Blue Moon Rendering Tools

User Manual — release 2.5

Larry Gritz
lg@bmrt.org

Blue Moon Systems
200 Kimblewick Dr
Silver Spring, MD 20904

November 1, 1999

Contents

Chapter 1

Introduction

This document explains how to use the Blue Moon Rendering Tools (BMRT), which are a set of rendering programs and libraries which adhere to the RenderMan(TM) standard as set forth by Pixar. This document is intended for the reader who is familiar with the concepts of computer graphics and already is fluent in both the RenderMan procedural interface and the RIB archival format. It is not designed to teach the RenderMan interface, but to give the reader documentation on how to use this implementation of the standard. 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 the 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 3-D scene description format in the same way that PostScript is a 2-D 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 almost no bearing on the use of the 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.

1.2 The Blue Moon Rendering Tools

The *Blue Moon Rendering Tools* described in this document adhere to the RenderMan standard. The parts you'll most likely use are outlined below:

rgl A previewer for RIB files which runs on top of OpenGL. Primitives display as lines or Gouraud-shaded polygons.

rendrib A high quality renderer which uses some of the latest techniques of radiosity and ray tracing to produce near photorealistic images.

slc A compiler for the RenderMan Shading Language, which allows you to write your own surface, light, displacement, imager, and volume shaders.

mkmip A program to pre-process texture, shadow, and environment map files for more efficient access during rendering.

libribout A library of 'C' language bindings for the RenderMan procedural interface which results in the output of RIB (both "static" and "dynamic" libraries are provided for most platforms).

slctell A utility that prints out the arguments and their defaults for a particular compiled shader.

libslc A library allowing you to query the argument names and defaults of a compiled shader.

1.3 Copyrights & Trademarks

The Blue Moon Rendering Tools (BMRT), all of the programs contained therein, and their documentation are:

©Copyright 1990-1999 Larry Gritz. All Rights Reserved.

Larry Gritz retains all copyright and other legal rights to it. I have full documentation that I developed it myself, not for anybody else, and not for any employer.

This software conforms to the RenderMan Interface Standard, developed by Pixar. They require me to state the following:

> The RenderMan (R) Interface Procedures and RIB Protocol are:
> Copyright 1988, 1989, Pixar. All rights reserved.
> RenderMan (R) is a registered trademark of Pixar.

In addition to the RI procedure and RIB protocol, certain components distributed with BMRT are Copyright Pixar. These include the standard shaders (such as "plastic.sl") and elements of the standard header files such as "ri.h."

According to the RenderMan Interface Specification document, anybody may create a program which generates the RenderMan procedure calls or RIB requests, as long as they include Pixar's copyright and trademark notice as above. Also, anybody may write a renderer which executes the RenderMan procedure calls or RIB requests, as long as they get a no-charge license from Pixar authorizing them to do so, which I have done.

This software is what's known as "shareware." This means you can get a copy and "evaluate it" for free. If you decide that you want to keep using it for commercial gain, you are requested to send me a "donation."

1.4 Licensing Arrangement

This software is not in the public domain, but it is "shareware." This means you can get a copy and evaluate it for free, and keep using it for free if you are a student or noncommercial user. If you decide that you want to use it for commercial purposes, you need to register. Please consult the "License" file that comes with the software for more information. I reserve the right to change the licensing of future versions of BMRT at any time and without notice.

Unregistered users may use the software for evaluation purposes for as long as they want. Unregistered users MAY NOT use this software for commercial gain of any kind, including profiting from any images or animations created by this software. Registered users may distribute any images or animations created by this software in any means desired (including for commercial gain). If possible, each copy should credit me as follows:

```
Rendering Software...      Blue Moon Rendering Tools
```

I understand that this is often not practical (for example, for feature films). I trust you'll do your best to give proper credit where possible.

Nobody, not even registered users, may distribute this software in any way without written authorization from me. This is mostly so that I can ensure quality control and keep track of who is using the software.

Don't even think about suing me. Legally, I am warning you right now that this software may not do what you want it to do, and it may do plenty that you don't want. I won't even guarantee that it will not cause damage to your computer, ruin your business, or even result in permanent injury or death. All of these things are pretty unlikely — I've been using this software for years, and it even makes very nice pictures for me. But as far as I'm concerned, *legally* this software is being distributed AS-IS, with no guarantees. Also, I reserve the right to change any of the terms of this software license for future releases.

1.5 Acknowledgments

Many thanks to the early users of my software, especially Daria Bergen, Rudy Darken, Tania Fraga, Dave Florek, Won Lee, several classes of CS-206 at GWU, the many others at the graphics lab at the George Washington University who served as my guinea pigs, and to James Hahn, my advisor. Also a special thanks to Michael

B. Johnson of the MIT Media Lab, who not only helped to shake loose some of the last bugs, but who gave me the kick in the butt that I needed to convince me to distribute my software. Finally, thanks to Tony Apodaca of Pixar, for hand-holding me through the licensing process, as well as illuminating some of the more obscure parts of the standard that nobody cares about but us.

Chapter 2

A Detailed Description of the RenderMan Standard

This user manual assumes that you already have a good working knowledge of the RenderMan Interface Standard or the popular professional implementation, Pixar's *PhotoRealistic RenderMan* ("*PRMan*" for short), or preferably, both. If you are not already familiar with these, then for Pete's sake run out and find the following invaluable references:

- *Advanced RenderMan: Creating CGI for Motion Pictures* by Anthony Apodaca and Larry Gritz, published by Morgan-Kaufmann, 1999.
- *The RenderMan Companion* by Steve Upstill, published by Addison-Wesley, 1989.
- *The RenderMan Interface Specification*, available online from:

  `http://www.pixar.com/products/renderman/toolkit/RISpec/`
- *The PhotoRealistic RenderMan User Manual*, available online from:

  `http://www.pixar.com/products/renderman/toolkit/Toolkit/`

The remainder of this user manual will assume that you have these resources handy and have already digested the bulk of the material contained therein. Therefore, this manual will refrain from describing the RenderMan standard or any features that BMRT has in common with *PRMan*, and instead list only *differences, extensions,* and *incompatibilities* between BMRT and those other published materials.

Chapter 3

Previewing RIB files with *rgl*

Once a RIB file is created, one may use the *rgl* program to display a preview of the scene. Geometric primitives are displayed either as Gouraud-shaded polygons with simple shading and hidden surface removal performed, or as a wireframe image.

The following command will display a preview of the animation in an OpenGL window:

```
rgl myfile.rib
```

There are several command line options which can be given (listed in any order, but prior to the filename). The following sections describe these options. The different options may be used together when they are not inherently contradictory.

If no filename is specified to *rgl*, it will attempt to read RIB from standard input (stdin). This allows you to pipe output of a RIB process directly to *rgl*. For example, suppose that myprog dumps RIB to its standard output. Then you could display RIB frames from myprog as they are being generated with the following command:

```
myprog | rgl
```

The RIB file which you specify may contain either a single frame or multiple frames (if it is an animation sequence). The *rgl* program is designed primarily for previewing animation sequences of many RIB frames. The default is to display all of the frames specified in the RIB file as quickly as possible.

When the last frame is displayed, it will remain in the window. If you hit the ESC key (with the mouse in the drawing window), *rgl* will terminate. If you click on the window with the left mouse button, the entire RIB sequence will be played again.

Though the output of *rgl* is in color, it is important to note that it is not designed to be a particularly accurate preview of a rendered image. It really cannot be, since there is no way for *rgl* to know very much about the types of shaders which you are using. It does a fairly good job of matching ambient, point, distant, and spot lights. But it can't figure out area lights or any nonstandard light source types. Also, every surface is displayed as if it were "matte," regardless of the actual surface specification.

Note that *rgl* can also display primitives as lines. This is done by invoking:

```
rgl -lines myfile.rib
```

This completely replaces the old *rendribv* program.

3.1 Command Line Options

The following subsection details command line options alter the way in which *rgl* creates and/or displays images.

3.1.1 Window Size and Position

`-res` *`xres yres`*

Sets the resolution of the output window. Note that if the RIB file contains a `Format` statement which explicitly specifies the image resolution, then the `-res` option will be ignored and the window will be opened with the resolution specified in the `Format` statement.

3.1.2 Drawing Styles

`-1buffer`

Rather than render the polygon preview to the "back buffer" and displaying frames as they finish (as you would want especially if you are previewing an animation), this option draws to the front buffer, thus allowing you to see the scene as rendering progresses. The `-1buffer` option may be used in combination with any of the other drawing style options.

`-unlit`

Lights all geometry with a single light at the camera position. This is useful for using *rgl* to preview a RIB file that does not contain light sources. The `-unlit` option may be used in combination with any of the other drawing style options.

`-lines`

Rather than the default drawing mode of filled-in Gouraud-shaded polygons, this option causes the images to be rendered as lines. Note that this cannot be used in combination with `-sketch`.

`-sketch`

It's not clear what the real use of this is, but it makes an image that looks a little like a human-drawn sketch of the objects. Note that this cannot be used in combination with `-lines`.

`-rd` *`multiplier`*

You can speed up *rgl* by changing the refinement detail that it uses to convert curved surfaces to polygons by using the `-rd` command line option, which takes a single numerical argument, generally between 0 and 1. The lower the value, the fewer polygons will be used to approximate curved surfaces. Using a value of 1 will result in identical results as if you did not use the `-rd` option at all. Good values to try are 0.75 and 0.5. If you go below 0.25, the curved surface primitives may become unrecognizable, though they will certainly be drawn quickly. If you use values larger than 1, even more polygons than usual will be used to approximate the curved surfaces.

IMPORTANT NOTE: the `-rd` option can only speed up the rendering of curved surface primitives (e.g. spheres, cylinders, bicubic patches, NURBS). It WILL NOT speed up the drawing of polygons. If your model contains too many polygons to be drawn quickly, the `-rd` option will not help you.

3.1.3 File Output Options

`-dumprgba`
`-dumprgbaz`

The default operation of *rgl* simply previews the scene to a window on your display. But using the `-dumprgba` option instead causes the resulting preview image to be saved to a TIFF file. The filename of the TIFF file is taken from the `Display` RIB command in the file itself, or `ri.tif` if no `Display` command is present in the RIB file. The `-dumprgbaz` option does the same thing as `-dumprgba`, but also saves the z buffer values to a file. The z values are saved in the same zfile format used by Pixar's *PhotoRealistic RenderMan*, and the name of the file is also taken from the `Display` RIB command, substituting "zfile" for "tif" in the filename.

`-offscreen`

When used in conjunction with either `-dumprgba` or `-dumprgbaz`, causes the image file(s) to be created without ever opening a window to the screen. This is handy for using *rgl* as a low quality batch renderer. Note that this option only works on some OpenGL implentations — in particular it will not work on SGI's, though it will work on the Linux and Sun ports of BMRT (which use Mesa for their OpenGL implementations).

3.1.4 Animation

`-frames` *`first last`*

Sometimes you may only want to preview a subset of frames from a multi-frame RIB file. You can do this by using the `-frames` command line option. This option takes two integer arguments: the first and last frame numbers to display. If you are going to use this option, it is recommended that your frames be numbered sequentially starting with 0 or 1.

