays that would exceed the trace depth limit. If mental ray reaches the maximum number of reflection or refraction rays or a combination of them my simple environment shader will return “black” instead of the color which was given to the shader as a parameter.

Now you can change the environment shader attached to the camera to render the same image as shown in figure 18:

```

...
link "myEnvironment.so"
#include <myEnvironment.mi>
...
camera "Camera"
...
    environment "myEnvironment" ("color" 9.0 9.0 10.0)
end camera
...

```

The high values for the color are not motivated by proper physics. I used “try and error” again to find values which do look good but be aware that if you would rotate the camera and look straight at the “window” the result would be very bright. In Radiance however you could do that because the `illum` material takes care of that.

Another thing which is missing in the final image rendered with mental ray are the shadows cast by light which was emitted from the sun, scattered around by hitting the ground, buildings or plants, and finally contributed to the illumination of the room. I thought about adding another area light at the window position which is not visible but does

emit a few rays to create a soft shadow on the wall behind the glass sphere. But I leave that as an exercise.

4. Finally, I didn't add caustics to the scene. I made some experiments with that but the effect didn't contribute that much to the quality of the picture. So I dropped it. Radiance didn't calculate caustics either. Beside that the rendering time can be quite high for caustics and the color values for the `physical_light` shader should match the energy used to emit photons. See section 4.4 for a better example with caustics.

4.4 Caustics

Caustics are light patterns that are created when light from a light source illuminates a diffuse surface via one or more specular reflections or transmissions.

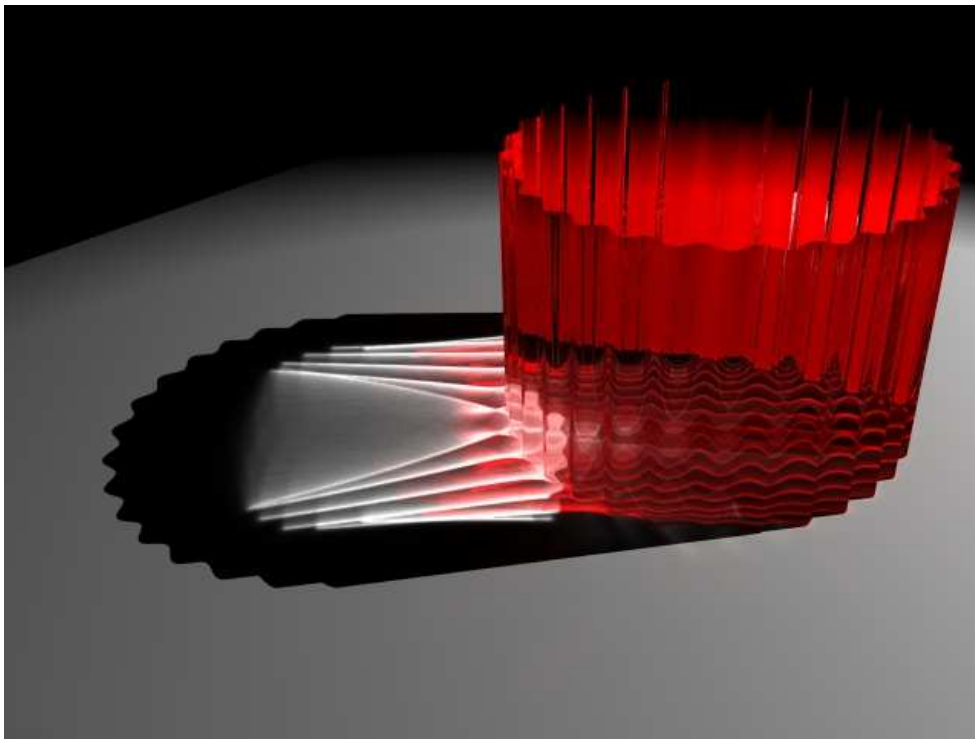


Figure 19: Caustics rendered with mental ray

In figure 19 you see a very simple geometry, a cylinder, being modified by a displacement shader to create an interesting pattern of caustics on the floor.

The displacement shader for RenderMan is quite simple:

```
/* myDisplacement */

displacement
myDisplacement(float factor = 50)
{
    float amp = 0.01 * sin(s*factor*PI);
    P += amp * normalize(N);
    N = calculatenormal(P);
}
```

Because ray tracing, global illumination, and photon mapping are relative new features of Pixar's RenderMan implementation PRMan there is no general documentation how to add the new features into the the standard interface specified by the *The RenderMan Interface*. Other renderers like BMRT had GI implemented for a long time and they added their own extensions to the standard interface to deal with the new features. Therefore I can't show how to create caustics with all the RenderMan compatible renderers which do implement their own method. I made a test with PRMan 11.0 which can be found in the documentation of the **RenderMan Artist Tools** (RAT) but this was a very basic test that caustics can be rendered with PRMan and for the more advanced test I used AIR. However **PRMan** has a very nice program called `ptviewer` which can be used to interactively look at a photon map to rotate it and zoom in and out. The basic approach is to render first a pass to create a photon map and store it before you render the final image in the second pass.

