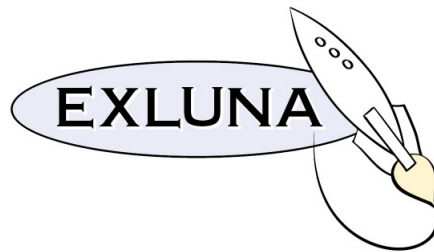


Blue Moon Rendering Tools

User Manual — release 2.6



Exluna, Inc.
1525 Josephine St.
Berkeley, CA 94703

December 1, 2000

Contents

1	Introduction	3
1.1	Reporting Bugs	4
1.2	Copyrights & Trademarks	4
1.3	Licensing Arrangement	5
2	Previewing scene files with <i>rgl</i>	8
2.1	Command Line Options	9
2.1.1	Window Size and Position	9
2.1.2	Drawing Styles	9
2.1.3	File Output Options	10
2.1.4	Animation	10
2.2	Implementation-dependent Options and Attributes	11
2.2.1	Search Paths	11
2.2.2	Drawing Options	12
2.3	Limitations of <i>rgl</i>	12
2.4	Odds and Ends	13
3	Photo-realistic rendering with <i>rendrib</i>	14
3.1	Command Line Options	14
3.1.1	Image Display Options	15
3.1.2	Status Output	16
3.1.3	Radiosity	16
3.1.4	Miscellaneous Options	17
3.2	Implementation-dependent Options and Attributes	17
3.2.1	Rendering Options	18
3.2.2	Search Paths	19
3.2.3	Visibility of Primitives	20
3.2.4	Displacement and Subdivision Attributes	20
3.2.5	Object Appearance	21
3.2.6	Light Source Attributes	22
3.2.7	Finite Element Radiosity Controls	22
3.2.8	Monte Carlo Global Illumination Controls	24
3.2.9	Options for Photon Mapping for Caustics	25
3.2.10	Other Options	26
3.3	Extra Ray Tracing Features	26

3.4	Indirect Illumination	28
3.4.1	Using Finite Element Radiosity	29
3.4.2	Using Monte Carlo Irradiance	30
3.5	Caustics	31
3.6	Optimizing Rendering Time	32
3.7	Compatibility Issues	34
3.7.1	RenderMan Interface Compliance	34
3.7.2	Issues with <i>PRMan</i>	35
3.8	Odds and Ends	35
4	Shaders and Textures	36
4.1	Compiling interpreted shaders with <i>slc</i>	36
4.2	Compiling .sl files to DSO's/DLL's	38
4.3	Using <i>slctell</i> to list shader arguments	39
4.4	Making tiled TIFF files with <i>mkmip</i>	40
5	Miscellaneous Tools	42
5.1	Writing RIB with <i>libribout</i>	42
5.2	Parsing Shader Arguments	43
5.3	<i>iv</i> – an Image Viewer	43
5.3.1	Invoking <i>iv</i> from the command line	43
5.3.2	<i>iv</i> hot keys and mouse commands	44
5.4	Simple Image Compositing with <i>composite</i>	45
5.5	Setting default options and attributes	46
5.6	<i>farm</i> : Poor Man's Render Farm	46
5.6.1	How to use <i>farm</i>	46
5.6.2	What <i>farm</i> does	47
5.6.3	Important <i>farm</i> restrictions	47
6	Using BMRT as a “Ray Server” for <i>PRMan</i>	48
6.1	Introduction	48
6.2	Background: DSO Shadeops in <i>PRMan</i>	49
6.3	How Much Can We Get Away With?	50
6.4	New Functionality	51
6.5	How to use it	53
6.6	Pros and Cons	54
6.7	Efficiency Tips	54
	Bibliography	56

Chapter 1

Introduction

The *Blue Moon Rendering Tools* (BMRT) are a collection of programs that render 3-D scene models.

BMRT uses some APIs that are very similar to those described in the published RenderMan Interface Specification. However, BMRT is not associated with Pixar, and no claims are made that BMRT is in any way a compatible replacement for RenderMan. Those who want a licensed implementation of RenderMan should contact Pixar directly.

Despite these technical/legal terms, you may find that most applications, scene files, and shaders written to conform to the RenderMan Interface can also use BMRT without modification.

