RenderMan® for Poets

version 2.0 - 27 February 1994

Larry Gritz

Abstract

The RenderMan® Interface Standard was created by Pixar to provide a standard for detailed scene specification for communication between 3D modeling programs and rendering programs. It is superficially similar to PostScript, but designed as a scene description format rather than a page description language. The Computer Graphics Research Group at GWU uses the RenderMan standard for much of its work, utilizing a number of software tools written by graduate students which adhere to the RenderMan standard. This document explains how to use the RenderMan Interface and to write simple RIB files. It is intended for the reader who is familiar with the concepts of computer graphics but has little experience using RenderMan. It is designed to give the reader just enough knowledge to create simple RIB files, but not to explain the advanced features of RenderMan.

Table of Contents

1. INTRODUCTION

This document explains how to use the RenderMan Interface and to write simple RIB (RenderMan Interface ByteStream) files. It is intended for the reader who is familiar with the concepts of computer graphics but has little experience using RenderMan. It is designed to give the reader just enough knowledge to create simple RIB files, but not to explain the advanced features of RenderMan. For more detailed information about the RenderMan standard, you should see *The RenderMan Companion*, by Steve Upstill, or the official *RenderMan Interface Specification*, available from Pixar. Both of these texts are fully detailed and clearly written, and no attempt will be made here to duplicate all but the most basic information in these references.

1.1 What is RenderMan?

RenderMan is a standard, created by Pixar, through which modeling programs can talk to rendering programs or devices. It may be thought of as a scene description format in the same way that PostScript is a page description format. This standard is hardware and operating system independent. RenderMan allows a modeling program to specify *what* to render, but not *how* to render it. A renderer which implements the RenderMan standard may choose to use scanline methods, ray tracing, radiosity, or any other method. These implementation details have no bearing on the writing of RIB.

1.2 Procedural vs. RIB Interface

The first version of the RenderMan standard described a procedural interface, i.e. the function calls for a library which could be linked to a modeling program. When the procedures are invoked, information is passed to the renderer. Later versions of the RenderMan interface also defined the RenderMan Interface Bytestream, or RIB protocol. RIB provides an ASCII interface to a renderer which supports the RIB protocol.

A modeling or motion control program may create RIB in two ways:

1. Make procedural interface calls which result in the output of RIB.

2. Output RIB directly (e.g., using printf).

In the remainder of this document, RIB and procedural interface calls will be intermixed. Since there is a nearly one-to-one correspondence between the two, it should be fairly straightforward for the reader to follow the discussion.

1.3 RIB Conventions

RIB files are ASCII files organized into RIB *requests*. The following information may be helpful in creating useful RIB files:

1. Each RIB request is on its own line, and subsequent requests are separated by carriage returns.

2. Any line beginning with a single hash character (#) indicates that the line is a comment and will be ignored by the RIB interpreter.

3. Any line beginning with two hash characters (##) indicates a *renderer hint*. The most useful hint is the `##Include` directive. When followed by a file name in double quotes, this causes the file to be read into the input stream at this point, just like the `#include` macro works in the C language. For example:

```
...
# This line is a comment
Translate 10 4 4
##Include "car.rib"
...
```

1.4 Procedural Conventions

The procedural interface is a linkable library with function call bindings for the C or C++ languages. The following information may be helpful in using the procedural interface:

1. All of the procedural calls are functions beginning with the prefix "Ri". Function prototypes and type definitions may be found in the header file `ri.h`.

2. Use of the RenderMan library should begin with a call to RiBegin, and end with a call to RiEnd:

```
...
RiBegin (RI_NULL);
...
RiEnd ();
...
```

No RenderMan calls should be made prior to `RiBegin` or after `RiEnd`.