So the main difference between the RIB file for the first pass and the RIB file used for the second pass is:

1. The first pass creates a photon map file by using a special `Hider`:

```
Hider "photon" "emit" 1000000
```

This line defines how many photons are emitted into the scene. The following line defines that the photons are used for caustics and the filename where to store it:

```
Attribute "photon" "causticmap" "prman11caustics.cpm"
```

The same line is used in the second pass to read the photon map.

Instead of using a surface shader in the first pass an attribute is defined for the ground plane which receives the photons:

```
Attribute "photon" "shadingmodel" "matte"
```


For the geometry refracting or in this case reflecting the photons you define a similar attribute:

```
Attribute "photon" "shadingmodel" "chrome"
```

Unfortunately this “shading models” don’t give you the freedom to use real shaders which specify how much photons are reflected or refracted, how many are absorbed or which additional parameters would influence the direction where the photons go.

2. The second pass is using the file where the photons were stored during the first pass and creates an image. Therefore you have to define a camera and all the other settings which influence the image generation. The first real difference between the two RIB files beside the camera and rendering settings is:

```
LightSource "causticlight" 2
```

This defines an additional light source which is used to add the effect the photons have on the caustics. Beside of defining the filename for the photon map a few additional commands are used to make the objects visible to rays, for shadow creation — basically ray tracing is activated with that — and to define a number which is used for collecting the photons nearby.

```
Attribute "visibility" "trace" 1 # make objects visible to rays
Attribute "visibility" "transmission" "opaque" # for shadows
Attribute "photon" "causticmap" "prman11caustics.cpm"
Attribute "photon" "estimator" 200
```

The ground plane is now using a real surface shader called `matte` which has a diffuse component:

```
Surface "matte"
```

The geometry reflecting the photons gets a simple reflecting surface shader called `simplemirror` which is taken from the documentation coming with the **RenderMan Artist Tools (RAT)**:

```
surface simplemirror ( )
{
    normal Nn = normalize(N);
    vector In = normalize(I);
    color reflection = Cs;

    if (Nn.In < 0) {
        vector reflDir = reflect(In,Nn);
        reflection = trace(P, reflDir);
    }

    Ci = Os * reflection;
    Oi = Os;
}
```

This shader uses the `trace` function to shoot a ray into the `reflect` direction and therefore ray tracing has to be activated.

As I said before I did use AIR for the other tests and created a very similar scene to the one coming with the RAT documentation:

```
FrameBegin 1
  Format 400 300 1
  PixelSamples 4 4
  ShadingInterpolation "smooth"
  Display "aircaustics.tiff" "file" "rgba"
  Projection "perspective" "fov" 22
  Translate 0 -0.5 8
  Rotate -40 1 0 0
  Rotate -20 0 1 0

  Option "render" "max_raylevel" [4]
  Option "model" "float unitsize" [0.001]
  Attribute "trace" "bias" [0.05]

WorldBegin
  LightSource "caustic" 1
  Attribute "light" "string shadows" "on"
  Attribute "light" "integer nphotons" [ 100000 ]
  LightSource "spotlight" 2 "from" [-4 7 -7] "to" [0 0 0]
  "intensity" 100 "coneangle" 0.2

  Attribute "render" "string casts_shadows" ["opaque"]
  Attribute "visibility" "integer shadow" [1]

# Ground plane
AttributeBegin
  Attribute "caustic" "float maxpixeldist" [ 20 ]
  Attribute "caustic" "integer ngather" [ 75 ]
  Attribute "visibility" "integer camera" [ 1 ]
  Attribute "visibility" "integer reflection" [ 1 ]
  Attribute "visibility" "integer shadow" [ 1 ]
  Attribute "visibility" "indirect" [ 1 ]
  Surface "matte"
  Color [1 1 1]
  Scale 3 3 3
  Polygon "P" [ -1 0 1 1 0 1 1 0 -1 -1 0 -1 ]
AttributeEnd

# Box
AttributeBegin
  Attribute "caustic" "color specularcolor" [ 0.4 0.4 0.4 ]
  Attribute "caustic" "color refractioncolor" [ 0.0 0.0 0.0 ]
  Attribute "caustic" "float refractionindex" [ 1.0 ]
  Color [1 1 0]
  Attribute "visibility" "integer camera" [ 1 ]
  Attribute "visibility" "integer reflection" [ 1 ]
  Attribute "visibility" "integer shadow" [ 1 ]
  Attribute "visibility" "indirect" [ 1 ]
  Translate 0.3 0 0
  Rotate -30 0 1 0
  Surface "simplemirror"
  Polygon "P" [ 0 0 0 0 0 1 0 1 1 0 1 0 ] # left side
  Polygon "P" [ 1 0 0 1 0 1 1 1 1 1 0 0 ] # right side
  Polygon "P" [ 0 1 0 1 1 0 1 0 0 0 0 0 ] # front side
  Polygon "P" [ 0 1 1 1 1 1 1 0 1 0 0 1 ] # back side
  Polygon "P" [ 0 1 0 1 1 0 1 1 1 0 1 1 ] # top
AttributeEnd
WorldEnd
FrameEnd
```