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 (due to BMRT's similarities to that published specification). For more detailed information about the RenderMan standard, we recommend *Advanced RenderMan: Creating CGI for Motion Pictures* by Anthony Apodaca and Larry Gritz, *The RenderMan Companion* by Steve Upstill, or the official RenderMan Interface Specification, available from Pixar. All of these texts are fully detailed and clearly written, and no attempt will be made here to duplicate the information in these references.

The parts of BMRT 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 shaders, allowing you to write your own procedures for defining the appearance of surfaces, lights, displacements, volume attenuation, and pixel operations.

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 procedures that result in an archival record that can be rendered at a later time.

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.

1.1 Reporting Bugs

If you come across a bug in the renderer, or if you think you’ve come across a bug in the renderer, please submit a bug report. However, please double check the documentation, both in this file and in `Win32README.html`, if appropriate, to ensure that you’re using the program correctly, before submitting a report. However, if the renderer is dumping core or reporting a Windows application error, you have certainly found a bug, regardless of how you were using the renderer.

The Exluna bug report e-mail address is `bugs@exluna.com`. Please include as much information as possible in bug reports, including:

1. The version of BMRT you’re using (run `rendrib -` to see the version number).
2. The operating system you’re using.
3. The exact error message printed, if any.
4. RIB files, shaders, and a precise description of how to trigger the bug using them.

Thanks for taking the time to report those bugs—it all leads to a better renderer for everyone in the end.

1.2 Copyrights & Trademarks

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

©Copyright 1990-2000 Exluna, Inc. and Larry Gritz. All Rights Reserved.

The TIFF I/O library used by BMRT is: Copyright ©1988-1997 Sam Leffler, Copyright ©1991-1997 Silicon Graphics, Inc. This library may be freely distributed, and is available from: www.libtiff.org

The JPEG I/O library used by BMRT is from the Independent JPEG Group and is copyright ©1991-1998, Thomas G. Lane. The software is available for free from: <http://www.ijg.org/>

On some platforms, the implementations of `rendrib` and `rgl` may make use of the Mesa library, by Brian Paul. This excellent software is available in source form from www.mesa3d.org

RenderMan (R) is a registered trademark of Pixar.
OpenGL is a registered trademark of Silicon Graphics.

1.3 Licensing Arrangement

BMRT release 2.6 is what's known as "freeware." This means that there is no charge to download and run it. However, you may not redistribute it in any way without a written agreement from Exluna, Inc. This software is not in the public domain.

These license terms apply to the BMRT 2.6 software (and minor revisions thereof, e.g. 2.6.1). They do not apply to any other releases and may change significantly and without notice in the future.

EXLUNA, INC.

BLUE MOON RENDERING TOOLS

END-USER SOFTWARE LICENSE AGREEMENT

IMPORTANT - READ BEFORE COPYING, INSTALLING OR USING.

Do not use or load Blue Moon Rendering Tools ("BMRT") and any associated materials (collectively, the "Software") until you have carefully read the following End-User Software License Agreement ("Agreement"). The term "Software" shall also include any third party software incorporated into BMRT and any upgrades, modified versions or updates of the Software licensed to you by Exluna, Inc. ("Exluna"). By loading or using the Software, you agree to the terms of this Agreement. If you do not wish to so agree, do not install or use the Software.

By clicking the "ACCEPT" or "YES" or any other button referenced to this Agreement that suggests you agree and/or by installing, using, or copying this Software, you are becoming a party to, indicating your consent to, and agreeing to be bound by the terms of this Agreement, without modification. If you do not understand and accept all of the following terms and conditions, including those terms and conditions regarding the collection of user information, click the "DO NOT ACCEPT" or "NO" or any other button referenced to this Agreement that suggests you disagree, and you must not install, use, or copy this Software.

1. License. Subject to the terms of this Agreement, Exluna hereby grants you a revocable, non-exclusive, non-transferable license to copy the Software onto a single computer for your personal use and to make one back-up copy of the Software, provided that any and all copies made must contain all of the original and unmodified proprietary notices, including, but not limited to, this License Agreement.

2. Restrictions. You acknowledge and agree that you shall not (a) modify or create any derivative works of the Software or documentation; (b) attempt to disable the Software by any means or in any manner; (c) attempt to decompile, disassemble, reverse engineer, or otherwise attempt to derive the source code for the Software (except to the extent applicable laws specifically prohibit such restriction); (d) redistribute, encumber, sell, rent, lease, sublicense, or otherwise transfer or disclose the Software to any third party; or (e) remove or alter any trademark, logo, copyright or other proprietary notice, legend, symbol or label in the Software.