3. The following useful type definitions are given in `ri.h`:

```
typedef char *RtToken;
typedef float RtFloat;
typedef RtFloat RtPoint[3], RtColor[3];
typedef RtFloat RtMatrix[4][4];
```

4. Many of the calls take optional arguments. The optional arguments are arranged into "token/value" pairs. The token is a quoted string (char *), and the value is a (void *) to a value appropriate to the token name. The list is terminated with the reserved token `RI_NULL`. For example, the declaration for the RiSurface procedure is as follows:

```
void RiSurface (RtToken name, ...);
```

This indicates that the function takes one mandatory argument, a string (RtToken), and a number of optional parameters. The example below shows how two token/value pairs are passed to the RiSurface procedure:

```
RiSurface ("matte", "Kd", &kd, "Ks", &ks, RI_NULL);
```

1.5 Acknowledgments

I am required to state the following:

Pixar's *RenderMan Interface Specification* document stipulates that you may write modeling programs that incorporate the RenderMan procedure calls or that output RIB, royalty-free, as long as you include the copyright notice above. Their document also allows for a no-charge license for anyone who wishes to write a renderer that executes the Pixar RenderMan procedure calls or RIB requests.

2. RIB File Structure

2.1 Introduction

Almost all RenderMan requests fall into the following three categories:

1. Modification of the graphics state options.
2. Modification of the graphics state attributes.
3. Declaration of geometry.

As geometry is declared, each object will inherit the "current" graphics state attributes. Modifications to the graphics state attributes will affect any subsequently defined geometry, but not geometry which has already been declared.

The graphics state may be saved and restored using `AttributeBegin` and `AttributeEnd`.

Most geometric objects have preferred coordinate systems. A sphere, for example, is always declared with its center at the origin. If you want the sphere someplace else, you have to give the appropriate transformation first, then declare the geometry. The default coordinate system (before any transformation are applied) is with the origin at the imaging plane, x axis to the right, y axis up, and z axis "into" the screen. Note that this is a left handed coordinate system.

A correct RIB file contains the following features:

1. Setting options that are constant for all frames. Examples include image resolution and pixel samples.
2. `FrameBegin` statement.
3. Setting graphics state options for this frame, such as output filename.
4. Setting graphics state options and attributes which are constant for this frame, such as the camera transformation and projection type, depth of field, and light source declarations.
5. `WorldBegin` statement.
6. Intermixed graphics state attribute modifications and geometry declarations for the current frame.
7. `WorldEnd` statement. This causes the following actions: The current frame is rendered and saved. Any geometry and lighting declared in (6) is destroyed. The graphics state is returned to the way it was right before (5).
8. `FrameEnd` statement. This causes the graphics state to return to its condition immediately before (2).
9. If more frames are to be specified, steps 2-8 may repeat.

2.2 Control Requests

RiBegin (RtToken name, ...)

When using the procedural interface, a call to RiBegin must precede any calls to other Ri routines. Generally, only the reserved token `RI_NULL` should be passed to RiBegin. There is no corresponding RIB request.

RiEnd (void)

This statement should be the last Ri call made by a program. It serves to clean up the mess caused by the renderer, free memory, etc. There is no corresponding RIB request (the equivalent operations are performed when the end of the RIB file is reached).

RiArchiveRecord (RtToken type, RtString format, ...)

When the procedural calls cause RIB to be output (as is the case with the `libribout.a` library), this call causes information to be put into the RIB stream. The value of *type* may be either `"comment"` or `"structure"`. In the case of `"comment"`, the call results in the specified information being place into the RIB file as a comment. In the case of `"structure"`, the specified information is placed into the RIB file as a "renderer hint." In either case, the format and optional arguments work just like the C 'printf' function. Here are two examples:

```
RiArchiveRecord ("comment", "This will be a comment", RI_NULL);
RiArchiveRecord ("structure", "Include \"myfile.rib\"", RI_NULL);
RiArchiveRecord ("structure", "Include \"%s\"", filename, RI_NULL);
```

FrameBegin framenum
RiFrameBegin (int framenum)

Denotes the beginning of frame *framenum* for a multi-frame RIB file, which results in pushing the current options onto the stack. Options may be changed outside of the FrameBegin/FrameEnd block, but all geometry and attribute declarations (including the WorldBegin/WorldEnd block) *must* occur inside the frame block.