The main difference between AIR and PRMan is that you can create the caustics in a **single pass**. Here in short the steps necessary to produce caustics with AIR. This is taken from the documentation coming with AIR:

1. Add a caustic light source to the scene:

```
LightSource "caustic" 1
```

2. Set the number of photons to trace for each light that is to produce caustics:

```
Attribute "light" "integer nphotons" [ 100000 ]
```

3. For each surface that is to receive caustics, set the minimum number of photons to gather for the irradiance estimate:

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

and the maximum distance (in pixels) for gathering photons:

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

Larger values produce smoother results and require that fewer photons be used overall.

4. Primitives that are to receive or produce caustics must be made visible to indirect/photon rays with:

```
Attribute "visibility" "indirect" [ 1 ]
```

5. For each primitive that is to produce caustics by reflecting or refracting light, set the reflective color, refractive color, and refraction index for caustics:

```
Attribute "caustic" "color specularcolor" [ 0.4 0.4 0.4 ]
Attribute "caustic" "color refractioncolor" [ 0.0 0.0 0.0 ]
Attribute "caustic" "float refractionindex" [ 1.0 ]
```

The specularcolor and refractioncolor attributes determine what happens to photons that intersect a surface. Photons will be reflected, refracted or absorbed in proportion to (respectively), the average intensity of specularcolor, the average intensity of refractioncolor, and 1 minus the sum of the average intensities. The sum of specularcolor and refractioncolor should be less than or equal to 1. If the sum is greater than or equal to 1, no photons will be stored on that surface and no caustics will be visible.

In AIR photons can also be saved to a file in the first pass and reused in a second pass. It's in your responsibility to make sure that no photons are emitted in the second pass.

I managed to render a very similar picture to figure 19 with AIR. Nevertheless I have the feeling that mental ray's control over how many photons are considered within a certain radius is more powerful if it comes to creating

fine details in the caustics pattern. AIR is measuring a similar distance in pixels which makes it harder to work on details which might be less than a pixel.

Before we talk about how to create caustics with mental ray I will give you the displacement shader for it:

```
/* myDisplacement.c */

#include <shader.h>

struct myDisplacement
{
    miScalar factor;
};

DLLEXPORT miBoolean
myDisplacement(
    miScalar* result,
    miState* state,
    struct myDisplacement* paras
)
{
    miScalar factor;

    /* get parameters */
    factor = *mi_eval_scalar(&paras->factor);
    /* result */
    *result += (0.01 * sin(state->tex_list->x * factor * M_PI));

    return(miTRUE);
}
```

It shouldn't be a problem to modify the Makefile I have shown earlier to compile this shader. To create the corresponding MI file where the shader and its parameters are declared should be easy as well. Let's focus on the relevant parts of the MI file which was used to render figure 19:

1. Caustics must be enabled in the options block.

```
options "#opt"
...
    caustic          on
    caustic accuracy 700 .05
    caustic filter cone 1.1
    photonmap file "mrphotonmap.cpm"
    photon trace depth      4 4
...
end options
```

Further options give finer control over the process and help to fine-tune the result. By writing the photon map to a file it is possible to create once a photon map with a lot of photons and change later only parameters which do not effect the process of storing the photons. This

process takes a while and will be done before the actual ray tracing of the scene is done. Because of its nature of being a post-process it's worth to make first a decision about the number of photons to emit and the energy used in the lights by simple tests without fine-tuning. Once that decision is made you can reuse the photon map and play with the parameters like `caustic accuracy`.