3. Ownership. All right, title and interest in and to all copies of the Software, including, but not limited to, intellectual property rights, remains with Exluna or its third party suppliers. The Software is copyrighted and protected by the laws of the United States and other countries, and international treaty provisions. Exluna may make changes to the Software, or to items referenced therein, at any time without notice, but is not obligated to support or update the Software. Except as otherwise expressly provided, Exluna grants no express or implied right under Exluna's or any of its third party suppliers' patents, copyrights, trademarks, or other intellectual property rights. You agree that you will take no action that might jeopardize, limit, or interfere in any way with Exluna's or its third party suppliers' ownership or other rights regarding the Software.

4. Disclaimer of Warranty and Limitation of Liability. THE SOFTWARE IS PROVIDED ON AN "AS IS" BASIS. EXLUNA DOES NOT WARRANT THAT OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED, ERROR FREE, OR VIRUS-FREE, OR THAT ANY DEFECT IN THE SOFTWARE WILL BE CORRECTED. EXLUNA EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, AND ANY WARRANTY OF NON-INFRINGEMENT. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS AGREEMENT, AND NO USE OF THE SOFTWARE IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, YOU AGREE THAT IN NO EVENT SHALL EXLUNA BE LIABLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR IN CONNECTION WITH THIS AGREEMENT, EVEN IF EXLUNA HAS BEEN ADVISED OF THE POSSIBILITY THEREOF, AND REGARDLESS OF THE LEGAL OR EQUITABLE THEORY (CONTRACT, TORT OR OTHERWISE) UPON WHICH THE CLAIM IS BASED. YOU ALSO AGREE THAT EXLUNA'S ENTIRE LIABILITY TO YOU OR ANY THIRD PARTY FOR ANY CLAIM OR DEMAND ARISING FROM OR RELATED TO THIS AGREEMENT SHALL NOT EXCEED, IN THE AGGREGATE, THE SUM OF THE FEE YOU PAID FOR THE PRODUCT (IF ANY), WITH THE SOLE EXCEPTION OF DEATH OR PERSONAL INJURY CAUSED BY THE NEGLIGENCE OF EXLUNA, TO THE EXTENT APPLICABLE LAW PROHIBITS THE LIMITATION OF SUCH DAMAGES.

5. Indemnity. You agree to indemnify and hold Exluna, its successors, assigns, subsidiaries, affiliates, officers, directors, agents, and employees harmless from any claim or demand, including reasonable attorneys' fees, made by any third party due to or arising out of your failure to comply with this Agreement or your violation of any law or the rights of any third party.

6. Termination. This Agreement shall be effective unless and until terminated. Exluna may, without prejudice to any other rights under this Agreement or applicable law, terminate the license granted in this Agreement at any time without notice to you if you fail to comply with any of the terms and conditions of this Agreement.

Upon any termination of this Agreement, all rights granted to you under this Agreement shall immediately terminate, and you shall immediately destroy the Software or return all copies of the Software to Exluna.

7. Miscellaneous. (a) This Agreement constitutes the entire agreement between the parties concerning the subject matter hereof; (b) this Agreement may be amended by Exluna at any time upon written notice of the revised terms hereof; (c) this Agreement and any dispute arising out of it shall be governed by the laws of the State of California, USA, excluding its principles of conflicts of law; (d) unless otherwise agreed in writing, all disputes relating to this Agreement (excepting any dispute relating to intellectual property rights) shall be subject to final and binding arbitration in San Francisco, California, conducted by the American Arbitration Association, with the losing party paying all costs of arbitration; (e) the parties hereby consent to the personal jurisdiction of, and agree that any legal proceeding with respect to or arising under this Agreement or necessary to protect the rights or property of that party pending the completion of arbitration will be brought in the state or federal courts sitting in the State of California, County of San Francisco; (f) this Agreement shall not be governed by the United Nations Convention on Contracts for the International Sale of Goods; (g) if any provision in this Agreement should be held illegal or unenforceable by a court having jurisdiction, such provision shall be modified to the extent necessary to render it enforceable without losing its intent or severed from this Agreement if no such modification is possible, and other provisions of this Agreement shall remain in full force and effect; (h) a waiver by either party of any term or condition of this Agreement or any breach thereof, in any one instance, shall not waive such term or condition or any subsequent breach thereof; (i) the provisions of this Agreement that require or contemplate performance after the expiration or termination of this Agreement shall be enforceable notwithstanding said expiration or termination; (j) this Agreement shall be binding upon and shall inure to the benefit of the parties, their successors, and assigns; (k) neither party shall be in default or be liable for any delay, failure in performance (excepting the obligation to pay), or interruption of service resulting directly or indirectly from any cause beyond its reasonable control, and; (l) if any dispute arises under this Agreement, the prevailing party shall be reimbursed by the other party for any and all legal fees and costs associated therewith.