`-sync` *`framespersecond`*

When previewing a series of frames for an animation, it is often necessary to synchronize the display of frames to the clock in order to check the timing of the animation when it is played back at a particular number of frames per second. The default action of *rgl* is to display the frames as fast as possible. You can override this, causing *rgl* to try to display a particular number of frames per second, by using the `-sync` command line option.

`-nowait`

By default, the last frame will stay in the drawing window until you hit the ESC key. The `-nowait` causes *rgl* to terminate immediately after displaying the last frame in the sequence (for example, if it is part of an automated demo).

3.2 Implementation-dependent Options and Attributes

The RenderMan Interface Specification allows various implementation-specific behaviors of a renderer to be set using two RIB directives: `Option` and `Attribute`. Options apply to the entire scene and should be specified prior to `WorldBegin`. Attributes apply to specific geometry, are generally set after the `WorldBegin` statement, and bind to subsequent geometry.

3.2.1 Search Paths

Various external files may be needed as the renderer is running, and unless they are specified as fully-qualified file paths, the renderer will need to search through directories to find those files. There exists an option to set the lists of directories in which to search for these files.

`Option "searchpath" "archive" [`*`pathlist`*`]`
`Option "searchpath" "procedural" [`*`pathlist`*`]`

Sets the search path that the renderer will use for files that are needed at runtime. The `"archive"` path specifies where to find RIB files that are inclued using the `ReadArchive` directive. The `"procedural"` path specifies where to find programs and DSO's that are required by `RiProcedural`.

Search path types in BMRT are specified as colon-separated lists of directory names (much like an execution path for shell commands). There are two special strings that have special meaning in BMRT's search paths:

- `&` is replaced with the *previous* search path (i.e., what was the search path before this statement).

- `$ARCH` is replaced with the name of the machine architecture (such as `linux`, `sgi_m3`, etc.). This allows you to keep compiled software (like DSO's) for different platforms in different directories, without having to hard-code the platform name into your RIB file.

For example, you may set your procedural path as follows:

```
  Option "searchpath" "procedural"
          ["/usr/local/bmrt:/usr/local/bmrt/$ARCH:&"]
```

The above statement will cause the renderer to find procedural DSO's by first looking in `/usr/local/bmrt`, then in a directory that is dependent on the architecture, then wherever the default (or previously set) path indicated.

3.2.2 Drawing Options

`Option "limits" "curvethinning" [`*`frequency`*`]`
`Option "limits" "curvethinthreshold" [`*`thresh`*`]`

> When *rgl* draws many RiCurves primitives, it can turn into a big unshaded mess. It may be that you decide that drawing fewer curves actually makes a more understandable preview. The `"curvethinning"` frequency value tells how often a curve should be drawn: a value of 2 indicates to draw every other hair, a value of 100 means that only every 100th hair should be drawn. Furthermore, this thinning is only performed for RiCurves statements that have more individual hairs than is specified with the `"curvethinthreshold"` parameter. Both take integer arguments. If the `"curvethinning"` frequency is set to zero, no curve thinning will take place at all.

`Attribute "division" "udivisions" [`*`nu`*`]`
`Attribute "division" "vdivisions" [`*`nv`*`]`

> *rgl* will dice curved primitives into flat polygons for OpenGL to draw. It basically guesses at how many polygons to subdivide into, and it usually chooses well enough for previews, but sometimes you may want to override the dicing criteria. This option allows you to explicitly specify how many subdivisions to make in subsequently curved surfaces. The arguments *nu* and *nv* are both integers.

3.3 Limitations of *rgl*

Since *rgl* is an OpenGL-based polygon previewer, it cannot possibly support all the features of the RenderMan Interface which would be supported by other types of renders. This section outlines the features which are not fully supported by *rgl*.

- The following RIB directives are ignored because they have no real meaning in an OpenGL previewer: `ColorSamples`, `DepthOfField`, `Shutter`, `PixelVariance`, `PixelSamples`, `PixelFilter`, `Exposure`, `Imager`, `Quantize`, `Hider`, `Atmosphere`, `Bound`, `Opacity`, `TextureCoordinates`, `ShadingRate`, `ShadingInterpolation`, `Matte`.
- The `LightSource` directive works as expected for `"ambientlight"`, `"distantlight"` and `"pointlight"`. It isn't smart enough to know exactly what to do for custom light source shaders, but it will try to make its best guess by examining the parameters to the shader, looking for clues like `"from"`, `"to"`, `"lightcolor"`, and so on. The `AreaLightSource` directive has no effect.
- Shaders do nothing. All surfaces are displayed as if they were using the standard `matte.sl` shader.
- When motion blocks are given, only the first time key is used.
- Multiple levels of detail are not supported.
- Solids are all displayed as unions, i.e., all of the components of a CSG primitive are displayed.
- Object instancing is not currently working. Instanced objects are ignored.
- Texture map generation functions (e.g., `MakeTexture`) do nothing in *rgl*.
- New primitives: *rgl* will correctly draw the new `Curves` and `Points` primitives (as dots and line segments, respectively). `SubdivisionMesh` primtives will be drawn as their control hull (with smoothed vertex normals). `Blobby` primitives are currently ignored by *rgl*.

3.4 Odds and Ends

There are a bunch of other things you should know about *rgl* but I coundn't figure out where it went in the manual. In no particular order:

- Before rendering any RIB specified on the command line or piped to it, *rgl* will first read the contents of the file `$BMRTHOME/.rendribrc`. If there is no environment variable named `$BMRTHOME`, then the file `$HOME/.rendribrc` is read instead. In either case, by putting RIB in one of these places, you can set various options for *rgl* before any other RIB is read.

Chapter 4

Photo-realistic rendering with *rendrib*

The *rendrib* program is a high-quality renderer incorporating the techniques of ray tracing and radiosity to make (potentially) very realistic images. This renderer supports not only the required features of the RenderMan Interface, but also many of the more advanced Optional Capabilities, such as: ray tracing, radiosity, solid modeling, depth of field, motion blur, area light sources, texture mapping, environment mapping, displacements, volume and imager shading, and support of the RenderMan Shading Language.

The format for invoking *rendrib* is as follows:

```
rendrib [options] myfile.rib
```

Usually, this will result in one or more TIFF image files to be written to disk. If the RIB file specified framebuffer display (as opposed to file), or you override with the `-d` flag, the resulting image will be displayed as a window on your screen. When the rendering is complete, *rendrib* will pause. Hitting the ESC key will terminate. Alternately, if you hit the 'w' key, the image in the window will be written to a file (using the filename specified in the RIB file's Display command).

If no filename is specified to *rendrib*, it will attempt to read RIB from standard input (stdin). This allows you to pipe output of a RIB process directly to *rendrib*. For example, suppose that myprog dumps RIB to its standard output. Then you could display RIB frames from myprog as they are being generated with the following command:

```
myprog | rendrib
```

The RIB file which you specify may contain either a single frame or multiple frames (if it is an animation sequence).

4.1 Command Line Options

The following subsection details command line options alter the way in which *rendrib* creates and/or displays images.

4.1.1 Image Display Options

`-d [`*`interleave`*`]`

By default, any fully rendered frames are sent to a TIFF image file (unless, of course, the RIB file specifies `framebuffer` output with the `Display` directive). The `-d` command line option overrides file output and forces output to be sent to a screen window. If the optional integer *interleave* is specified, scanlines will be computed in an interleaved fashion, giving you a kind of progressive refinement display. For example,

```
rendrib -d 8 myfile.rib
```

will display every 8th scanline first (making a very quick, but blocky image), then compute every 4th scanline, then every 2nd, and so on, until you get the final image. This is extremely useful if you want to quickly see a rough version of the scene.

`-res` *`xres yres`*

Sets the resolution of the output image. Note that if the RIB file contains a `Format` statement which explicitly specifies the image resolution, then the `-res` option will be ignored and the window will be opened with the resolution specified in the `Format` statement.

`-pos` *`xpos ypos`*

Specifies the position of the window on your display (obviously, this only works if used in combination with the `-d` option or if your `Display` line in your RIB file specifies `framebuffer` output).

`-crop` *`xmin xmax ymin ymax`*

Specify that only a portion of the whole image should be rendered. The meaning of this command line switch is precisely the same as if the `CropWindow` directive was in your RIB file (and like the other options of this section, a `CropWindow` option in your RIB takes precedence over any command line arguments).

`-samples` *`xsamp ysamp`*

Sets the number of samples per pixel to *xsamp* (horizontal) by *ysamp* (vertical). Note that if the RIB file contains a `PixelSamples` statement which explicitly specifies the sampling rate, then the `-samples` option will be ignored and the sampling rate will be as specified by the `PixelSamples` statement.

`-var` *`variance minsamples maxsamples`*

An alternative to setting a fixed number of samples per pixel is to use variance-based sampling. This method samples highly only in regions with high variance. To invoke this option, you can either put a `PixelVariance` directive in your RIB file and set the minimum and maximum samples per pixel using the `Option "render" "minsamples"` and `Option "render" "maxsamples"`, or use the `-var` command line option. This option takes one floating-point and two integer arguments: the desired maximum variance, the minimum number of samples per pixel, and the maximum number of samples per pixel. Here is an example:

```
rendrib -res 320 240 -var 0.01 4 32 myfile.rib
```

Note that this example set the maximum variance in a pixel to 0.01, which totally overrides the `PixelSamples` directive or `-samples` option. In this example, all pixels will have at least 4, and at most 32 samples. For most effective variance-based sampling, you need to properly set the minimum and maximum numbers of samples per pixel. This is explained in the "implementation-specific Options" section of this chapter.

NOTE: in my experience, `PixelSamples` (or `-samples`) is the way to go. The `PixelVariance` may seem like a good idea, but in practice is not as good as using a fixed number of samples per pixel.

4.1.2 Status Output

`-stats`

Upon completion of rendering, output various statistics about memory and time usage, number of primitives, and all sorts of other debugging information. Using this option on the command line is equivalent to putting `Option "statistics" "endofframe" [1]` in your RIB file.

`-v`

Verbose output — this prints more status messages as rendering progresses, such as the names of shaders and textures as they are loaded.

You can combine the `-v` and `-stats` options if you want.

4.1.3 Radiosity

`-radio` *`steps`*

By default, *rendrib* calculates images using the rendering technique of ray tracing. Ray tracing alone does no energy balancing of the scene. In other words, it does not account for interreflected light. However, *rendrib* supports radiosity, which is a method for performing these calculations. You can instruct *rendrib* to perform a radiosity pass prior to the ray tracing by using the `-radio` command line switch. This command is followed by a single integer argument, which is the number of radiosity steps to perform. For example, the following command causes *rendrib* to perform 50 radiosity steps prior to forming its image:

```
rendrib -radio 50 myfile.rib
```

If the energy is balanced in fewer steps than you specify, *rendrib* will skip the remaining steps (saving time). Depending on your scene, the radiosity calculations can take a long time, but they are independent of the final resolution of your image.

Specifying the number of radiosity steps on the command line is exactly equivalent to including a `Option "radiosity" "nsteps"` line in your RIB file.