2. The light source must have an energy statement:

```
light "#/obj/light1_obj"
    "physical_light" (
        "color" 1000.0 1000.0 1000.0,
        "cone" 0.9867880932509171
    )
    origin      0 0 0
    direction   0 0 -1
    spread      0.96891242171064473
    energy      1000.0 1000.0 1000.0
    caustic photons 5000000
end light
```

It is advisable to use a light shader like `physical_light` that ensures physical correctness. The energy is distance-dependent and often has to be chosen quite large.

3. Both the caustic-casting and caustic-receiving objects must have a material that contains a photon shader statement:

```
material "#mtl0"
    "dgs_material" (
        "diffuse" 0.8 0.8 0.8,
        "lights" [ "/obj/light1" ]
    )
    photon "dgs_material_photon" ()
end material
...
material "#mtl1"
    "dielectric_material" (
        "ior"      0.8,
        "col"      1.0 0.0 0.0,
        "phong_coef" 0
    )
    displace "myDisplacement" ("factor" 60)
    photon "dielectric_material_photon" ()
end material
```

Photon shaders may store and either absorb, reflect, or transmit photons.

4. The material should **not** have a shadow shader to avoid having light pass through the object twice, once directly and once indirectly.
5. The caustic-casting material should not be diffuse because diffuse objects spread the light rather than focusing it.

6. The caustics–receiving material **must** be diffuse, because otherwise no photons are stored there. Its material shader must support collecting stored photons.
7. If the caustics–casting object is refractive the index of refraction should be greater than 1.0 to create a focusing effect.

To generate caustics more efficiently, objects can be flagged such that the photons are only emitted toward certain objects and stored only on selected objects. Objects are then divided into caustic–casting (caustic 1 flag) and caustic–receiving (caustic 2 flag), or both (caustic 3 flag), or neither (caustic 0 flag).

4.5 Baking

Larry Gritz described at SIGGRAPH 2002 “A Recipe for Texture Baking” in course 16 called “RenderMan in Production”. In the same course Hayden Landis, working for Industrial Light + Magic, described several “occlusion” passes in “Production–Ready Global Illumination”. At that time PRMan didn’t support global illumination (GI) and Landis basically showed how to fake certain effects of GI.

Today a lot of renderers do support GI but it might be still worth to invest some time to think about efficiency. The problem with GI is that most of the time you will end up with flickering in an animation because the solution might be slightly different from frame to frame.

The image in figure 20 shows the Chrysler Building rendered with a renderer called **Lucille**³⁵. The renderer is in very early development but it allows you to render RIB files using geometry with only triangle polygons. Which means that you can’t easily render RIB files from production yet but on the other side the examples coming with Lucille can be modified to render with other RenderMan compatible renderers and we can use them to test the techniques described at the SIGGRAPH course and compare rendering times, quality, etc. with the results from Lucille.

Even if the baking process takes quite a long time it might be worth to do it if you can reuse the baked information for a whole sequence. Larry Gritz’ arguments why you might want to bake complex computations into texture maps were:

³⁵See <http://lucille.sourceforge.net>

- Baking can speed up render-time shader execution by replacing a section of expensive view-independent and lighting-independent computations with a small number of `texture()` calls. For patterns that are truly dependent on only parametric (or reference-space) position, and have no time-dependence at all, a single baking can be reused over many shots. This means that it's okay for the original computations to be absurdly expensive, because that expense will be thinly spread across all frames, all shots, for an entire production.
- RenderMan-compatible renderers do an excellent job of automatically antialiasing texture lookups. High-res baking can therefore effectively antialias patterns that have resisted all the usual attempts to analytically or phenomenologically antialias.
- Modern graphics hardware can do filtered texture map lookups of huge numbers of polygons in real time. Although the programmability of GPUs is rapidly increasing, they are still years away from offering the full flexibility offered by the RenderMan Shading Language. But by pre-baking complex procedural shaders with an offline renderer, those shaders can be reduced to texture maps that can be drawn in real-time environments. This revitalizes renderers such as Entropy or PRMan as powerful tools for game and other realtime graphics content development, even though those renderers aren't used in the realtime app itself. As a case in point, consider that you could use Entropy (or another renderer with GI capabilities) to bake out the contribution of indirect illumination into light maps. Those light maps can be used to speed up GI appearance in offline rendering, or to give a GI lighting look to realtime-rendered environments.

The process described in the SIGGRAPH paper was quite technical and showed how to implement the baking by writing “DSO Shadeops”³⁶. The main reason for that is that in the shading language of RenderMan you can't open files. In mental ray this would be easy because the shading language is using C or C++ anyway. I use this technique quite often to dump any information needed later in a shader to a binary file, read the information in the `init` phase of the shader, attach it to the state as `user` data, and reuse the information in the main shader.