9. US Government Restricted Rights Legend. The Software is provided with "RESTRICTED RIGHTS." Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor. Use of the Software by the Government constitutes acknowledgment of Exluna's proprietary rights therein. Contractor or Manufacturer is Exluna, Inc., 1525 Josephine Street, Berkeley, California 94703. CA1 - 243992.2

Chapter 2

Previewing scene files with *rgl*

Once a scene 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 the scene from standard input (stdin). This allows you to pipe output of a scene-generating process directly to *rgl*. For example, suppose that *myprog* writes to its standard output. Then you could display frames from *myprog* as they are being generated with the following command:

```
myprog | rgl
```

The scene file that 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 frames. The default is to display all of the frames specified in the 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.

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
```

2.1 Command Line Options

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

2.1.1 Window Size and Position

-res *xres yres*

Sets the resolution of the output window. Note that if the scene 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.

2.1.2 Drawing Styles

-lbuffer

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 **-lbuffer** 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 scene 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.

2.1.3 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` command in the file itself, or `ri.tif` if no `Display` command is present in the 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` command, substituting "zfile" for "tif" in the filename.

2.1.4 Animation

`-frames first last`

Sometimes you may only want to preview a subset of frames from a multi-frame 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).

2.2 Implementation-dependent Options and Attributes

Various implementation-specific behaviors of a renderer may be set using two commands: `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.

2.2.1 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 files that are included 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 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.

2.2.2 Drawing Options

Option "limits" "curvethinning" [*frequency*]

Option "limits" "curvethinthreshold" [*thresh*]

When *rgl* draws many **Curves** 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 curve, a value of 100 means that only every 100th curve should be drawn. Furthermore, this thinning is only performed for **Curves** 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.

2.3 Limitations of *rgl*

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

- The following commands 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*.

2.4 Odds and Ends

There are a bunch of other things you should know about *rgl* but we couldn't figure out where they should go in the manual. In no particular order:

- Before rendering any file specified on the command line or commands 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 commands in one of these places, you can set various options for *rgl* before any other file is read.

Chapter 3

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 ray tracing, global illumination, solid modeling, area light sources, texture mapping, environment mapping, displacements, volume and imager shading, and programmable shading.

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 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 file’s Display command).

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

```
myprog | rendrib
```

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

3.1 Command Line Options

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

3.1.1 Image Display Options

-d [*interleave*]

By default, any fully rendered frames are sent to a TIFF image file (unless, of course, the 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 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** command in your 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 file (and like the other options of this section, a **CropWindow** option 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 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.

3.1.2 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 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.

3.1.3 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 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
```

3.1.4 Miscellaneous Options

-frames *first last*

Sometimes you may only want to render a subset of frames from a multi-frame 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 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 scene file for rendering, the image files will have filenames as specified in the 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 `myfile.rib`, 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.).

3.2 Implementation-dependent Options and Attributes

Various implementation-specific behaviors of a renderer can be set using two commands: **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** command. 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.

3.2.1 Rendering Options

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 "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`.

3.2.2 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` files included by `ReadArchive`.
- `texture` texture image files.
- `shader` compiled shaders.
- `procedural` DSO's and executables for `Procedural` calls.
- `display` DSO's for custom 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 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.

3.2.3 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 visible from primary (camera) rays.
- 2 The object is visible from reflection rays.
- 4 The object is visible 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" ["0s"]`

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.

"0s" 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 `"0s"`. 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.

3.2.4 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).

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 99% of scenes, but occasionally 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.

3.2.5 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).

3.2.6 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`.

3.2.7 Finite Element Radiosity Controls

If you are using finite element radiosity (one of the two global illumination methods supported by BMRT), there are some additional options that you can set.

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.