`-rsamples` *`samps`*

By default, *rendrib* calculates the visibility between geometric elements by casting a minimum of one ray between the two elements. You can increase this number to get better accuracy (but at a big decrease in speed) by using the `-rsamples` option. This option takes a single integer argument. The minimum number of rays used to determine visibility will be the square of this argument. For example, the following command will perform a radiosity pass of 100 steps, using a minimum of 4 sample rays per visibility calculation:

```
rendrib -radio 100 -rsamples 2 myfile.rib
```

4.1.4 Miscellaneous Options

`-frames` *`first last`*

Sometimes you may only want to render a subset of frames from a multi-frame RIB file. You can do this by using the `-frames` command line option. This option takes two integer arguments: the first and last frame numbers to display. For example,

```
rendrib -frames 10 10 myfile.rib
```

This example will render only frame number 10 from this RIB file. If you are going to use this option, it is recommended that your frames be numbered sequentially starting with 0 or 1.

`-safe`

When you submit a RIB file for rendering, the image files will have filenames as specified in the RIB file with the Display directive. If a file already exists with the same name, the original file will be overwritten with the new image. Sometimes you may want to avoid this. Using the -safe command line option will abort rendering of any frame which would overwrite an existing disk file. This is mostly useful if you are rendering many frames in a sequence, and do not want to overwrite any frames already rendered. Here is an example:

```
rendrib -safe -frames 100 200 myfile.rib
```

This example will render a block of 100 frames from the RIB file, but will skip over any frames which happen to already have been rendered.

`-ascii`

Will produce an *ASCII* (yes, exactly what you think) representation of your scene to the terminal window!

`-beep`

Rings the terminal bell upon completion of rendering.

`-arch`

Just print out the architecture name (e.g., `sgi_m3`, `linux`, etc.).

4.2 Implementation-dependent Options and Attributes

The RenderMan Interface Specification allows various implementation-specific behaviors of a renderer to be set using two RIB directives: `Option` and `Attribute`. Options apply to the entire scene and should be specified prior to `WorldBegin`. Attributes apply to specific geometry, are generally set after the `WorldBegin` statement, and bind to subsequent geometry.

Several of the features of this renderer can be controlled as nonstandard options. The mechanism for this is to use the `Option` RIB directive. The syntax for this is:

```
Option name params
```

Where *name* is the option name, and *params* is a list of token/value pairs which correspond to this option. Remember that options apply to an entire rendered frame, while attributes apply to specific pieces of geometry.

Similarly, other renderer features can be controlled as nonstandard attributes, with the following syntax:

```
Attribute name params
```

Attributes apply to specific pieces of geometry, and are saved and restored by the `AttributeBegin` and `AttributeEnd` commands.

Remember that both of BMRT's renderers (*rendrib* and *rgl*) read from a file called `.rendribrc` both in the local directory where it is run, and also in your home directory. This file can be plain RIB, which means that if you want to set any defaults of the options discussed below, you can just put the Option or Attribute lines in this file in your home directory.

The remainder of this chapter explain the various nonstandard options and attributes supported by *rendrib*. In most cases, the new "inline declaration" syntax is used to clarify the expected data types, and the default values are provided as examples.

4.2.1 Rendering Options

`Option "render" "integer minsamples" [8]`
`Option "render" "integer maxsamples" [64]`

Sets the minimum and maximum number of samples per pixel, for scenes when the pixel variance metric is used. This only works if the RIB file contains a PixelVariance statement, but not a `PixelSamples` statement.

`Option "render" "integer max_raylevel" [4]`

Sets the maximum number of recursive rays that will be cast between reflectors and refractors. This has no effect if there are no truly reflective or refractive objects in the scene (in other words, shaders which use the trace function).

`Option "render" "float minshadowbias" [0.01]`

Sets the minimum distance that one object has to be in order to shadow another object. This keeps objects from self-shadowing themselves. If there are serious problems with self-shadowing, this number can be increased. You may need to decrease this number if the scale of your objects is such that 0.01 is on the order of the size of your objects. In general, however, you will probably never need to use this option if you don't notice self-shadowing artifacts in your images.

`Option "render" "integer prmanspecular" [1]`

Pixar's *PRMan* does not use the SL specular() function as listed in the RI standard. BMRT uses the standard function, which results in specular highlights looking very different in the two renderers. This option, which takes an integer, causes BMRT's specular function to behave much more like *PRMan*'s with a value of 1, and uses the specular function from the standard when given a value of 0. **The default used to be zero (standard), but starting with BMRT 2.3.4, the default is 1 (*PRMan*). Use this option with a value of 0 in order to revert to the old (RI standard) behavior.**

`Option "render" "integer useprmandspy" [1]`

If this option is passed a nonzero integer value (the default) and the `Display` request a driver type that *rendrib* doesn't know about, *rendrib* will search for installed *PRMan* display drivers and use the appropriately named one. This allows you to use any of *PRMan*'s display drivers, including any custom ones that you have written as DSO's.

`Option "statistics" "integer endofframe" [0]`

When nonzero, this option will cause *rendrib* to print out various statistics about the rendering process. Greater values print more detailed data: 1 just prints time and memory information, 2 gives more detail, 3 is all the data that the renderer ever wants to print. (Usually 2 is just fine for lots of data.)

`Option "statistics" "string filename" [""]`

When non-null, this option will cause *rendrib*'s statistics to be echoed to the given filename, rather than printed to `stdout`.

4.2.2 Search Paths

Various external files may be needed as the renderer is running, and unless they are specified as fully-qualified file paths, the renderer will need to search through directories to find those files. There exists an option to set the lists of directories in which to search for these files.

`Option "searchpath" "archive" [`*pathlist*`]`
`Option "searchpath" "texture" [`*pathlist*`]`
`Option "searchpath" "shader" [`*pathlist*`]`
`Option "searchpath" "procedural" [`*pathlist*`]`
`Option "searchpath" "display" [`*pathlist*`]`

Sets the search path that the renderer will use for files that are needed at runtime.

The different search paths recognized by *rendrib* are:

`archive` RIB filess included by `ReadArchive`.

`texture` texture image files.

`shader` compiled shaders.

`procedural` DSO's and executables for `Procedural` calls.

`display` DSO's for custom PRMan display services.

Search path types in BMRT are specified as colon-separated lists of directory names (much like an execution path for shell commands). There are two special strings that have special meaning in BMRT's search paths:

- `&` is replaced with the *previous* search path (i.e., what was the search path before this statement).
- `$ARCH` is replaced with the name of the machine architecture (such as `linux`, `sgi_m3`, etc.). This allows you to keep compiled software (like DSO's) for different platforms in different directories, without having to hard-code the platform name into your RIB file.

For example, you may set your procedural path as follows:

```
Option "searchpath" "procedural"
       ["/usr/local/bmrt:/usr/local/bmrt/$ARCH:&"]
```

The above statement will cause the renderer to find procedural DSO's by first looking in `/usr/local/bmrt`, then in a directory that is dependent on the architecture, then wherever the default (or previously set) path indicated.

4.2.3 Visibility of Primitives

`Attribute "render" "integer visibility" [7]`

Controls which rays may see an object. The integer parameter is the sum of:

1 The object is visibile from primary (camera) rays.

2 The object is visibile from reflection rays.

4 The object is visibile from shadow rays.

This attribute is useful for certain special effects, such as having an object which appears only in the reflections of other objects, but is not visible when the camera looks at it. Or an object which only casts shadows, but is not in reflections or is not seen from the camera.

`Attribute "render" "string casts_shadows" ["Os"]`

Controls how surfaces shadow other surfaces. Possible values for shadowval are shown below, in order of increasing computational cost:

`"none"` The surface will not cast shadows on any other surface, therefore it may be ignored completely for shadow computations.

`"opaque"` The surface will completely shadow any object which it occludes. In other words, this tells the renderer to treat this object as completely opaque.

`"Os"` The surface may partially shadow, depending on the value set by the Opacity directive. In other words, it has a constant opacity across the surface. (This is the default.)

`"shade"` The surface may have a complex opacity pattern, therefore its surface shader should be called on a point-by-point basis to determine its opacity for shadow computations.

The default value is `"Os"`. You can optimize rendering time by making surfaces known to be opaque `"opaque"`, and surfaces known not to shadow other surfaces `"none"`. It is important, however, to use `"shade"` for any surfaces whose shaders modify the opacity of the surface in any patterned way.

4.2.4 Displacement and Subdivision Attributes

`Attribute "render" "integer truedisplacement" [0]`

If the argument is nonzero, subsequent primitives will truly be diced and displaced using their displacement shader (if any). If the value of 0 is used, bump mapping will be used rather than true displacement. Only a displacement shader can move the diced geometry – altering `P` in a surface shader will not move the surface, only the normals. Using a displacement shader without this attribute also only results in the normals being modified, but not the surface. Be sure to set displacement bounds if you displace! Please see the section on "limitations of *rendrib*" for details on the limitations placed on true displacements.

```
Attribute "displacementbound" "string coordinatesystem" ["current"]
          "float sphere" [0]
```

For truly displaced surfaces, specifies the amount that its bounding box should grow to account for the displacement. The box is grown in all directions by the `radius` argument, expressed in the given coordinate system (a string). This works just like the nonstandard option of the same name in *PRMan*.

```
Attribute "render" "float patch_multiplier" [1.0]
```

Takes an float argument giving a multiplier for the dicing rate that BMRT computes for displaced surfaces and for certain curved surfaces which are subdivided. Smaller values will make the scene render faster and using less memory, but may produce a more faceted appearance to certain curved surfaces. Larger values will make more accurate surfaces, but will take longer and more memory to render. The default is probably just right for 99occasionally you may need to tweak this.

```
Attribute "render" "float patch_maxlevel" [256]
Attribute "render" "float patch_minlevel" [1]
```

Takes an integer argument giving the maximum (or minimum) subdivision level for bicubic and NURBS patches. These patches are subdivided based on the screen size of the patch and their curvature. This attribute will split the patches into at least (minlevel x minlevel) and at most (maxlevel x maxlevel) subpatches. The default is min=1, max=256. In general, you shouldn't ever need to change this, but occasionally you may need to set a specific subdivision rate for some reason.

4.2.5 Object Appearance

```
Attribute "trimcurve" "string sense" ["inside"]
```

By default, trim curves on NURBS will make the portions of the surface that are *inside* the closed curve. You can reverse this property (by keeping the inside of the curve and throwing out the part of the surface outside the curve) by setting the trimcurve sense to `"outside"`.

```
Attribute "render" "integer use_shadingrate" [1]
```

When non-zero (the default), *rendrib* will attempt to share shaded colors among nearby screen rays that strike the same object (specifically, it shares among rays that are within the screen space area defined by the `ShadingRate`). Occasionally, you may see a blocky or noisy appearance resulting from this shared computation. In such a case, setting this attribute to 0 will cause subsequent primitives to compute their shading for *every* screen ray, resulting in much more accurate color (though at a higher cost).