In the meantime a lot of renderers support baking natively. For example Si-TeX Graphics offers a special version of their renderer AIR called **BakeAIR** for especially the purpose of baking and **mental ray** 3.0 supports “**light mapping**” for similar reasons.

³⁶DSO stands for Dynamic Shared Objects

Before I show a few examples how to bake information with BakeAIR I want to go back to the example I mentioned earlier. In the part called “Production-Ready Global Illumination” of the SIGGRAPH course the author talks about both HDR³⁷ images and ambient occlusion.



Figure 20: Chrysler Building rendered with Lucille

I just took the example coming with **Lucille** and changed the RIB file a bit to use it with a renderer called **3Delight**³⁸. With 3Delight there comes an example how to use HDR images and I just added the commands necessary to use a light probe as an environment map and to add some ambient occlusion to the result. It makes a big difference to the picture you would get with just the `defaultsurface` shader. And it's rendered with just **one pass**.

The steps necessary are quite simple. First I removed all `Surface` lines, the `AreaLightSource` and all lines the renderer was complaining about from the original RIB file. This allows me to render without any lights using the `defaultsurface` shader. Then I added the following lines for 3Delight:

³⁷High Dynamic Range

³⁸See <http://www.3delight.com>

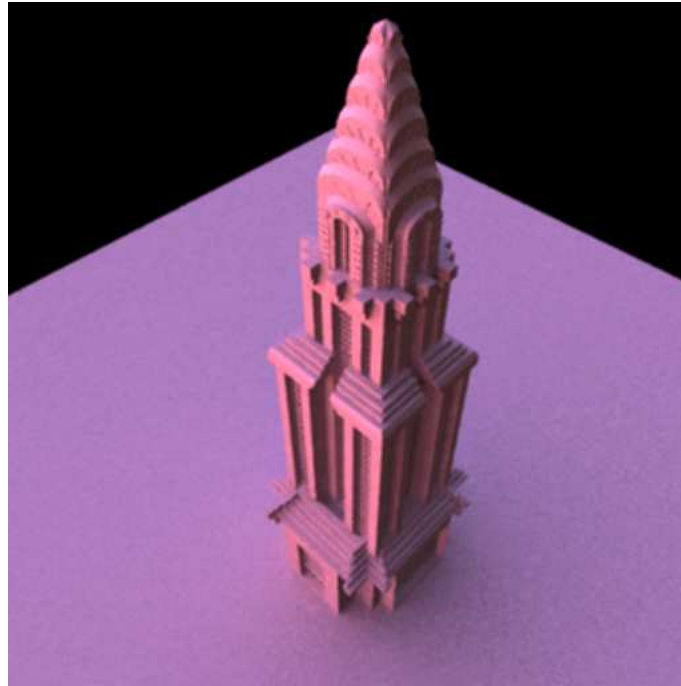


Figure 21: Chrysler Building rendered with 3Delight

```
Attribute "visibility" "transmission" "opaque"
LightSource "envlight" 3 "float samples" 32
    "string envmap" "grace_probe.tdl"
    "float Kenv" .5 "float Kocc" 0.5
Surface "matte" "float Kd" 0.5
```

Objects contributing to occlusion have to be tagged as visible to transmission rays. This is done by the first line. The ambient occlusion is calculated by the light shader `envlight`. Have a look at the implementation of the shader. The same shader takes the HDR image, which was downloaded from <http://www.debevec.org/Probes> and converted to a texture, as an environment map to light the scene purely based on that image. **AIR** has a very similar shader with the same name but different parameters. I didn't try the HDR image with AIR but I checked the ambient occlusion with the following line³⁹:

```
LightSource "envlight" 3 "float occlusionsamples" 32
```

Unfortunately we can't use this example easily to demonstrate baking with BakeAIR because polygons and subdivision surfaces lack a natural coordinate system for 2D textures. If the model would have been made of NURBS

³⁹All other lines are the same as in the RIB file for 3Delight.

it would be easier to bake the ambient occlusion. Nevertheless ambient occlusion can be used with polygons without baking it and there are examples coming with BakeAIR which demonstrate how to bake with NURBS surfaces. An interesting example how to bake displacements can be found online at the following two addresses:

http://www.drone.org/tutorials/displacement_maps_air.html
<http://www.seanomlor.com/owenr>