3.2.8 Monte Carlo Global Illumination Controls

In addition to finite element radiosity, whose options are described in the previous subsection, BMRT also supports Monte Carlo-based global illumination calculations. There are a few options related to this technique. Many of the options are related to the fact that it's ridiculously expensive to recompute the indirect illumination at every pixel. So it's only done periodically, and results from the sparse sampling are interpolated or extrapolated. Many options relate to how often it's done. Most of the settings are attributes so that they can be varied on a per-object basis. They are shown here with their default values as examples:

Attribute "indirect" "float maxerror" [0.25]

A maximum error metric. Smaller numbers cause recomputation to happen more often. Larger numbers render faster, but you will see artifacts in the form of obvious "splotches" in the neighborhood of each sample. Values between 0.1-0.25 work reasonably well, but you should experiment. But in any case, this is a fairly straightforward time/quality knob.

Attribute "indirect" "float maxpixeldist" [20]

Forces recomputation based roughly on (raster space) distance. The above line basically says to recompute the indirect illumination when no previous sample is within roughly 20 pixels, even if the estimated error is below the allowable maxerror threshold.

Attribute "indirect" "integer nsamples" [256]

How many rays to cast in order to estimate irradiance, when generating new samples. Larger is less noise, but more time. Should be obvious how this is used. Use as low a number as you can stand the appearance, as rendering time is directly proportional to this.

There are also two options that make it possible to store and re-use indirect lighting computations from previous renderings.

Option "indirect" "string savefile" ["indirect.dat"]

If you specify this option, when rendering is done the contents of the irradiance data cache will be written out to disk in a file with the name you specify. This is useful mainly if the next time you render the scene, you use the following option:

Option "indirect" "string seedfile" ["indirect.dat"]

This option causes the irradiance data cache to start out with all the irradiance data in the file specified. Without this, it starts with nothing and must sample for all values it needs. If you read a data file to start with, it will still sample for points that aren't sufficiently close or have too much error. But it can greatly save computation by using the samples that were computed and saved from the prior run.

3.2.9 Options for Photon Mapping for Caustics

Attribute "caustic" "float maxpixeldist" [16]

Limits the distance (in raster space) over which it will consider caustic information. The larger this number, the fewer total photons will need to be traced, which results in your caustics being calculated faster. The appearance of the caustics will also be smoother. If the maxpixeldist is too large, the caustics will appear too blurry. As the number gets smaller, your caustics will be more finely focused, but may get noisy if you don't use enough total photons.

Attribute "caustic" "integer ngather" [75]

Sets the minimum number of photons to gather in order to estimate the caustic at a point. Increasing this number will give a more accurate caustic, but will be more expensive.

There's also an attribute that can be set per light, to indicate how many photons to trace in order to calculate caustics:

Attribute "light" "integer nphotons" [0]

Sets the number of photons to shoot from this light source in order to calculate caustics. The default is 0, which means that the light does not try to calculate caustic paths. Any nonzero number will turn caustics on for that light, and higher numbers result in more accurate images (but more expensive render times). A good guess to start might be 50,000 photons per light source.

The algorithm for caustics doesn't understand shaders particularly well, so it's important to give it a few hints about which objects actually specularly reflect or refract lights. These are controlled by the following attributes:

Attribute "caustic" "color specularcolor" [0 0 0]

Sets the reflective specularity of subsequent primitives. The default is [0 0 0], which means that the object is not specularly reflective (for the purpose of calculating caustics; it can, of course, still look shiny depending on its surface shader).

Attribute "caustic" "color refractioncolor" [0 0 0]

Attribute "caustic" "float refractionindex" [1]

Sets the refractive specularity and index of refraction for subsequent primitives. The default for **refractioncolor** is [0 0 0], which means that the object is not specularly refractive at all (for the purpose of calculating caustics; it can, of course, still look like it refracts light depending on its surface shader).

3.2.10 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 Mbytes) 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**".

3.3 Extra Ray Tracing Features

The default rendering method used by BMRT is ray tracing. Of course, if you only use standard surfaces and light sources, the results will not be very dramatic. But you can also write shaders that cast reflection and refraction rays. Light sources which cast ray-traced shadows can be added automatically, even from area light sources.

This section describes the Shading Language functions that provide extra support for ray tracing.

`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 be 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.

3.4 Indirect Illumination

Ray tracing will determine illumination only via direct paths from light sources to surfaces being shaded. No knowledge of indirect illumination, or interreflection between objects, is available to the ray tracer. This kind of illumination is responsible for effects such as indirect lighting, soft shadows, and color bleeding. BMRT actually has two different algorithms for computing these kinds of global illumination effects: finite element radiosity, and Monte Carlo irradiance calculations.