4.2.6 Light Source Attributes

`Attribute "light" "string shadows" ["off"]`

Turns the automatic ray cast shadow calculations on or off on a light-by-light basis. This attribute can be used for any LightSource or AreaLightSource which is declared. For example, the following RIB fragment declares a point light source which casts shadows:

```
Attribute "light" "shadows" ["on"]
LightSource "pointlight" 1 "from" [ 0 10 0 ]
```

`Attribute "light" "integer nsamples" [1]`

Sets the number of times to sample a particular light source for each shading calculation. This is only useful for an area light which is being undersampled — i.e., its soft shadows are too noisy. By increasing the number of samples, you can reduce the noise by increasing sampling of this one light, independently of overall PixelSamples.

`Attribute "light" "integer samplingstrategy" [0]`

Selects among different area light sampling strategies. Currently only 0 (the old default) and 1 (a new, slightly different pattern) are available. It's unclear if 1 is really any better than 0 – try it and find out.

4.2.7 Radiosity Controls

Two options are used to enable radiosity computations and control how much work is to be done in computing the indirect illumination:

`Option "radiosity" "integer steps" [0]`

In addition to using the `-radio` command line option to *rendrib*, you can specify the number of radiosity steps with this option. Setting steps to 0 indicates that radiosity should not be used. Nonzero indicates that radiosity should be used (with the given number of steps) even if the `-radio` command line switch is not given to *rendrib*.

`Option "radiosity" "integer minpatchsamples" [1]`

Just like the `-rsamples` command line option to *rendrib*, this option lets you set the minimum number of samples per patch to determine radiosity form factors. Actually, the minimum total number of samples per patch is this number squared (since it is this number in each direction). In some cases, the render will decide to use more samples, but this is the minimum.

A number of attributes control specific features of the radiosity computations on a per-primitive basis. These attributes have absolutely no effect if you are not performing radiosity calculations.

`Attribute "radiosity" "color averagecolor" [`*`color`*`]`

By default, the radiosity renderer assumes that the diffuse reflectivity of a surface is the default color value (set by Color) times the Kd value sent to the shader for that surface. For the lighting calculations to be accurate, the reflective color should be the average color of the patch. For surfaces with a solid color, this is fine. However, some surface shaders create surfaces whose average colors have nothing to do with the color set by the Color directive. In this case, you should explicitly set the average color using the attribute above. You may have to guess what the average color is for a particular surface.

`Attribute "radiosity" "color emissioncolor" [`*`color`*`]`

All surfaces which are not light sources (Lightsource or AreaLightsource) are assumed to be reflectors only (i.e. they do not glow). If you want a piece of geometry to actually emit radiative energy into the environment, you can either declare it as an AreaLightSource, or you could declare it as regular geometry but give it an emission color (see above). The tradeoffs are discussed further in the radiosity section of this chapter.

```
Attribute "radiosity" "float patchsize" [4]
Attribute "radiosity" "float elemsize" [2]
Attribute "radiosity" "float minsize" [1]
```

This attribute tells *rendrib* how finely to mesh the environment for radiosity calculations. The statement above instructs to chop all geometry into patches no larger than 4 units on a side. Each patch is then diced into elements no larger than 2 units on a side. As a result of analyzing the radiosity gradients, elements may be diced even finer, but a particular element will not be diced if its longest edge is shorter than 1 unit. The smaller these numbers, the longer the radiosity calculation will take (but it will be more accurate). This attribute can be used to set these numbers on a surface-by-surface basis (i.e., different surfaces in the scene may have different dicing rates). The values are measured in the *current* (i.e., local) coordinate system in effect at the time of this Attribute statement. **NOTE: The default values are probably bad — if you are using radiosity, you should set these to appropriate sizes for your particular scene.**

`Attribute "radiosity" "string zonal" ["fully_zonal"]`

This attribute controls which radiosity calculations are performed on surfaces. This can be set on a surface-by-surface basis. Possible values are shown below, in order of increasing computational cost:

`"none"` The surface will neither shoot or receive energy, i.e. it will be ignored by the radiosity calculation.

`"zonal_receives"` The surface receives radiant energy, but does not shoot it back into the environment.

`"zonal_shoots"` The surface reflects (or emits) energy, but does not receive energy from other patches.

`"fully_zonal"` The surfaces both receives and shoots energy. This is the default zonal property of materials.

4.2.8 Other Options

`Option "limits" "integer texturememory" [1000]`

Sets the texture cache size, measured in Kbytes. The renderer will try to keep no more than this amount of memory tied up with textures. Setting it low keeps memory consumption down if you use many textures. But setting it too low may cause thrashing if it just can't keep enough in cache. The default is 1000 (i.e., 1 Mbyte). The texture cache is only used for *tiled* textures, i.e. those made with the *mkmip* program. For regular scanline TIFF files, texture memory can grow very large.

`Option "limits" "integer geommemory" [`*`unlimited`*`]`

Analogous to the texturememory option, this sets a limit to the amount of memory used to hold the diced pieces of NURBS, bicubics, and displaced geometry. It is an integer, giving a measurement in Kbytes. The default is unlimited, but setting this to something smaller (like 100000, or 100 M-bytes) can keep your memory consumption down for large scenes, but setting it too low may cause you to continually be throwing out and regenerating your NURBS or displaced surfaces.

`Option "limits" "integer derivmemory" [2]`

A certain amount of memory is needed to allow *rendrib*'s Shading Language interpreter to correctly compute derivatives. Very occasionally, you may need to increase this number (generally only if you have absolutely humongous shaders with many texture or other derivative calls). The default is 2 (i.e., 2 Kbytes), which is almost always adequate. If your frames are not crashing mysteriously in the shaders, don't screw with this number!

`Option "runtime" "string verbosity" ["normal"]`

This option controls the same output as the `-v` and `-stats` command line options. The verb parameter is a string which controls the level of verbosity. Possible values, in order of increasing output detail, are: `"silent"`, `"normal"`, `"stats"`, `"debug"`.

4.3 Nonstandard Shading Language Features

In order to more fully support the ray tracing features of BMRT, a few nonstandard functions have been added to BMRT's Shading Language. The next chapter describes how to use the Shading Language Compiler.

Note that these functions are **not** supported by *PRMan*'s SL compiler. However, using the "ray server," PRMan's functionality can be extended to support these functions by using BMRT to compute them.

`color trace (point from, vector dir)`

> Traces a ray from position `from` in the direction of vector `dir`. The return value is the incoming light from that direction.

`color visibility (point p1, p2)`

> Forces a visibility (shadow) check between two arbitrary points, retuning the spectral visibility between them. If there is no geometry between the two points, the return value will be (1,1,1). If fully opaque geometry is between the two points, the return value will be (0,0,0). Partially opaque occluders will result in the return of a partial transmission value.
>
> An example use of this function would be to make an explicit shadow check in a light source shader, rather than to mark lights as casting shadows in the RIB stream (as described in the previous section on nonstandard attributes). For example:

```
light
shadowpointlight (float intensity = 1;
               color lightcolor = 1;
          point from = point "shader" (0,0,0);
                  float raytraceshadow = 1;)
{
    illuminate (from) {
        Cl = intensity * lightcolor / (L . L);
        if (raytraceshadow != 0)
            Cl *= visibility (Ps, from);
    }
}
```

```
float rayhittest (point from, vector dir,
                 output point Ph, output vector Nh)
```

> Probes geometry from point `from` looking in direction `dir`. If no geometry is hit by the ray probe, the return value will be very large (1e38). If geometry is encountered, the position and normal of the geometry hit will be stored in `Ph` and `Nh`, respectively, and the return value will be the distance to the geometry.

```
 float fulltrace (point pos, vector dir,
                 output color hitcolor, output float hitdist,
                 output point Phit, output vector Nhit,
                 output point Pmiss, output point Rmiss)
```

Traces a ray from `pos` in the direction `dir`.

If any object is hit by the ray, then `hitdist` will be set to the distance of the nearest object hit by the ray, `Phit` and `Nhit` will be set to the position and surface normal of that nearest object at the intersection point, and `hitcolor` will be set to the light color arriving from the ray (just like the return value of `trace`).

If no object is hit by the ray, then hitdist will be set to 1.0e30, hitcolor will bet set to (0,0,0).

In either case, in the course of tracing, if any ray (including subsequent rays traced through glass, for example) ever misses all objects entirely, then `Pmiss` and `Rmiss` will be set to the position and direction of the deepest ray that failed to hit any objects, and the return value of this function will be the depth of the ray which missed. If no ray misses (i.e. some ray eventually hits a nonreflective, nonrefractive object), then the return value of this function will be zero. An example use of this functionality would be to combine ray tracing of near objects with an environment map of far objects.

The code fragment below traces a ray (for example, through glass). If the ray emerging from the far side of the glass misses all objects, it adds in a contribution from an environment map, scaled such that the more layers of glass it went through, the dimmer it will be.

```
missdepth = fulltrace (P, R, C, d, Ph, Nh, Pm, Rm);
if (missdepth > 0)
    C += environment ("foo.env", Rm) / missdepth;
```

`float isshadowray ()`

Returns 1 if this shader is being executed in order to evaluate the transparency of a surface for the purpose of a shadow ray. If the shader is instead being evaluated for visible appearance, this function will return 0. This function can be used to alter the behavior of a shader so that it does one thing in the case of visibility rays, something else in the case of shadow rays.

`float raylevel ()`

Returns the level of the ray which caused this shader to be executed. A return value of 0 indicates that this shader is being executed on a camera (eye) ray, 1 that it is the result of a single reflection or refraction, etc. This allows one to customize the behavior of a shader based on how “deep” in the reflection/refraction tree.

4.4 Global Illumination

The purpose of the RenderMan standard is to allow one to specify the description of a scene without having to worry about how to render it. Thus, no mention is made in the standard of any particular rendering method. A particular implementation of the standard may support any one (or many) of the following: wireframe; flat shaded, Gouraud shaded, or Phong shaded z-buffered polygons; A-buffer; REYES; ray tracing; radiosity; or any other rendering method. Since this renderer stems largely from my own research in the area of illumination, it incorporates many of the latest advances in global illumination and rendering technology.

The default rendering method used by this renderer is ray tracing. Of course, if you only use standard surfaces and light sources, the results will not be very dramatic. But the shaders which use the trace() function can cast reflection and refraction rays. Light sources which cast ray-traced shadows can be added automatically, even from area light sources. However, ray tracing will still determine illumination only via direct paths from light sources to surfaces being shaded. No knowledge of interreflection between objects is available to the ray tracer.

That's where *radiosity* come into the picture. Radiosity will help to capture effects such as indirect illumination, soft shadows, and color bleeding. To render the scene using radiosity, just type:

```
rendrib -radio n myfile.rib
```

The parameter n is a number giving the maximum number of radiosity steps to perform. A typical number might be 50. Higher values of n will yield more accurate illumination solutions, but will also take much longer to compute. If the solution to the illumination equations converges in fewer steps, the program will simply terminate early, and not perform the additional steps. After the radiosity calculation ends, ray tracing will be used to form an image. This method of using radiosity followed by ray tracing is known as "two-pass" rendering. There are several advantages to the two-pass method:

- The advantages of radiosity are available: accurate diffuse interreflection, color bleeding, and correct illumination from area light sources.
- The advantages of ray tracing are retained: accurate curved surface rendering, sharp shadows (when needed), mirror-like reflections and refractions (when the proper shaders are used).
- With my particular method, the direct illumination is recalculated on the second pass, which results in fewer meshing artifacts and shadows which appear more realistic than most radiosity-only solutions.
- It is possible with my method to calculate indirect specular illumination. This is also known as specular-to-diffuse illumination, and is a current research topic.