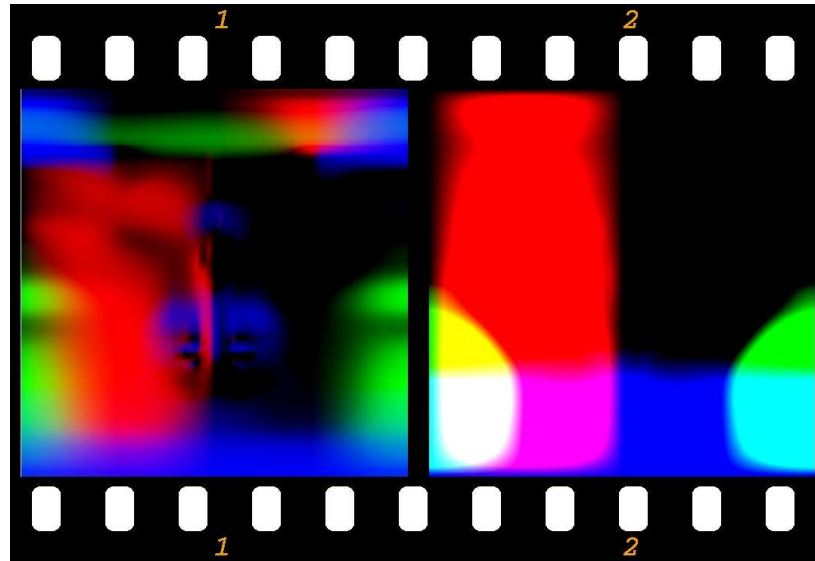


Figure 22: Baked normals and positions of a head

The idea is to use a low-res version of a subdivision surface and measure the distance to a high-res version. Once that information is baked you can render the low-res version and use the baked information to displace it back to the original high-res version. But why would you like to do that? Well, this way you can use arbitrary shapes and **morph** them into each other by animating the displacement over time.

Let's have a look at one of the examples where you bake the normals and positions of a face and use them later to place hairs over the whole face. Figure 22 shows the resulting image for baking the normals on the left, and for baking the positions on the right. First you have to define for each texture you want to create through the baking process a `Display` line:

```
Display "headP.tif" "file" "P" "quantize" [0 0 0 0]
Display "headN.tif" "file" "N" "quantize" [0 0 0 0]
```

The next step is to either specify a camera which will produce a tessellation during the rendering process which will be used for the baking or to explicitly control the tessellation with some additional commands:

```
Attribute "divide" "u" 64
```

Now you specified the tessellation and the filenames for the baking but you didn't say which geometry should be used. This is done by adding the following lines to the geomtry definition:

```
NuPatch ... "P" [ ... ]  
"constant string bakemap_P" "headP.tif"  
"constant string bakemap_N" "headN.tif"
```

Figure 22 shows the baked textures which are used to grow hair on the NURBS surface. Look at figure 23 for the result.

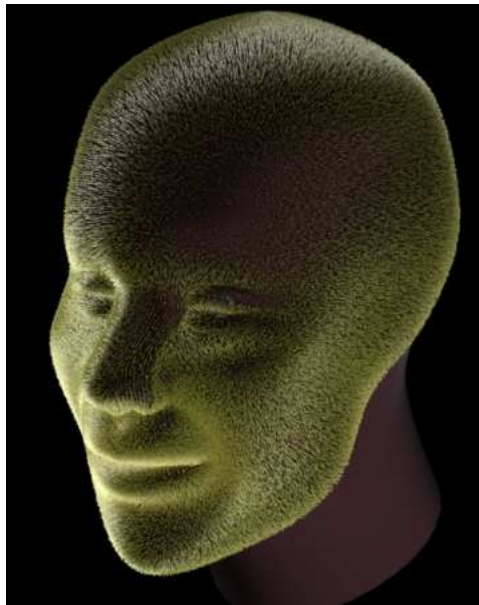


Figure 23: Use a procedural shader to grow hair

But how was the hair placed onto the head? **AIR** allows you to write — beside the five standard shader types mentioned earlier — shaders of two additional shader types: The `transformation` and `procedure` shaders. In the RIB file creating the image of figure 23 you will find at the very end the following lines:

```

...
    Surface "VHair" "color RootColor" [.03 .02 .01]
                      "color TipColor" [.6 .6 .2]
    Procedure "growhair"
        "float ns" 400
        "float nt" 400
        "string Pfile" "headP.tif"
        "string Nfile" "headN.tif"
        "float height" .3
        "float width" .02
        "float vary" .2
    WorldEnd
FrameEnd

```

This attaches a surface shader called `VHair` with some parameters to the geometry which will be created by a procedure shader with the name `growhair`. The surface shader is coming with the AIR distribution and the compiled version of the shader was created from file which can be found in the `$AIRHOME/vshaders` folder. All the file in this folder were created visually by using **VShade** on Windows. See section 1 for a screenshot of this program. Nevertheless the file is stored in a human readable way and shouldn't be too different from standard shaders. The source code for the other shader can be found in `$AIRHOME/bakeexamples/fuzz`. Basically a procedure shader uses `ribprintf` commands to create the lines for the RIB file where the shader is called.