Finite element radiosity subdivides all of your geometric primitives into patches, then subdivides the patches into “elements.” In a series of progressive steps, the

patch with the most energy “shoots” its energy at all of the vertices of all of the elements. This distributes the energy around the scene. The more steps you run, and the smaller your patches and elements are, the more accurate your image will be (but the longer it will take to render). The radiosity steps are all computed up front, before the first pixel is actually rendered.

Finite element radiosity has some big drawbacks, almost all of which are related to the fact that it has to pre-mesh the entire scene. First, it gets inaccuracies whenever you use CSG, or have trim curves on your NURBS patches. It can use lots of time and memory when you have many geometric primitives, especially if your objects are made out of lots of little polygons or subdivision meshes. If you use procedural primitives, the renderer will have to expand them all right at the beginning, in order to mesh them. Overall, FE radiosity just doesn’t scale particularly well with large scenes.

The newer Monte Carlo irradiance approach has a different set of tradeoffs. Rather than enmeshing the scene and solving the light transport up front, the MC approach is “pay as you go.” As it’s rendering, when it needs information about indirect illumination, it will do a bunch of extra ray tracing to figure out the irradiance. It will save those irradiance values, and try to reuse them for nearby points. The MC approach works just fine with CSG and trim curves. It doesn’t unpack procedural primitives until they are really needed. It takes longer than FE radiosity for small scenes, but it scales better and should be cheaper for large scenes. As this technique continues to be improved in BMRT, we will probably phase out the FE radiosity.

3.4.1 Using Finite Element Radiosity

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.

Alternately, you could just use **Attribute** “radiosity” “nsteps” as described in the previous section.

When using radiosity, there are a few more things you need to do:

- You should set the meshing rates for the patches and elements. See “patch-size,” “elemsize,” and “minsize” in the “radiosity 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.

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.

3.4.2 Using Monte Carlo Irradiance

To use the Monte Carlo irradiance calculations for global illumination, you need to follow a different set of steps.

1. Don’t use any of the radiosity options or the **-radio** flag. The old radiosity and the new irradiance stuff are not meant to be used together.
2. Add a light source to the scene using the **"indirect"** light shader, which is in the shaders directory of BMRT. This light is built into BMRT, so the shader will not actually be accessed. If there are any objects that you specifically do not want indirect illumination to fall on, you can just use **Illuminate** to turn the light off, and subsequent surfaces won’t get indirect illumination.
3. There are several options that control the behavior of the computations. See section 3.2.8 for their description. You may need to adjust several of them on a per-shot basis, based on time/memory/quality tradeoffs.

There are a few limitations with the irradiance calculations that you should be aware of:

- Right now, it only works for *front lit* objects, and assumes that you’re interested in the side with the outward-pointing normal. Translucent surfaces will be okay eventually, but for now they’re not operational.
- No volumes yet.

- If you compare simple scenes with the old-style BMRT radiosity, you'll find that the radiosity is much faster than the new method. However, for large scenes the new method will most likely win out. Furthermore, the new method can also be used with the ray server, unlike the old.

You shouldn't use the `"seedfile"` option if the objects have moved around. But if the objects are static, either because you have only moved the camera, or because you are rerendering the same frame, the combination of `"seedfile"` and `"savefile"` can *tremendously* speed up computation.

Here's another way they can be used. Say you can't afford to set the other quality options as nice as you would like, because it would take too long to render each frame. So you could render the environment from several typical viewpoints, with only the static objects, and save all the results to a single shared seed file. Then for main frames, always read this seed file (but don't save!) and most of the sampling is already done for you, though it will redo sampling on the objects that move around. Does this make sense?

You can also use the Monte Carlo irradiance global illumination in ray server mode, to serve global illumination to *PRMan*. If you look at `indirect.sl` (which is only used on the *PRMan* side — the light is built into BMRT), you'll see that the light shader simply makes a call to `rayserver_indirect`, and then stashes the results into `C1` so that it looks like an ordinary light, and hence it will work with any existing surface shaders without modifications. Don't forget to compile `indirect.sl` for *PRMan*. Note that *PRMan* doesn't know anything about the indirect options, so you'll see warnings about them. This is perfectly harmless.