If you want to use these features, you will need to make a few slight modifications to your RIB files, outlined below:

- You should set the meshing rates for the patches and elements. See “patchsize,” “elemsize,” and “minsize” in the “nonstandard attributes” section of this document.

- For any nonobvious surfaces, you need to give the average diffuse reflectivity. (Obvious means that the average diffuse reflectivity is the same as the color set by the Color directive.) See the “nonstandard attributes” section of this document for details on setting the average and emissive colors for surfaces. The renderer is smart enough to query shaders for their “Kd” values, so there is no need to premultiply the average color by Kd. However, that’s about as smart as it gets, so don’t expect any tricks done by the surface shader to be somehow divined by the radiosity engine. Any texture mapped objects must also have their average color declared in order to specify the average color of the texture map.

- Specular highlights will be included in the final image, but will not be taken into consideration in the first pass. That’s okay, they provide very little global illumination anyway, with the following exception...

- Mirrors may reflect light directly from a light source to another object in a specular manner. Curved reflectors will tend to either spread the light out or to focus the light to form caustics. These features are supported, provided that you put the following additional light source declaration in your file:

  ```
  LightSource "bouncer"
  ```

  The “bouncer” light source is actually a program which calculates these additional light paths for specular-to-diffuse illumination. In addition, any objects which you wish to act as specular reflectors, regardless of their shader type need to have the following additional attribute declared before the geometry is instanced:

  ```
  Attribute "radiosity" "specularcolor" [ color ]
  ```

When rendering with radiosity, there are two ways to make area light sources. One way is to use the AreaLightSource directive, explicitly making area light sources. The second way is to declare regular geometry, but setting an emission color:

```
Attribute "radiosity" "emissioncolor" [color],
```

The difference is subtle. Both ways will make these patches shoot light into their environment. However, only the geometry declared with AreaLightSource will be resampled again on the second pass. This results in more accurate shadows and nicer illumination, but at the expense of much longer rendering time on the second pass.

```
Attribute "radiosity" "has_caustics" [int],
```

This option must be turned on (passed a nonzero integer) on an object that *receives* caustics. To save computations, BMRT will not compute caustics arriving on surfaces that don't have this attribute turned on (it is off by default).

4.5 Optimizing Rendering Time

Please note that rendering full color frames can take a really long time! High quality rendering, especially ray tracing, is notoriously slow. Try a couple test frames first, to make sure you have everything right before you compute many frames. Multiply the time it takes for each frame by the total number of frames you need. If your total rendering time is prohibitive (say, 5 months), you'd better change something!

Don't bother praying or panicking: I have it on good authority that neither does much to increase rendering throughput. Some optimization hints are listed below. Obvious, effective, easy optimizations are listed first. Trickier or subtler optimizations are listed last.

1. **Resolution**

   Use low resolution when you can. You may want to do test frames at 320 x 240 resolution or lower. Remember that video resolution is only about 640 x 480 pixels. It's pointless to render at higher resolution if you intend to record onto videotape, since any higher resolution will be lost in the scan conversion. Even film can be done at very high quality with about 2048 pixels wide, so don't go wasting time with 4k renders.

2. **Pixel Sampling Rate and Antialiasing**

   Try to specify only 1 sample per pixel for test frames. You can sometimes get away with one sample per pixel for final video frames, too. However, to get really good looking frames you probably need to do higher sampling for antialiasing. There are several sources of aliasing: geometric edges, motion blur, area light shadows, depth of field effects, reflections/refractions, and texture patterns.

   Usually, 2x2 sampling is perfectly adequate to antialias geometric edges for video images. Higher than 3x3 does not usually give noticeable improvements for geometric edges, but you may require even more samples to reduce noise from motion blur and depth of field. There's not much you can do about that if you must using these effects.

   You should prefer using `Attribute "light" "nsamples"` to increase sampling of area lights, rather than increasing `PixelSamples`. Similarly, if the source of your aliasing is blurry reflections or refractions from shaders which use the trace() function, you should consult the documentation for those shaders — many give the option of firing many distributed ray samples, rather than being forced to increase the screen space sampling rate.

Higher sampling rates should never be used to eliminate aliasing in shaders. Well written shaders should be smart enough to analytically antialias themselves by frequency clamping or other trickery. It's considered bad style to write shaders which alias badly enough to require high sampling at the image level.

3. **Geometric Representation**

   Keep your geometry simple, and use curved surface primitives instead of lots of polygons whenever possible. Try writing surface or displacement shaders to add detail to surfaces. It's generally faster to fake the appearance of complexity than it is to create objects with real geometric complexity. Try to make your images interesting through the use of complex textures used on relatively simple geometry.

4. **Lights and Shadows**

   Shadows are important visual cues, but you must use them wisely. Shadowed light sources can really increase rendering time. Only cast shadows from light sources that really need them. If you have several light sources in a scene, you may be able to get away with having only the brightest one cast shadows. Nobody may know the difference!

   Similarly, most objects can be treated as completely opaque (this assumption speeds rendering time). Some objects do not need to cast shadows at all (for example, floors or walls in a room). See the "nonstandard options and attributes" section of this chapter for information on giving the renderer shadowing hints.

5. **Shading Models**

   Keep your shading models simple. Complex procedural textures (such as wood or marble) take much more time to compute than plastic. On the other hand, it is much cheaper to use custom surface or displacement shaders to make surfaces look complex than it is to actually use complex geometry.

   Distribution of rays results in noise. The fewer samples per pixel, the higher the noise. So if you want to keep sampling rates low and reduce noise in the image, you should: avoid using the "blur" parameter in the "shiny" and "glass" surfaces unless you really need it; do not use depth of field if you can get away with a post-processing blur; use nonphysical lights ("pointlight", "distantlight", etc.) instead of physical and area lights.

6. **Tuning Ray Tracing Parameters**

   Several time/quality knobs exist in the ray tracing engine – see the earlier section on nonstandard options and attributes for details. In addition to ensuring that opaque and non-shadow-casting objects are tagged as such, also be sure that your max ray recursion level (`Option "render" "max_raylevel"`) is set as low as possible (the default is 4, but you may be able to get away with as little as 1 or 2 if you don't have much glass or mutual reflection.

4.6 Compatibility Issues

This section details how BMRT differs from the RenderMan Interface 3.1 Specification, as well as any issues related to other renderers.

4.6.1 RenderMan Interface 3.1 Compliance

The *rendrib* renderer is a fairly faithful implementation of the RenderMan Interface 3.1 standard. However, there are a few unimplemented (but hopefully rarely used) features, and a few limitations:

- True displacement of surface points is only partially supported. If you displace in a surface shader, or even in a displacement shader without using the "truedisplacement" attribute, only the surface normals will be purturbed, the points will not move. This usually looks fine as long as the bumps are small. However, if you use the "truedisplacement" attribute, a displacement shader will actually do what you expect and move the surface points.

  True displacements are somewhat limited: (1) it only works for displacement shaders, not surface shaders; (2) it uses *lots* of memory, and also takes more time to render; (3) you cannot use "message passing" between the displacement and surface shaders; (4) you must remember to set displacement bounds; (5) you may get odd self-shadowing of surfaces during radiosity calculations if you use too small a shadow bias.

- Motion Blur is supported, but all motion must be linear (i.e., specified only at the beginning and ending of the motion). Transformations motion blur correctly in all cases, but only some types of geometric primitives will support "vertex blur" (at this time, only NURBS and bicubic patches and meshes). Shading commands simply use the first of the two specifications, and don't blur (i.e., you cannot blur shader parameters).

- The following optional capabilities are not supported: Special Camera Projections, Nonlinear Deformations, Spectral Colors.

- Visible surfaces are always resolved, even if the argument to Hider is "paint."

4.6.2 Supported Modern Extensions

The RI 3.1 spec is quite old, and there is a set of "modern RenderMan" extensions which were informally introduced with more recent versions of *PRMan*, or detailed in SIGGRAPH course notes or the *Advanced RenderMan* book. The following modern extensions are supported by BMRT:

- New RIB "storage classes" for geometric primitive variables `constant` and `vertex`, and new variable types `vector`, `normal`, `matrix`, and `hpoint`. The new "inline declaration" syntax is supported.

- "Locally scoped" functions in Shading Language, `extern` variable declarations (to break to the outer scope), and `void` functions.

- New Shading Language (and RIB) data types `vector`, `normal`, `matrix`, as well as the usual operators for them, plus fixed-length arrays of any of the basic data types.

- New functions in Shading Language: `vtransform()`, `ntransform()`, `ctransform()`, `determinant()`, `translate()`, `rotate()`, `scale()`. `inversesqrt()`, `concat()`, `format()`, `ptlined()`, `filterstep()`, 4-D `noise()`, `cellnoise()`, `pnoise()`, `specularbrdf()`, `min()` and `max()` with multiple arguments and compound arguments, `clamp()` and `mix()` with compound arguments, `spline()` with basis name and/or an array of knots.

- Shader `output` parameters and corresponding "message passing" routines that allow shaders to peek at each other's output variables: `lightsource()`, `surface()`, `displacement()`, `atmosphere()`. Also, the standard light source output variables `__nondiffuse` and `__nonspecular` are respected by the built-in `diffuse()` and `specular()` functions.

- Shading Language routines to ask about renderer state: `attribute()`, `option()`, `texture()`, `rendererinfo()`.

- Support for "light category specifiers" in `illuminance` loops.

- New `Ri` routines: `CoordSysTransform`, `ReadArchive`, `Procedural`.

- Very limited support for new primitives: `Points`, `Curves`, `SubdivisionMesh`, `Blobby`.

4.6.3 Issues with *PRMan*

As I write this, the BMRT and Pixar's PhotoRealistic RenderMan ((R) Pixar) (sometimes called *PRMan*) are the only two widely available implementations of RenderMan. While *rendrib* uses ray tracing and radiosity, *PRMan* uses a scanline method called REYES. Though both renderers should take nearly the same input, the difference in their underlying methods necessarily results in different subsets of the RenderMan standard supported by the two programs. This section lists some of the incompatibilities of the two programs. These differences should not be construed as bugs in either program, but are mostly natural limitations of the two rendering methods. This list is for the user who uses both programs, or wishes to use one program to render output meant for the other.

- *slc* outputs compiled Shading Language as ".slc" files (either interpreted ASCII or DSO's), which are not compatible with Pixar's ".slo" files. The Shading Language source files (".sl") are compatible, and BMRT supports nearly all of the language extensions that PRMan does (as detailed in the previous section).