5 Light Source Shaders

In the chapter *Other Kinds of Shader* in book [3] you find a section about light shaders. In book [1] there are two chapters dealing with light shaders: The chapter *Lighting and Shading* gives an overview of light source shaders and the chapter *A Gallery of Shaders* shows the source code of the standard shaders. There you will find the following light source shaders explained:

- **ambientlight**⁴⁰
- **distantlight**⁴¹
- **pointlight**⁴²
- **spotlight**⁴³

⁴⁰See section 5.1.

⁴¹See section 5.2.

⁴²See section 5.3.

⁴³See section 5.4.

In the book [2] there is section of chapter *Illumination Models and Lights* which deals with *Light Sources* and beside the standard light shaders you will find the source code for the following shaders:

- **arealight**⁴⁴
- **shadowspot**⁴⁵

The book [3] will show you some non-standard light shaders:

- **nearfarlight**: Is an example of a non-physical light.
- **beamlight**: Is a simplified version of the standard spotlight shader with sharp edges.
- **colorlight**: Shows how to modify the light color based on the "hsv" color space:

```
...
lightcolor = color "hsv" (cosangle*10, 1, 1);
...
```

Here are all the color coordinate systems supported in the shading language:

Coordinate system	Description
"rgb"	Red, green, blue
"hsv"	Hue, saturation, value
"hsl"	Hue, saturation, lightness
"xyz", "XYZ"	CIE XYZ coordinates
"YIQ"	NTSC coordinates

Table 2: Color spaces

From the RIB file perspective you should know the following facts about lights. The RenderMan interface defines two types of light sources:

- **Non-geometric lights** are light sources for which there is a single “ray” of light from the light source to any given point on the surface and they are created with the RIB command `LightSource`.

⁴⁴See section 5.5.

⁴⁵See section 5.6.

- **Geometric lights** are light sources for which there are multiple rays of light, emanating from independent positions, streaming towards each point on the surface and they are created with the RIB command `AreaLightSource`. The easiest way to describe those positions is to associate some geometric shape with the light source.

You can make further distinctions based on the commands used in the light shader:

- An **ambient light source** is one with no `illuminate` or `solar` statement. See section 5.1 for an example.
- The `illuminate` construct needs a `position` and optional you can specify a `direction axis` and an `angle` to define a “cone of light” with the apex at `position`. See section 5.3 for the use of this construct in the standard `pointlight` shader by using only the `position`. See section 5.4 for the use of this construct with all possible parameters in the standard shader `spotlight`.

```
illuminate(position [,axis,angle]) statement
```

- The `solar` construct restricts illumination to a range of directions without specifying a position for the source. The `direction axis` and the `angle` are optional. See section 5.2 for the use of this statement in the standard shader `distantlight`. In the book [1] you will find an example how to use the `solar` construct for a skylight shader which defines a hemisphere.

```
solar([axis,angle]) statement
```

Table 5 on page 74 shows a list of predefined light source shader variables.

5.1 Ambient Light Sources

If a light source does not have an *illuminate* or *solar* statement, it is a non-directional ambient light source. An ambient light source supplies light of the same color and intensity to all points on all surfaces. The source code for the standard shader `ambientlight` is quite simple:

```
light ambientlight(
    float intensity = 1;
    color lightcolor = 1)
```

```

{
    C1 = intensity * lightcolor;
}

```

The result of the **ambientlight** shader is placed in the global variable C1, which is the output of all light sources.

5.2 Distant Light Sources

Unlike an ambient light source, a distant source casts its light in only one direction:

```

light distantlight (
    float intensity = 1;
    color lightcolor = 1;
    point from = point "shader" (0,0,0);
    point to = point "shader" (0,0,1))
{
    solar (to-from, 0)
    C1 = intensity * lightcolor;
}

```

The vector defined by the difference (*to-from*) is used in a **solar** statement to restrict the emission to that direction.

5.3 Point Light Sources

A point light source is the converse of a distant light. It radiates light in all directions, but from a single location.

```

light pointlight (
    float intensity = 1;
    color lightcolor = 1;
    point from = point "shader" (0,0,0))
{
    illuminate (from)
    C1 = intensity * lightcolor / (L.L);
}

```

Within **illuminate**⁴⁶ L is a vector from the position of the light source to the surface point being illuminated. The dot product implements the square-law falloff of light intensity with distance from the source.