FrameEnd
RiFrameEnd ()

Denotes the end of a frame for a multi-frame RIB file, resulting in the restoration of the graphics state to that which was in effect at the corresponding `FrameBegin` statement.

WorldBegin
RiWorldBegin ()

Saves the current options and uses the current coordinate transformation to denote "world space." All declarations of geometry or attribute changes must occur between `WorldBegin` and

`WorldEnd`, but options may be changed outside of this block. Coordinate transformations before `WorldBegin` denote movements of the camera, while transformations inside the World block apply to individual geometric primitives.

WorldEnd
RiWorldEnd ()

> Denotes the end of the world block, generally resulting in rendering of the scene.

2.3 Modifying Graphics State Options

***Display* name *type* mode**
RiDisplay (char *name, RtToken type, RtToken mode, ...)

> *name*, enclosed in double quotes, is the name of the file to which the current image should be saved (if *type* is `"file"`), or the name of the framebuffer in which to display the image (if *type* is `"framebuffer"`).

> *mode* is a string, enclosed in double quotes, which gives the file type of the image. *mode* may be any combination of `"rgb"`, `"a"`, and `"z"`.

> For example:

> | | |
> |---|---|
> | `"rgba"` | Renders an RGB image plus an alpha channel (the default). |
> | `"rgb"` | Renders an RGB three-channel image. |
> | `"z"` | Renders a depth-only image. |

Examples:

```
Display "myfile.tif" "file" "rgba"

RiDisplay ("myfile.tif", "file", "rgba", RI_NULL);
```


***Format* xres yres pixelaspectratio**
RiFormat (RtFloat xres, RtFloat yres, RtFloat pixelaspectratio)

> Set the total image resolution to *xres* columns by *yres* rows. *pixelaspectratio* is the width of a screen pixel divided by the height of a screen pixel. If screen pixels are square, *pixelaspectratio* should be 1. According to the RenderMan standard, the default values (if no `Format` statement is found) are 640 x 480 resolution and 1.0 aspect ratio. *xres*, *yres* and *pixelaspectratio* are all numerical arguments.

```
Format 640 480 1

RiFormat (640, 480, 1);
```

PixelSamples x y
RiPixelSamples (RtFloat x, RtFloat y)

> Set the degree of image supersampling. For one sample per pixel, both *x* and *y* should be 1. If both *x* and *y* are 2, the image will be computed with a total of 4 samples per pixel. Note that more samples per pixel will reduce aliasing, but will increase rendering time linearly with the total number of samples. Both *x* and *y* are numerical arguments.

```
            PixelSamples 1 1

            RiPixelSamples (1,1);
```

Projection type parameters
RiProjection (RtToken type, ...)

> Set the camera projection type. *type* may be `"perspective"` or `"orthogonal"`. If *type* is `"perspective"`, the following parameter may be used:

```
       "fov"     fov
```

> where *fov* is a numerical argument for the field of view, in degrees.

> Example:

```
            Projection "perspective" "fov" 45

            RiProjection ("perspective", "fov", &fov, RI_NULL);
```

2.4 Modifying Graphics State Attributes

TransformBegin
TransformEnd
RiTransformBegin()
RiTransformEnd()

> `TransformBegin` pushes the current transformation onto the stack so that it may be restored later with `TransformEnd`.

AttributeBegin
AttributeEnd
RiAttributeBegin()
RiAttributeEnd()

> `AttributeBegin` pushes the entire graphics state (including the current transformation) onto the stack so that it may be restored later with by `AttributeEnd`.

Translate x y z
RiTranslate (RtFloat x, RtFloat y, RtFloat z)

> Modify the current transformation by appending a translation.

```
Translate 5 0 23
RiTranslate (5.0, 0.0, 23.0);
```

Rotate **theta x y z**
RiRotate (RtFloat theta, RtFloat x, RtFloat y, RtFloat z)

Modify the current transformation by appending a rotation of *theta* degrees about an axis defined by (*x*, *y*, *z*). Note that rotation is governed by the left hand rule, unless the `Orientation` statement has previously been used.

```
Rotate 45.0  1 0 0
RiRotate (45.0, 1, 0, 0);
```

Scale **sx sy sz**
RiScale (RtFloat sx, RtFloat sy, RtFloat sz)

Append a linear scaling factor to the current transformation.

```
Scale 2 2 2
RiScale (2, 2, 2);
```

Identity
RiIdentity()

Change the current transformation to identity.

Transform **<16 floats>**
RiTransform (RtMatrix m)

Change the current transformation to that given by the matrix, specified in row major order. Note that it is assumed that transformations are done by postmultiplying a vector by a matrix. In other words, the translation part of the matrix is the bottom row, *not* the right column.

```
Transform [ 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 ]
RiTransform (m);
```

ConcatTransform **<16 floats>**
RiConcatTransform (RtMatrix m)

Append the given matrix to the current transformation. The matrix is specified in row major order. Note that it is assumed that transformations are done by postmultiplying a vector by a matrix. In other words, the translation part of the matrix is the bottom row, *not* the right column.

```
ConcatTransform [ 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 ]
RiConcatTransform (m);
```

***Color* r g b**
RiColor (RtColor c)

> Change the current surface color to (*r, g, b*).

>> ```
>> Color 1 .5 .5
>> RiColor (C);
>> ```

***Opacity* r g b**
RiOpacity (RtColor c)

> Change the current surface opacity to (*r*, *g*, *b*).   Completely transparent is (0,0,0). Completely opaque, which is the default, is (1,1,1).

>> ```
>> RiOpacity (C);
>> Opacity 1 1 1
>> ```

***Surface* name parameters**
RiSurface (RtToken name, ...)

> *name* is a the surface type. Standard surfaces and their acceptable parameters are shown in the following section. A particular RenderMan implementation may include more surface types.

>> ```
>> Surface "matte" "Kd" 0.5
>> RiSurface ("matte", "Kd", &kd, RI_NULL);
>> ```

***LightSource* name sequence parameters**
RiLightSource (RtToken name, ...)

> *name* is one of the light source types available. *sequence* is a numerical argument. The first light source declared should have sequence 1, the second should be 2, etc.

> Standard light sources and their acceptable parameters are explained later in this chapter. Examples of Light source declarations:

```
LightSource "ambientlight" 1 "intensity" 0.5
LightSource "pointlight" 2 "from" [ 0 0 10 ] "intensity" 8
LightSource "distantlight" 3 "from" [ 0 0 0 ] "to" [ 0 0 1 ]

RiLightSource ("ambientlight", "intensity", &intensity, 0.5, RI_NULL);
RiLightSource ("pointlight", "from", from, "intensity", &intensity,
        RI_NULL);
RiLightSource ("distantlight", "from", from, "to", to, RI_NULL);
```

2.5 Specifying Surface Shaders

Surface **name parameters**
RiSurface (RtToken name, ...)

  name is a surface type. Standard surfaces and their acceptable parameters are shown below. A particular RenderMan implementation may include more surface types. Please note that only minimal information about these shaders is given below.

2.5.1 Surface "constant" --- constant shading with no lighting
Parameters and Defaults

  This shader has no parameters.

Description

  This shader simply makes the object appear on the screen exactly as the surface color has been set when the object is instanced. The surface color is set by using the `Color` RIB request. Note that this is the only standard surface shader available which does not require a light source to illuminate it.