- The texture mapping and environment mapping routines in *rendrib* take TIFF files directly (either scanline or tiled), and do not read *PRMan*'s proprietary texture format.
- *PRMan* doesn't support true area light sources (but instead places a point light at the current origin), but *rendrib* supports area light sources correctly.
- *rendrib*'s support of true displacement is somewhat more limited than *PRMan*'s, as detailed in the previous subsection.
- *PRMan*'s trace() function always returns 0, and does not support the nonstandard `visibility`, `fulltrace`, `raylevel`, and `isshadowray` functions which *rendrib* implements.
- *PRMan* does not support Imager, Interior, or Exterior shaders. *rendrib* fully supports these kinds of shaders.
- *PRMan* supports several nonstandard primitives which are not fully supported by *rendrib*. `Curves`, `Points`, and `Blobby` primitives are currently ignored by *rendrib*. `SubdivisionMesh` primitives are rendered as control hulls with smoothed normals, but do not currently perform any actual subsivision steps.

4.7 Odds and Ends

There are a bunch of other things you should know about *rendrib* but I coundn't figure out where it went in the manual. In no particular order:

- Before rendering any RIB specified on the command line or piped to it, *rendrib* will first read the contents of the file `$BMRTHOME/.rendribrc`. If there is no environment variable named `$BMRTHOME`, then the file `$HOME/.rendribrc` is read instead. In either case, by putting RIB in one of these places, you can set various options for *rendrib* before any other RIB is read.
- When using the new Shading Language `rendererinfo()` function to query the `"renderer"`, the value returned is ”BMRT”.

Chapter 5

Shaders and Textures

If you are using *rendrib*, you are probably already used to one aspect of working with the RenderMan Interface: specifying scenes. In other words, using the procedural interface (the C library calls) or using RIB (the ASCII archival format).

There is another aspect to using the RenderMan Interface: writing your own shaders. You've already used some of the "standard" shaders such as `"matte"`, `"metal"`, `"plastic"`, and so on. Part of the real power of the RenderMan interface is the ability to write your own shaders to control the appearance of your objects.

Shaders control the appearance of objects. There are several types of shaders:

Surface shaders describe the appearance of surfaces and how they react to the lights that shine on them.

Displacement shaders describe how surfaces wrinkle or bump.

Light shaders describe the directions, amounts, and colors of illumination distributed by a light source in the scene.

Volume shaders describe how light is affected as it passes through a participating medium such as smoke or haze.

Imager shaders describe color transformations made to final pixel values before they are output.

The RenderMan interface specifies that there are several standard shaders available. Standard surface shaders include `"constant"`, `"matte"`, `"metal"`, `"plastic"`, `"shiny"`, and `"paintedplastic"`. Standard light source shaders are `"ambientlight"`, `"distantlight"`, `"pointlight"`, and `"spotlight"`. Standard volume shaders are `"depthcue"` and `"fog"`. The only standard displacement shader is `"bumpy"`, and there are no standard transformation or imager shaders.

5.1 Compiling interpreted shaders with *slc*

Once you have written a shader, save it in a file whose name ends with the extension `.sl`. To compile it, do the following:

```
slc myshader.sl
```

This will result either in a compiled shading language object file called myshader.slc, or you will get error messages. Hopefully, the error message will direct you to the line in your file on which the error occurred, and some clue as to the type of error. *slc* only can compile one .sl file at a time.

The *slc* program takes the following command line arguments:

`-I`*`path`*

Just like a C compiler, the `-I` switch, followed immediately by a directory name (without a space between `-I` and the path), will add that path to the list of directories which will be searched for any files that are requested by any `#include` directives inside your shader source. Multiple directories may be specified by using multiple `-I` switches.

`-D`*`symbol`*
`-D`*`symbol=val`*

Just like a C compiler, the `-D` switch, followed immediately by a symbol name (and possibly with an initial value), will define a proprocessor macro symbol. This allows you to have conditional compilation based on defined symbols using the `#if` and `#ifdef` statements in your shader source code files. The *slc* program automatically defines the symbol `BMRT`.

`-o` *`name`*

Specifies an alternate filename for the resulting .slc file. Without this switch, the output file is derived from the name of your shader.

`-q`

Quiet mode, only reports errors without any chit-chat.

`-v`

Verbose mode, lots of extra chit-chat.

`-x`

Encrypts the resulting .slc file.

`-arch`

Just print out the architecture name (e.g., `sgi_m3`, `linux`, etc.).

`-dso`

On some platforms, this will compile your shader to a machine-code DSO file. See the following section for details.

IMPORTANT NOTE: *slc* uses the C preprocessor (cpp). This executable is usually kept in the /lib directory, so that's where *slc* looks for it. Therefore, if you keep it someplace else (or, like on Windoze, it doesn't normally exist at all), you need to have this directory in your execution PATH or *slc* will not be able to run properly.

Since .sl files are passed through the C preprocessor, you can use the `#include` directive, just as you would for C language source code. You can also give an explicit path for include files using the `-I` command line option to slc (just like you would for the C compiler). You can also use `#ifdef` and other C preprocessor directives in a shader. A variable named BMRT is defined, so you can do something like `#ifdef BMRT`.

The output of *slc* is an ASCII file for a sort of "assembly language" for a virtual machine. When *rendrib* renders your frame and needs a particular shader, this assembly code is read, converted to bytecodes, and interpreted to execute your shader. Because the *slc*'s output is ASCII and is for a virtual machine, it is completely machine-independent. In other words, you can compile your shader on one platform, and use that `.slc` file on any other platform. But, like any other interpreted bytecode, even though BMRT's interpreter is fairly efficient, it is not as efficient as compiled machine code.

5.2 Compiling .sl files to DSO's

On some platforms (specifically Linux, SGI, and Sun), *slc* is also capable of compiling programs to native machine code (by first translating into C++ and then invoking the system's C++ compiler), and dynamically loading the code and executing it directly when the shader is needed by *rendrib*. Some complex shaders can run significantly faster (translating into overall rendering speedups of between 10-50%) if you compile your shaders into DSO's.

You can do this with the `-dso` flag:

```
slc -dso myshader.sl
```

This will create a file called `myshader.`*`ARCH`*`.slc`, where *`ARCH`* is the code name of the platform (such as `linux`, `sgi_m3`, etc.).

There are several very important limitations and caveats to remember when using DSO's:

- The resulting DSO file (the *`ARCH`*`.slc` file) is specific to one platform. If you have a multiplatform environment or wish to distribute the DSO shader to users with different platforms, you will have to recompile the source on that platform.
- Not all shaders will have speed benefits by being compiled into DSO's. Generally, the biggest benefit will be from shaders that have lots of instructions.

Short, inexpensive shaders like `plastic` will render no faster as a DSO than when interpreted. Shaders which are expensive specifically because they have many `noise` or `texture` calls will not speed up much as DSO's, because the time is already being spent within those expensive operations, which are already compiled in the renderer. But some shaders do speed up quite a bit – for example, the `smoke.sl` shader that comes with BMRT runs about twice as fast when compiled into a DSO as when interpreted. If your scene rendering time is dominated by executing complex shaders, you can probably speed up rendering by around 25% by selectively compiling your most expensive shaders as DSO's.

- This feature is relatively new and untested, having first been documented and enabled with BMRT 2.5. Therefore, it's likely that some people will try to compile shaders that *slc* cannot figure out how to translate into C++. In such a case, you will receive error messages that appear to eminate from the C compiler. If this happens, it will be very helpful if you could send the original shader source to `lg@bmrt.org` so that I can fix the compiler.

- It's also possible that the translation to C++ is buggy. If you experience any quirky behavior, you should first delete the compiled .slc file and compile using ordinary *slc*, without using the `-dso` flag. If the shader behavior differs depending on whether or not you use the `-dso` flag, please report the problem (with an example) to `lg@bmrt.org`.

5.3 Using *slctell* to list shader arguments

The *slctell* program reports the type of a shader and its parameter names and default values. Usage is simple: just give the shader name on the command line. For example,

```
slctell plastic
```

reports:

```
surface "shaders/plastic.slc"
    "Ka" "uniform float"
            Default value: 1
    "Kd" "uniform float"
            Default value: 0.5
    "Ks" "uniform float"
            Default value: 0.5
    "roughness" "uniform float"
            Default value: 0.1
    "specularcolor" "uniform color"
            Default value: "rgb" [1 1 1]
```

The *slctell* program should correctly report shader information for both interpreted and compiled DSO shaders. Note, however, that in either case, *slctell* can only report the default values for parameters that are given defaults by simple assignment. In other words, if a constant (or a named space point) is used as the default value, *slctell* will report it correctly, but if the default is the result of a function, complex computation, or involves a graphics state variable, there is no way that *slctell* will correctly report the default value.

5.4 Making tiled TIFF files with *mkmip*

BMRT has always used TIFF files for stored image textures (as opposed to *PRMan*, which requires you to convert to a proprietary texture format). Though BMRT accepts regular scanline (or strip) oriented TIFF files, it is able to perform certain optimizations if the TIFF files you supply happen to be tile-oriented. In particular, BMRT is able to significantly reduce the memory needed for texture mapping with tiled TIFF files.

The *mkmip* program converts scanline TIFF files into multiresolution, tiled TIFF files. The *mkmip* program will also convert zfiles into shadow maps (tiled float TIFFs) and will combine six views into a cube face environment map. Command line usage is:

- **For textures:**

  ```
  mkmip [options] tifffile texturefile
  ```

- **For shadows:**

  ```
  mkmip -shadow [options] zfile shadowfile
  ```

- **For cube-face environment maps:**

  ```
  mkmip -envcube [options] px nx py ny pz nz envfile
  ```

- **For latitude-longitude environment maps:**

  ```
  mkmip -envlatl [options] tifffile envfile
  ```

where options include:

`-smode` *`wrapmode`*
`-tmode` *`wrapmode`*
`-mode` *`wrapmode`*

> where *wrapmode* is one of: `periodic`, `black`, or `clamp`. This specifies the behavior of the texture when outside the [0,1] lookup range. Note that `-smode` and `-tmode` specify wrapping behavior separately for the s and t directions, while `-mode` specifies both at the same time. The default behavior is `black`.

`-resize` *`option`*

Controls the resizing of non-square and non-power-of-two textures when being converted to MIP-maps. The *option* may be any of: `up`, `down`, `round`, `up-`, `down-`, `round-`. The `up`, `down`, and `round` indicates that the texture should be resized to the next highest power of two, the next lowest power of two, of the "nearest" power of two, respectively. For each option, the trailing dash indicates that the texture coordinates should always range from 0 to 1, regardless of the aspect ratio of the original texture. Absence of the dash indicates that the texture should encode its original aspect ratio and adjust the texture coordinates appropriately at texture lookup time. I think that the option that gives the most intuitive use is `up-`. The default is `up`.

`-fov` ***fovangle***

for envcube only, specifies the field of view of the faces.

Note: *rendrib* specifically wants TIFF files as texture and environment maps. The files can be 8, 16, or 32 bits per channel, but cannot be palette color files. Single channel greyscale is okay, as are 3 channel RGB or 4 channel RGBA files. Ordinary scanline TIFF is fine, but if you use the *mkmip* program to pre-process the textures into multiresolution tiled TIFF, your rendering will be much more efficient.

Chapter 6

Miscellaneous Tools

6.1 Writing RIB with *libribout*

You may wish to write a C or C++ program which makes calls to the procedural interface, resulting in the output of RIB. The resulting RIB may be piped directly to another process (such as a previewer), or redirected to a file for later rendering. The library `libribout.a` does this. This library provides a 'C' language binding for the RenderMan Procedural Interface.

The `libribout.a` library has all its public routines use the C language binding, but its implementation contains C++ code, so it is important to either use a C++ compiler to link with it, or else to manually include the standard C++ libraries.

If your program is written in C++, you can link `libribout.a` in the usual way. The following example shows how to link with this library on a typical Unix machine:

```
CC myprog.c -o myprog -lribout -lm
```

If your program is written in ordinary C, then you could compile with C, then link with C++:

```
cc -c myprog.c
CC myprog.o -o myprog -lribout -lm
```

On an SGI, it's apparently important to include `-lC` on the linkage line, to ensure that the C++ standard library is linked properly.

In any case, this will result in an executable, `myprog`, which outputs RIB requests to standard output. This may be redirected to a specific RIB file as follows:

```
myprog > myfile.rib
```

Remember that the `RiBegin` statement usually only takes the argument `RI_NULL`:

```
RiBegin (RI_NULL);
```

The default of sending RIB to stdout can be overridden by providing a filename to the RiBegin statement in your program. For example, suppose your program contains the following statement instead:

```
        RiBegin ("myfile.rib");
```

In this case, the RIB requests corresponding to the Ri procedure calls will be sent to the file `"myfile.rib"` rather than to standard output. In addition, if the filename you specify starts with the '`|`' character, the library will open a *pipe* to the program specified after the '`|`' symbol. For example, `RiBegin ("|rgl");` will cause the RIB you produce to be piped directly to a running *rgl* process without creating an intermediate RIB file.

Remember to tell the C compiler where the `ri.h` and `libribout.a` files are, or it won't be able to find them.

On most platforms, BMRT is also distributed with a *dynamic* library, `libribout.so` (or perhaps `libribout.so.2.3.6`). This library performs the same function as `libribout.so`, but is a dynamic library, which means that it is shared by all programs that link against it, rather than being separately copied into every resulting executable needs the library. Please consult the manual pages to your C compiler, or your local system administrator, for the fine points of using dynamic libraries.

6.2 Parsing Shader Arguments

Pixar's *PhotoRealistic RenderMan* implementation provides a linkable library which allows a developer to read a compiled shader file (.slo) to determine what type of shader it is and what parameter names and defaults belong to that shader. Since Pixar's .slo format is different from BMRT's .slc format, similar functionality is provided to parse the .slc files. The C language header file for these is slc.h. This file should be fairly self-documenting, and certainly anybody with experience using Pixar's libsloargs.a library ought to have an easy time using it.

These routines are all contained in `libribout.a`, so you should link your software against `libribout.a` if you are outputting RIB or parsing shader arguments or both.

However, if you want to parse BMRT shader arguments but use some other RIB client library (such as PRMan's `librib.a`), then there is an additional library you can use, `libslcargs.a`, which contains only the routines for .slc file parsing, but none of the symbols which are also expected to be in a RIB client library.

6.3 Simple Image Compositing with *composite*

Starting with release 2.3.6, BMRT includes a program to perform elementary image compositing operations. If you render your images with alpha channels (i.e. "rgba"), then coverage information will be stored with every pixel in the image. For the purposes of *composite*, RGB images without alpha channels will be assumed to have an alpha of 1.0 at every pixel.

composite may be run as follows:

```
composite file1 over file2 -o output
composite file1 in file2 -o output
composite file1 out file2 -o output
```

`composite` *`file1`* `atop` *`file2`* `-o` *`output`*
`composite` *`file1`* `xor` *`file2`* `-o` *`output`*

> Composite images *file1* and *file2* using one of the standard image compositing operators described in (Porter & Duff, "Digital Image Compositing", Proceedings of SIGGRAPH '84, pp. 253-259), storing the composited image in file *output*.

`composite` *`file1`* `plus` *`file2`* `-o` *`output`*
`composite` *`file1`* `minus` *`file2`* `-o` *`output`*

> Add or subtract two files, storing the results in file *output*. Pixels are clamped to [0,maxval], where maxval==255 for 8 bit images, maxval==65535 for 16 bit images.

`composite` *`file1`* `scale` *`float`* `-o` *`output`*
`composite` *`file1`* `dissolve` *`float`* `-o` *`output`*
`composite` *`file1`* `opaque` *`float`* `-o` *`output`*

> These three unary operators take a floating point number, rather than a filename, as their second operand. They all scale the channels of the image, but in slightly different ways. The `scale` operator multiples the RGB channels, but leaves the alpha alone – i.e. it can brighten or darken an image without changing its transparency. The `dissolve` operator scales the alpha along with the RGB. Finally, the `opaque` operator will scale *only* the alpha channel.

Hint for beginners: you probably want `over`.

6.4 Setting default options and attributes

Remember that both of BMRT's renderers (*rendrib* and *rgl*) read from a file called `.rendribrc` both in the local directory where it is run, and also in your home directory. This file can be plain RIB, which means that if you want to set any defaults (default resolution, shader search path, texture cache size, etc.) you can just put the Option or Attribute lines in this file in your home directory.

6.5 *farm*: Poor Man's Render Farm

Many people ask how they can divide rendering of a single frame among several processors or machines. Starting with BMRT 2.4, the Perl script *farm* accomplishes this task, in a relatively rudimentary way.

6.5.1 How to use *farm*

1. Set the environment variable `BMRT_FARM` to be a blank-separated list of the names of machines which can be used as render servers. Machines with multiple processors should be listed multiple times. For example, if you have a

machine named "fred" with two processors, and one named "wilma" with one processor, then run:

```
setenv BMRT_FARM "fred fred wilma"
```

if you use `csh`. If you use `sh`, try:

```
export BMRT_FARM="fred fred wilma"
```

2. Make sure that *rendrib* is in the default path of each remote machine, and that *mkmosaic* is in the path on the local machine.

3. Run farm: `farm myfile.rib`

6.5.2 What *farm* does

First, *farm* will look at your RIB file to figure out the resolution and the name of the TIFF file that it will render. It will choose an appropriate number of subwindows to render.

One by one, it will send the frame to machines on your `BMRT_FARM` list, using the `-crop` and `-of` flags to make *rendrib* render particular crop windows. Machines whose load averages are too high will automatically refuse the frames.

When *farm* sees that all the subsections are finished (each will leave a little file indicating that it's done), it will assemble all the pieces using the *mkmosaic* program, and clean up all the cruft files.

6.5.3 Important *farm* restrictions

1. Because *farm* relies on *rsh*, you can only use it on UNIX (or UNIX-like) operating systems.

2. You can't use *farm* to render to the display (the -d flag). It must be rendering to a TIFF file.

3. Don't try using any other rendrib command line flags. Request all image options (like radiosity options) in the RIB file with Option and Attribute statements.

4. Hitting Control-C to interrupt *farm* will kill only *farm*, but will leave the individual crop windows rendering on the remote machines. Beware.

Chapter 7

Using BMRT as a "Ray Server" for *PRMan*

This chapter explains how to render scenes using *PRMan* with ray traced shadows and reflections, using BMRT as an "oracle" to provide answers to computations that PRMan cannot solve. We describe a method of actually stitching the two renderers together using a Unix pipe, allowing each renderer to perform the tasks that it is best at.

7.1 Introduction

PhotoRealistic RenderMan has a Shading Language function called trace(), but since there is no ability in PRMan to compute global visibility, the trace() function always returns 0.0 (black). This is no way to ask for any other global visibility information in PRMan. Though PRMan often can fake reflections and shadows with texture mapping, there are limitations:

- Environment mapped reflections are only "correct" from a single point. Environment mapping a large reflective object has errors (which, to be fair, are often very hard to spot). Mutually reflective objects are a big pain in PRMan.
- Environment and shadow maps require multiple rendering passes, and require TD time to set up properly.
- Dealing with shadow maps - selecting resolution, bias, blur, etc. - can be time consuming and still show artifacts in the shadows. Also, shadows cannot motion blur in PRMan, and cannot correctly handle opacity (or color) changes in the object casting a shadow.
- Refraction is nearly impossible to do correctly, since even when environment mapping is acceptable, PRMan cannot tell the direction that a ray exits a refractive object, since the "backside" is not available for ray tracing.

- The Blue Moon Rendering Tools (BMRT) contains a renderer, rendrib, which is fully compliant with the RenderMan 3.1 specification and supports ray tracing, radiosity, area lights, volumes, etc. It can compute ray traced reflections, shadows, and so on, but is much slower than PRMan for geometry which doesn't require these special features.

Both renderers share much of their input - by being RenderMan compliant, they both read the same geometry description (RIB) and shader source code files. (Note: The compatibility is limited to areas dictated by the RMan spec. The two renderers each have different formats for stored texture maps and compiled shaders, and support different feature subsets of the spec.) It's tempting to want to combine the effects of the two renderers, using each for those effects that it achieves well. Several strategies come to mind:

1. Choosing one renderer or the other based on the project, sequence, or shot. Perhaps a strategy might be to use PRMan most of the time, BMRT if you need radiosity or ray tracing.

2. Rendering different objects (or layered elements) with different renderers, then compositing them together to form final frames.

3. Rendering different lighting layers with different renderers, then adding them together. For example, one might render base color with PRMan, but do an "area light pass" (or radiosity, or whatever) in BMRT.

All of these approaches have difficulties (though all have been done). Strategy #1 may force you to choose a slow renderer for everything, just because you need a little ray tracing. There may also be problems matching the exact look from shot to shot, if you are liberally switching between the two renderers. Strategies #2 and #3 have potential problems with "registration," or alignment, of the images computed by the renderers. Also, #3 can be very costly, as it involves renders with each renderer.

The attraction of using the two renderers together, exploiting the respective strengths of both programs while avoiding undue expense, is alluring. I have developed a method of literally stitching the two programs together.

7.2 Background: DSO Shadeops in PRMan

RenderMan Shading Language has always had a rich library of built-in functions (sometimes called "shadeops"), already known to the SL compiler and implemented as part of the runtime shader interpreter in the renderer. This built-in function library included math operations (sin, sqrt, etc.), vector and matrix operations, coordinate transformations, etc. It has also been possible to write SL functions in Shading Language itself (in the case of PRMan, this ability was greatly enhanced with the new compiler included with release 3.7). However, defining functions in SL itself has several limitations.

The newest release of PRMan (3.8) allows you to write new built-in SL functions in 'C' or 'C++'. Writing new shadeops in C and linking them as DSO's has many advantages over writing functions in SL, including:

- The resulting object code from a DSO shadeop is shared among all its uses in a renderer. In contrast, compiled shader function code is inlined every time the function is called, and thus is not shared among its uses, let along among separate shaders that call the same function.
- DSO shadeops are compiled to optimized machine code, whereas shader functions are interpreted at runtime. While PRMan has a very efficient interpreter, it is definitely slower than native machine code.
- DSO shadeops can call library functions from the standard C library or from other third party libraries.
- Whereas functions implemented in SL are restricted to operations and data structures available in the Shading Language, DSO shadeops can do anything you might normally do in a C program. Examples include creating complex data structures or reading external files (other than textures and shadows). For example, implementing an alternative noise() function, which needs a stored table to be efficient, would be exceptionally difficult in SL, but very easy as a DSO shadeop.

DSO shadeops also have several limitations that you should be aware of:

- DSO shadeops only have access to information passed to them as parameters. They have no knowledge of "global" shader variables such as P, parameters to the shader, or any other renderer state. If you need to access global variables or shader parameters or locals, you must pass them as parameters.
- DSO shadeops act as strictly point processes. They possess no knowledge of the topology of the surface, derivatives, or the nature of surface grids (in the case of a REYES renderer like PRMan). If you want to take derivatives, for example, you need to take them in the shader and pass them as parameters to your DSO shadeop.
- DSO shadeops cannot call other builtin shadeops or any other internal entry points to the renderer itself.

Further details about DSO shadeops, including exactly how to write them, are well beyond the scope of these course notes. For more information, please see the RenderMan Toolkit 3.8 User Manual.

7.3 How Much Can We Get Away With?

So PRMan 3.8 has a magic backdoor to the shading system. One thing it's good for is to make certain common operations much faster, by compiling them to machine code. But it also has the ability to allow us to write functions which would not be expressible in SL at all — for example, file I/O, process control or system calls, constructing complex data structures, etc.

How far can we push this idea? Is there some implementation of trace() that we can write as a DSO which will work? Yes! The central idea is to render using PRMan, but implement trace as a call to BMRT. In this sense, we would be using BMRT as an oracle, or a ray server, that could answer the questions that PRMan needs help with, but let PRMan do the rest of the hard work.

BMRT (release 2.3.6 and later) has a ray server mode, triggered by the command line option -rayserver. When in this mode, instead of rendering the frame and writing an image file, BMRT reads the scene file but it just waits for "ray queries" to come over stdin. When such queries (specified by a ray server protocol) are received, BMRT computes the results of the query, and returns the value by sending data over stdout.

The PRMan side is a DSO which, when called, runs rendrib and opens a pipe to its process. Thereafter, calls to the new functions make ray queries over the pipe, then wait for the results.

7.4 New Functionality

This hybrid scheme effectively adds six new functions that you can call from your shaders:

`color trace (point from, vector dir)`

> Traces a ray from position `from` in the direction of vector `dir`. The return value is the incoming light from that direction.

`color visibility (point p1, p2)`

> Forces a visibility (shadow) check between two arbitrary points, retuning the spectral visibility between them. If there is no geometry between the two points, the return value will be (1,1,1). If fully opaque geometry is between the two points, the return value will be (0,0,0). Partially opaque occluders will result in the return of a partial transmission value.
>
> An example use of this function would be to make an explicit shadow check in a light source shader, rather than to mark lights as casting shadows in the RIB stream (as described in the previous section on nonstandard attributes). For example:

```
light
shadowpointlight (float intensity = 1;
                  color lightcolor = 1;
          point from = point "shader" (0,0,0);
                float raytraceshadow = 1;)
{
    illuminate (from) {
            Cl = intensity * lightcolor / (L . L);
            if (raytraceshadow != 0)
                Cl *= visibility (Ps, from);
        }
    }
```

```
float rayhittest (point from, vector dir,
                  output point Ph, output vector Nh)
```

Probes geometry from point `from` looking in direction `dir`. If no geometry is hit by the ray probe, the return value will be very large (1e38). If geometry is encountered, the position and normal of the geometry hit will be stored in `Ph` and `Nh`, respectively, and the return value will be the distance to the geometry.

```
float fulltrace (point pos, vector dir,
                 output color hitcolor, output float hitdist,
                 output point Phit, output vector Nhit,
                 output point Pmiss, output point Rmiss)
```

Traces a ray from `pos` in the direction `dir`.

If any object is hit by the ray, then `hitdist` will be set to the distance of the nearest object hit by the ray, `Phit` and `Nhit` will be set to the position and surface normal of that nearest object at the intersection point, and `hitcolor` will be set to the light color arriving from the ray (just like the return value of `trace`).

If no object is hit by the ray, then hitdist will be set to 1.0e30, hitcolor will bet set to (0,0,0).

In either case, in the course of tracing, if any ray (including subsequent rays traced through glass, for example) ever misses all objects entirely, then `Pmiss` and `Rmiss` will be set to the position and direction of the deepest ray that failed to hit any objects, and the return value of this function will be the depth of the ray which missed. If no ray misses (i.e. some ray eventually hits a nonreflective, nonrefractive object), then the return value of this function will be zero. An example use of this functionality would be to combine ray tracing of near objects with an environment map of far objects.

The code fragment below traces a ray (for example, through glass). If the ray emerging from the far side of the glass misses all objects, it adds in a contribution from an environment map, scaled such that the more layers of glass it went through, the dimmer it will be.

```
missdepth = fulltrace (P, R, C, d, Ph, Nh, Pm, Rm);
if (missdepth > 0)
    C += environment ("foo.env", Rm) / missdepth;
```

```
float isshadowray ()
```

Returns 1 if this shader is being executed in order to evaluate the transparency of a surface for the purpose of a shadow ray. If the shader is instead being evaluated for visible appearance, this function will return 0. This function can be used to alter the behavior of a shader so that it does one thing in the case of visibility rays, something else in the case of shadow rays.

`float raylevel ()`

Returns the level of the ray which caused this shader to be executed. A return value of 0 indicates that this shader is being executed on a camera (eye) ray, 1 that it is the result of a single reflection or refraction, etc. This allows one to customize the behavior of a shader based on how "deep" in the reflection/refraction tree.

7.5 How to use it

Using PRMan as a ray tracer is straightforward:

1. Use these functions in your shaders. In any shader that uses the functions, you should:

   ```
   #include "rayserver.h"
   ```

   If you inspect `rayserver.h` (in the examples directory), you'll see that most the functions described above are really macros. When compiling with BMRT's compiler, the functions are unchanged (all three are actually implemented in BMRT). But when compiling with PRMan's compiler, the macros transform their arguments to world space and call a function called rayserver().

2. Compile the shaders with both BMRT and PRMan's shader compilers. When compiling for PRMan, make sure that the DSO `rayserver.so` (in the BMRT lib directory) is in your include path (-I).

3. Render the file using the `frankenrender` script that comes with BMRT. This is a Perl script that sets up the environment that controls the ray server, and passes the correct arguments to both PRMan and BMRT. Just look at the script for more details on how it works and what arguments are valid.

   If you are rendering the same geometry with both renderers, just use `frankenrender` in the same way as you would use prman or rendrib:

   ```
   frankenrender teapots.rib
   ```

   If you want to give separate RIB files to each renderer, use the -prman and -bmrt flags:

   ```
   frankenrender common.rib -bmrt bmrt.rib -prman prman.rib
   ```

That's it!

7.6 Pros and Cons

The big advantage here is that you can render most of your scene with PRMan, using BMRT for tracing individual rays on selected objects or calculating shadows for selected lights. This is much faster than rendering in BMRT, particularly if you only tell the ray tracer about a subset of the scene that you want in the shadows or reflections. The following effects are utterly trivial to produce with this scheme:

- Ray cast shadows, including shadows that correctly respond to color and opacity of occluding objects. Moving objects can cast correct motion-blurred shadows.
- Correct reflections, including motion blur.
- Real refraction for glass, water, etc.
- No setup time or multi-pass rendering for these effects.

The big disadvantage is that it requires two renderers to both have the scene loaded at the same time. This can be alleviated somewhat by reducing the scene that the ray tracer sees, or by telling the ray tracer to use a significantly reduced tessellation rate, etc. But still, it's a significant memory hit compared to running PRMan alone.

All of the usual considerations about compatibility between the two renderers apply. Be particularly aware of new PRMan primitives and SL features not currently supported by BMRT, texture file format differences, results of noise() functions, etc.

All of the usual considerations about compatibility between the two renderers apply. Be particularly aware of new PRMan primitives and SL features not currently supported by BMRT, texture file format differences, results of noise() functions, etc.

Note that by default, all rays will be traced from the positions at the shutter open time. Thus, reflections and shadows will not be blurred. Indeed, they will strobe in exactly the same way (and for largely the same reasons) as ordinary shadows do with PRMan. If you are ray tracing multipe reflection rays (or multiple shadow rays) per sample, you could try jittering the rays in time. This can be accomplished simply by setting the environment variable `RAYSERVER_JITTER_TIMES` to `1`. Beware, though – this doesn't necessarily make the scene look better, and in some circumstances could make additional artifacts. Try both ways and decide which is best. To turn it off, either don't set that environment variable at all, or set it to `0`.

7.7 Efficiency Tips

Here are several tips to help you speed up the ray server.

- Use the `-prman` and `-bmrt` flags to give separate RIB files to each renderer, eliminating the objects which do not need to be visible in reflections or refractions from the file for *rendrib*. Where this is not possible, at least use `Attribute "render" "visibility"` to make objects invisible in reflections if they are not needed to be seen in reflections (and similarly for shadows).

- Be sure that your max ray recursion level (`Option "render" "max_raylevel"`) is set as low as possible (the default is 4, but you may be able to get away with as little as 1 or 2 if you don't have much glass or mutual reflection.

- It's possible that objects which are only visible in reflections or refractions can be tessellated even more coarsely than usual. Try:

  `Attribute "render" "patch_multiplier" [`*n*`]`}

  The `-rayserver` mode automatically sets n to 0.5, indicating that patches should be diced only half as finely when serving rays as when rendering whole frames. Try reducing n to 0.25 or even lower, to increase speed and decrease memory use. Make n as low as you can get it without seeing visible artifacts.

Bibliography

Apodaca, A. A., editor (1990). *ACM SIGGRAPH '90 Course Notes #18: The RenderMan Interface and Shading Language.*

Apodaca, A. A., editor (1992). *ACM SIGGRAPH '92 Course Notes #21: Writing RenderMan Shaders.*

Apodaca, A. A., editor (1995). *ACM SIGGRAPH '95 Course Notes #4: Using RenderMan in Animation Production.*

Apodaca, A. A. and Gritz, L., editors (1999a). *ACM SIGGRAPH '98 Course Notes #11: Advanced RenderMan: Beyond the Companion.*

Apodaca, A. A. and Gritz, L. (1999b). *Advanced RenderMan: Creating CGI for Motion Pictures.* Morgan-Kaufmann.

Gritz, L. (1993). Computing specular-to-diffuse illumination for two-pass rendering. M.s. thesis, Department of Electrical Engineering and Computer Science, The George Washington University.

Gritz, L. and Apodaca, A. A., editors (1999). *ACM SIGGRAPH '99 Course Notes #25: Advanced RenderMan: Beyond the Companion.*

Gritz, L. and Hahn, J. K. (1996). BMRT: A global illumination implementation of the renderman standard. *Journal of Graphics Tools*, 1(3). ISSN 1086-7651.

Pixar (1989). *The RenderMan Interface, Version 3.1.* Pixar.

Upstill, S. (1990). *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics.* Addison-Wesley.