5.4 Spot Light Sources

This shader also uses the **illuminate** construct, but it supplies not just the spotlight position, but also a cone of illumination:

```
light spotlight (
    float intensity = 1;
    color lightcolor = 1;
    point from = point "shader" (0,0,0);
    point to = point "shader" (0,0,1);
    float coneangle = radians(30);
    float conedeltaangle = radians(5);
    float beamdistribution = 2)
{
    float atten, cosangle;
    uniform vector A = normalize(to-from);

    illuminate (from, A, coneangle) {
        cosangle = (L.A) / length(L);
        atten = pow(cosangle, beamdistribution) / (L.L);
        atten *= smoothstep(cos(coneangle),
                           cos(coneangle-conedeltaangle),
                           cosangle);
        Cl = atten * intensity * lightcolor ;
    }
}
```

The latter is specified as a direction, A , and a width, specified in radians by the `coneangle` instance variable.

5.5 Area Light Sources

Area light sources are those that are associated with geometry. Similar to the RIB command `LightSource` there is another RIB command called `AreaLightSource` which specifies the shader name, a handle, and a parameter list.

```
light arealight (
    float intensity = 1;
    color lightcolor = 1)
```

⁴⁶The only place it is defined.


```

{
    illuminate (P, N, PI/2)
        Cl = intensity * lightcolor / (L.L);
}

```

Note the similarity to the **pointlight** shader — the main difference is that, rather than using a `from` parameter as the light position, we illuminate from the position `P` that the renderer chose for us by sampling the area light geometry.

5.6 Light Sources And Shadows

Similar to the **spotlight** shader shown before you can define a **shadowspot** shader which casts shadows. The main difference is that before the final color is assigned to the global variable `Cl` there might be an attenuation of the light contribution because the illuminated point is in shadow. Here are the two lines (beside additional parameters for the shader) which make the difference:

```

...
if (shadowname != "") {
    atten *= 1-shadow(shadowname, Ps,
                      "samples", shadownsamps,
                      "blur", shadowblur,
                      "bias", shadowbias);
}
Cl = atten * intensity * lightcolor;
...

```

Notice that the source code of the shader printed in the book is slightly different from the version you can download from the companion web site:

<http://www.mkp.com/renderman>

6 Surface Shaders

7 Displacement Shaders

8 Imager Shaders

9 Volume Shaders

10 Tables

Name	Type	Description
Cs	color	Surface color
Os	color	Surface opacity
P	point	Surface position
dPdu	vector	Derivative of surface position along u
dPdv	vector	Derivative of surface position along v
N	normal	Surface shading normal
Ng	normal	Surface geometric normal
u,v	float	Surface parameters
du,dv	float	Change in surface parameters
s,t	float	Surface texture coordinates
L	vector	Incoming light ray direction
Cl	color	Incoming light ray color
OI	color	Incoming light ray opacity
E	point	Position of the eye
I	vector	Incident ray direction
ncomps	float	Number of color components
time	float	Current shutter time
dtime	float	The amount of time covered by this shading sample
dPmtime	vector	How the surface position P is changing per unit time, as described by motion blur in the scene
CI	color	Incident ray color
OI	color	Incident ray opacity

Table 3: Predefined surface shader variables

Name	Type	Description
P	point	Pixel raster position
Ci	color	Pixel color
Oi	color	Pixel opacity
alpha	float	Fractional pixel coverage
ncomps	float	Number of color components
time	float	Shutter open time
dtime	float	The amount of time the shutter was open
Ci	color	Output pixel color
Oi	color	Output pixel opacity

Table 4: Predefined imager shader variables

Name	Description
P	Surface position on the light
dPdu	Derivative of surface position along u
dPdv	Derivative of surface position along v
N	Surface shading normal on the light
Ng	Surface geometric normal on the light
u,v	Surface parameters
du,dv	Change in surface parameters
s,t	Surface texture coordinates
L	Outgoing light ray direction
Ps	Position being illuminated
E	Position of the eye
ncomps	Number of color components
time	Current shutter time
dtime	The amount of time covered by this shading sample
Cl	Outgoing light ray color
Ol	Outgoing light ray opacity

Table 5: Predefined light source shader variables

References

- [1] Steve Upstill. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison Wesley, 1990.
- [2] Anthony A. Apodaca; Larry Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann, 1999.
- [3] Ian Stephenson. *Essential RenderMan fast*. Essential series. Springer-Verlag, 2003.
- [4] Thomas Driemeyer. *Rendering With Mental Ray*, volume 1 of *mental ray Handbooks*. Springer-Verlag, 2000.
- [5] Thomas Driemeyer. *Programming Mental Ray*, volume 2 of *mental ray Handbooks*. Springer-Verlag, 2000.