Example

```
Color [ 1 .5 .5 ]
Surface "constant"
RiSurface ("constant", RI_NULL);
```

2.5.2 Surface "matte" --- a Lambertian diffuse surface
Parameters and Defaults

  float Ka = 1, Kd = 1;

Description

  This surface is a perfectly uniformly diffuse (Lambertian) surface. There are no specular highlights or reflections on such an object. The base color of the object is given by the graphics state's surface color attribute (set by a `Color` RIB request). *Kd* is the coefficient of diffuse reflection, and *Ka* is the ambient coefficient.

Example

```
Color [ 1 .5 .5 ]
Surface "matte" "Kd" 0.8
RiSurface ("matte", "Kd", &kd, RI_NULL);
```

2.5.3 Surface "metal" --- a metallic surface
Parameters and Defaults

  float Ka = 1, Ks = 1, roughness = .1;

Description

This surface has a metallic appearance with specular highlights from the light sources, but does not have any mirror-like reflections. Therefore it looks more like a metal object with a roughly textured surface.

As is typical of metals, specular highlights on this surface will have the same color as the base color of the object, which is specified by the graphics state's surface color attribute (set by a `Color` RIB request).

Ks is the coefficient of specular reflection, and *Ka* is the ambient coefficient. *roughness* is related to the size of the specular highlight. Note that there is no `"Kd"` parameter, and therefore no uniform diffuse reflection.

Example

```
Color [ 1 1 .5 ]
Surface "metal" "Ks" .8 "roughness" .01
RiSurface ("metal", "Ks", &ks, "roughness", &roughness, RI_NULL);
```

2.5.4 Surface "plastic" --- a plastic surface

Parameters and Defaults

> float Ka = 1, Kd = .5, Ks = .5, roughness = .1;
> color specularcolor = [1 1 1];

Description

This surface has an appearance that looks like plastic: uniform diffuse reflection that is the base color of the object and specular highlights from the light sources. To give an appearance that looks like plastic, the color of the specular highlights should be the same as the color of the light sources. Therefore the value of "specularcolor" should remain as a neutral filter (the default).

Example

```
Color [ 1 1 .5 ]
Surface "plastic" "Ks" .8 "roughness" .01
RiSurface ("plastic", "Ks", &ks, "roughness", &rough, RI_NULL);
```

2.5.5 Surface "paintedplastic" --- a texture mapped plastic surface

Parameters and Defaults

> float Ka = 1, Kd = .5, Ks = .5, roughness = .1;
> color specularcolor = [1 1 1];
> string texturename = " ";

Description

This surface has the same reflectance characteristics as the `"plastic"` surface, but the base color at any particular point is determined by lookup into an image file stored on disk. The `"texturename"` parameter should be set to the name of the disk file containing a texture map image.

Example

```
Color [ 1 1 .5 ]
Surface "paintedplastic" "Kd" .8 "texturename" "myfile.tif"
```

```
char *txtname = "myfile.tif"
RiSurface ("paintedplastic", "Kd", &Kd, "texturename",
          &txtname, RI_NULL);
```

2.6 Specifying Light Sources

2.6.1 Light "ambientlight" --- ambient light source
Parameters and Defaults

> float intensity = 1;
> color lightcolor = [1 1 1];

Description

> This is an ambient light source.

Example

```
LightSource "ambientlight" "intensity" 0.5
```

2.6.2 Light "distantlight" --- solar light
Parameters and Defaults

> float intensity = 1;
> color lightcolor = [1 1 1];
> point from = point "shader" [0 0 0];
> point to = point "shader" [0 0 1];

Description

> This is a sun-like light source. Light rays are parallel, and show no falloff with distance. This light has no actual physical location, but only a direction, which is determined by the vector connecting the "from" and "to" points specified.

Example

```
LightSource "distantlight" "to" [ 1 1 -1 ]
```

2.6.3 Light "pointlight" --- point light source
Parameters and Defaults

> float intensity = 1;
> color lightcolor = [1 1 1];
> point from = point "shader" [0 0 0];

Description

> This is a point light source at the location given by the "from" parameter. The light has a falloff of $1/r^2$. The parameter given by "intensity" is the flux of the light source when one unit from the source position.

Example

```
LightSource "pointlight" "from" [ 1 1 10 ] "intensity" 10
```

2.6.4 Light "spotlight" --- spot light source

Parameters and Defaults

> float intensity = 1;
> color lightcolor = [1 1 1];
> point from = [0 0 0], to = [0 0 1];
> float coneangle = radians(30), conedeltaangle = radians(5);
> float beamdistribution = 2;

Description

    This is a spot light source, illuminating a particular solid angle. The angles must be specified in radians, *not* degrees. Please refer to [Upstill89] or [Pixar89] for more details on the meanings of the parameters.

Example

```
LightSource "spotlight" "from" [ 1 1 10 ] "to" [ 1 1 0 ]
```

2.7 Specifying Geometric Primitives

***Polygon* parameters**
RiPolygon (int nvertices, ...)

    Give a simple convex polygon. The `"P"` parameter is required, all others are optional. The number of vertices is specified implicitly by the number of points given after `"P"`. Points should be given in clockwise order. Valid parameters include:

> `"P"` *points*   Enclosed in brackets, all points in the polygon are given as x,y,z triples.

> `"Cs"` *points*   RGB values are given for colors at each vertex in the polygon. This is how Gouraud shading is done.

> `"N"` *points*   Vectors are given for normals at each vertex in the polygon. This is how Phong shading is done.

Examples:

```
Polygon "P" [ 1 1 0   0 4 5   0 9 0 ]
Polygon "P" [ 0 1 0 0 8 0 4 4 0 ] "N" [ 1 0 0 1 0 0 0 1 0 ]

RiPolygon (3, "P", (float *)p, "N", (float *)n, RI_NULL);
```

***Sphere* radius zmin zmax thetamax**
RiSphere (RtFloat rad, RtFloat zmin, RtFloat zmax, RtFloat thetamax, ...)

    Declare a sphere of given *radius*. Chop off the parts below *zmin* and above *zmax*, and sweep a total of *thetamax* degrees. A full sphere has *zmin = -radius*, *zmax = radius*, and *thetamax*=360.

```
Sphere 1 -1 1 360
```

```
RiSphere (1, -1, 1, 360, RI_NULL);
```

Cone `height radius thetamax`
RiCone (RtFloat height, RtFloat radius, RtFloat thetamax, ...)

Declare a cone with a base of *radius* at the origin, and its tip on the z axis with a z value of *height*. Argument *thetamax* defines the sweep angle in degrees, and should be 360 for a "complete" cone.

```
Cone 5 1 360
RiCone (5, 1, 360, RI_NULL);
```

Cylinder `radius zmin zmax thetamax`
RiCylinder (RtFloat radius, RtFloat zmin, RtFloat zmax, RtFloat thetamax, ...)

Declare a cylinder aligned with the z axis, with given *radius*, minimum and maximum z values and with a sweep angle of *thetamax*.

```
Cylinder 2 0 8 180
RiCylinder (2, 0, 8, 180, RI_NULL);
```

Disk `height radius thetamax`
RiDisk (RtFloat height, RtFloat radius, RtFloat thetamax, ...)

Declare a disk of given radius perpendicular to the z axis and with a z value of height.

```
Disk 5 0.5 360
RiDisk (5, 0.5, 360, RI_NULL);
```

Note: Many more primitives may be declared in RenderMan, including nonconvex polygons with multiple loops, paraboloids, hyperboloids, tori, bilinear and bicubic patches and patch meshes, and NURBS. Please consult *The RenderMan Companion* or *The RenderMan Interface Specification* for more details on these primitives and the use of constructive solid geometry (CSG).

2.8 A Sample RIB File

A sample RIB file of two frames is shown below:

```
Format 512 512 1
PixelSamples 2 2
FrameBegin 1
  Display "t1.tif" "file" "rgb"
  Projection "perspective"  "fov" 45
  Translate 0 -1.5 10
  Rotate -90   1 0 0
  Rotate -10   0 1 0
  WorldBegin
    LightSource "ambientlight" 1 "intensity" 0.5
    LightSource "distantlight" 2 "from" [ 0 0 1 ] "to" [ 0 10 0 ]
    Surface "plastic" "Ka" 0.5  "Kd" 0.8  "Ks" 0.2
    Translate .5 .5 .8
    Sphere 5 -5 5 360
  WorldEnd
FrameEnd
FrameBegin 2
  Display "t2.tif" "file" "rgb"
  Projection "perspective"  "fov" 45
  Translate 0 -2 10
  Rotate -90  1 0 0
  Rotate -20  0 1 0
  WorldBegin
    LightSource "ambientlight" 1 "intensity" 0.5
    LightSource "distantlight" 2 "from" [ 0 0 1 ] "to" [ 0 10 0 ]
    Surface "plastic"  "Ka" 0.5  "Kd" 0.8  "Ks" 0.2
    Translate 1 1 1
    Sphere 8 -8 8 360
  WorldEnd
FrameEnd
```

2.9 A Sample Program

A sample C program to generate two frames is shown below:

```c
#include <math.h>
#include "ri.h"

void main (void)
{
  static RtFloat fov = 45, intensity = 0.5;
  static RtFloat Ka = 0.5, Kd = 0.8, Ks = 0.2;
  static RtPoint from = {0,0,1}, to = {0,10,0};

  RiBegin (RI_NULL);
    RiFormat (512, 512, 1);
    RiPixelSamples (2, 2);
    RiFrameBegin (1);
      RiDisplay ("t1.tif", "file", "rgb", RI_NULL);
      RiProjection ("perspective", "fov", &fov, RI_NULL);
      RiTranslate (0, -1.5, 10);
      RiRotate (-90, 1, 0, 0);
      RiRotate (-10, 0, 1, 0);
      RiWorldBegin ();
        RiLightSource ("ambientlight", "intensity", &intensity,RI_NULL);
        RiLightSource ("distantlight", "from", from, "to", to, RI_NULL);
        RiSurface ("plastic", "Ka", &Ka, "Kd", &Kd, "Ks", &Ks, RI_NULL);
        RiTranslate (.5, .5, .8);
        RiSphere (5, -5, 5, 360, RI_NULL);
      RiWorldEnd ();
    RiFrameEnd ();
    RiFrameBegin (2);
      RiDisplay ("t2.tif", "file", "rgb", RI_NULL);
      RiProjection ("perspective", "fov", &fov, RI_NULL);
      RiTranslate (0, -2, 10);
      RiRotate (-90, 1, 0, 0);
      RiRotate (-20, 0, 1, 0);
      RiWorldBegin ();
        RiLightSource ("ambientlight", "intensity", &intensity,RI_NULL);
        RiLightSource ("distantlight", "from", from, "to", to, RI_NULL);
        RiSurface ("plastic", "Ka", &Ka, "Kd", &Kd, "Ks", &Ks, RI_NULL);
        RiTranslate (1, 1, 1);
        RiSphere (8, -8, 8, 360, RI_NULL);
      RiWorldEnd ();
    RiFrameEnd ();
  RiEnd ();
}
```

3. References

[Upstill89]   Upstill, Steve.   The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics. Addison-Wesley, 1989.

[Pixar89]   Pixar. "The RenderMan Interface, version 3.1 official specification." Published by Pixar, 1989.

[Siggraph90]   "The RenderMan Interface and Shading Language," Siggraph '90 course notes (course 18), 1990.

[Siggraph92]   "Writing RenderMan Shaders," Siggraph '92 course notes (course 21), 1992.

[Gritz93]   Gritz, Larry. "Computing Specular-to-Diffuse Illumination for Two-Pass Rendering," Master's Thesis, The George Washington University, Dept. of EE & CS, May, 1993.