3.5 Caustics

BMRT also has the option of computing *caustics*, which (to mangle the true meaning just a bit) refers to light that reflects or refracts from specular objects to focus on other objects. To compute caustics, you must follow these steps:

1. Declare a magic light source with the `"caustic"` shader (like `"indirect"`, it's built into BMRT rather than being an actual shader). You should use `RiIlluminate` to turn the caustic light on for objects that receive caustics, and turn it off for objects that are known to not receive caustics. Illuminating just the objects that are known to receive caustics can save lots of rendering time.
2. For any light sources that should reflect or refract from specular object, thereby causing caustics, you will need to set the number of photons with:

`Attribute "light" "integer nphotons"`

This sets the number of photons to shoot from this light source in order to calculate caustics. The default is 0, which means that the light does not try to calculate caustic paths. Any nonzero number will turn caustics on for that

light, and higher numbers result in more accurate images (but more expensive render times).

3. The algorithm for caustics doesn't understand shaders particularly well, so it's important to give it a few hints about which objects actually specularly reflect or refract lights. This is done with `Attribute "radiosity"`, `Attribute "specularcolor"`, `Attribute "radiosity" "refractioncolor"`, `Attribute "radiosity" "refractionindex"`, See section 3.2.9 for details.
4. Finally, you may want to adjust several global options that control basic time/quality tradeoffs. These are also described in section 3.2.9.

3.6 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: we 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 use 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).

3.7 Compatibility Issues

This section details how BMRT differs from the RenderMan Interface 3.2 Specification, as well as any issues related to other renderers.

3.7.1 RenderMan Interface Compliance

The *rendrib* renderer uses API's that are very similar to the RenderMan Interface 3.2 standard. In fact, you may find that your scenes written to comply with the RenderMan Interface 3.2 standard can be rendered with BMRT without modification. The book *Advanced RenderMan: Creating CGI for Motion Pictures* by Apodaca and Gritz, (Morgan-Kaufmann, 1999) should apply almost totally to BMRT. However, compared to the published RenderMan standard, BMRT has several differences, unimplemented features, and 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 perturbed, 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.

- The following optional capabilities are not supported: Special Camera Projections, Spectral Colors.
- Motion Blur, Depth of Field, and Level of Detail are not supported in BMRT 2.6. Also, the `Blobby` and `Curves` primitives are not currently supported (but they do work fine with *rgl*). We hope to have all of these features working in future releases.

3.7.2 Issues with *PRMan*

Many people use both BMRT and Pixar's PhotoRealistic RenderMan ((R) Pixar) (sometimes called *PRMan*). 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 almost completely compatible
- 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.

3.8 Odds and Ends

There are a bunch of other things you should know about *rendrib* but we couldn't figure out where they should go 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 `renderinfo()` function to query the "`renderer`", the value returned is "BMRT".

Chapter 4

Shaders and Textures

You've probably already used some of the "standard" shaders such as `"matte"`, `"metal"`, `"plastic"`, and so on. Part of the real power of BMRT is the ability to write your own shaders to control the appearance of your 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.

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"`.

4.1 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 preprocessor 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**). On Unix-like operating systems, this executable is usually kept in the **/lib** directory, so that's where *slc* looks for it. If it can't find it there (or, like on Windows, it doesn't normally exist at all), *slc* will look for it in **\$BMRTHOME/bin**. So you will need to set the environment variable **BMRTHOME** to point to the directory in which you have installed BMRT.

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.

4.2 Compiling `.sl` files to DSO’s/DLL’s

`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 (or `-dll` on Windows):

```
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`, `intelnt`, `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 (and only enabled for Windows with BMRT 2.6). 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 emanate from the C compiler. If this happens, it will be very helpful if you could send the original shader source to bugs@exluna.com so that we 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 bugs@exluna.com.

4.3 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.

4.4 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] tiff file 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] tiff file 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. The option that probably 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 5

Miscellaneous Tools

5.1 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` (or `libribout.lib` on Windows) does this. This library provides a ‘C’ language binding for the RenderMan Procedural Interface.

The `libribout` 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` 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.

5.2 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.

5.3 *iv* – an Image Viewer

Once you render images, you need to view them. There are dozens, or possibly hundreds, of programs that can display your ordinary TIFF images that BMRT produces. But probably none of them can display the tiled TIFF images used for textures, environment maps, and shadow maps. Nor can most of them handle 16-bit and floating point images. And even for ordinary images, many image viewers are lacking in certain features that you may find handy. So we have provided *iv*, the Image Viewer tool.

5.3.1 Invoking *iv* from the command line

Invoking *iv* is very simple:

```
iv [options] file1 ... fileN
```

Any number of files may be specified on the command line. Several options may also be specified before the files are listed:

```
-g gamma
```

Sets the gamma correction for subsequent images. The *gamma* parameter is a floating point number, which default to 1.0. Without the *-g* option, the gamma correction will be taken from the *\$GAMMA* environment variable. If no such environment variable exists, no gamma correction will be performed. Note that you can have multiple *-g* options on the command line, interspersed with image file names (this lets you correct different images with different gamma values).

-info

When this flag is used, the name and resolution of each file will be printed to *stdout*.

-sb

Normally, you can use the middle mouse button to “drag” the image around if the image resolution is greater than your display window. If you use the *-sb* command line option, *iv* will also display scroll bars at the edge of the window.

5.3.2 *iv* hot keys and mouse commands

Once you are running *iv* and viewing images, there are several keyboard and mouse commands that you may find useful:

PgUp PgDn

The **PgUp** and **PgDn** keys cycle you to the previous and next images in the list of images.

ENTER

The **ENTER** key will reload the current image from disk.

r g b a c

The **r**, **g**, **b**, and **a** keys will cause *iv* to display just the red, green, blue, or alpha channels of images. The **c** key will display full color again.

f

The **f** key reframes the window. That is, it will readjust the size of the display window to match the resolution of the currently viewed image.

p

The **p** key opens a *pixel view window* that shows you a zoomed view of the pixels surrounding the mouse position, and numeric values for the pixel under the cursor. Hitting **ESC** with the cursor in the pixel view window will close the pixel view window (but not the main window).

q

The `q` key causes *iv* to close its windows and exit.

s

The `s` invokes pixel select mode. In this mode, a single pixel is selected for the pixel view window. The selected pixel no longer follows the mouse cursor, but can be moved with the four arrow keys. Hitting `s` again returns to the usual mouse cursor.

Left-click

Clicking the left mouse button inside the image window zooms in (makes the pixels bigger on screen).

Right-click

Clicking the right mouse button inside the image window zooms out (makes the pixels smaller on screen).

Middle-drag

Moving the mouse with the middle button held down will drag the image around the window, if the image resolution is greater than the window size.

5.4 Simple Image Compositing with *composite*

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 file-name, as their second operand. They all scale the channels of the image, but in slightly different ways. The **scale** operator multiplies 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**.

5.5 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.

5.6 *farm*: Poor Man's Render Farm

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

5.6.1 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`

5.6.2 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.

5.6.3 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 6

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.

6.1 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 largely compatible with the RenderMan 3.2 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, and to a very large extent can read the same geometry description and shader source code files. (Note: The two renderers each have different formats for stored texture maps and compiled shaders, and support different feature sets.) 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. Larry Gritz has developed a method of literally stitching the two programs together.

6.2 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, however, native SL functions have several limitations.

PRMan 3.8 (and later) 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 alone 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.

6.3 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.

6.4 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 be 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.

6.5 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!

6.6 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.

6.7 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, 1990] Apodaca, A. A., editor (1990). *ACM SIGGRAPH '90 Course Notes #18: The RenderMan Interface and Shading Language*.
- [Apodaca, 1992] Apodaca, A. A., editor (1992). *ACM SIGGRAPH '92 Course Notes #21: Writing RenderMan Shaders*.
- [Apodaca, 1995] Apodaca, A. A., editor (1995). *ACM SIGGRAPH '95 Course Notes #4: Using RenderMan in Animation Production*.
- [Apodaca and Gritz, 1999a] Apodaca, A. A. and Gritz, L., editors (1999a). *ACM SIGGRAPH '98 Course Notes #11: Advanced RenderMan: Beyond the Companion*.
- [Apodaca and Gritz, 1999b] Apodaca, A. A. and Gritz, L. (1999b). *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan-Kaufmann.
- [Gritz, 1993] 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 and Apodaca, 1999] Gritz, L. and Apodaca, A. A., editors (1999). *ACM SIGGRAPH '99 Course Notes #25: Advanced RenderMan: Beyond the Companion*.
- [Gritz and Hahn, 1996] 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] Pixar (1989). *The RenderMan Interface, Version 3.1*. Pixar.
- [Upstill, 1990] Upstill, S. (1990). *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley.