Rayshade User's Guide and Reference Manual

Craig E. Kolb

Draft 0.4
January 10, 1992

Contents

Preface

Rayshade is a program for creating ray-traced images. It reads a description of a scene to be rendered and produces a color image corresponding to the description. *Rayshade* was designed to make it easy to create nice pictures. It was also meant to be flexible, easy to modify, and relatively fast.

The first version of *rayshade* was written in 1987-1988 at Princeton University with help and encouragement from David Dobkin and David Hoffman. That version was heavily based on a public-domain "introductory" ray tracer written by Roman Kuchkuda. Changes to *rayshade* from that point until version 4.0 were evolutionary in nature. The current version is to a large extent a re-write, and an attempt has been made to remove some of the fundamental problems present in previous incarnations.

I wish to thank the many people who have made contributions to the development of *rayshade* during the past four years. Thanks to Marc Andreessen, Ray Bellis, Dominique Boisvert, William Bouma, Allen Braunsdorf, Jeff Butterworth, Nick Carriero, Nancy Everson, Tom Friedel, Robert Funchess, David Gelernter, Mike Gigante, Ed Herderick, John Knuston, Raphael Manfredi, Lee Moore, Dietmar Saupe, Brian Wyvill, and the hundreds of others who have provided bug-fixes, suggestions, input files, encouragement, support, and other feedback.

David Dobkin first suggested that an extensible ray tracer would be a worthwhile project. Gavin Bell, David Hoffman, Lefteris Koutsofios, and Steven North were the first users of the original *rayshade*, and their feedback showed that the project might indeed have a future. In the Fall of 1988, Przemyslaw Prusinkiewicz encouraged me to develop *rayshade* further, and was, as always, full of "insanely great" ideas. The resulting version of *rayshade* was released on Usenet in 1989. Allan Snider was particularly

helpful in finding bugs in version 3.0 and in making valuable suggestions as to how the program might be improved.

Rayshade version 4.0 was written by Craig Kolb and Rod Bogart during 1990-1991, with contributions of ideas and code made by many others. Pat Hanrahan's *OOGL* provided the spirit, if not the letter, of the modularity of the version 4.0. Thanks to Pat and to Mark VandeWettering for the "net tracer" conversations and for the inspiration to do something to clean up *rayshade*. Eric Haines saved the day on more than one occasion by suggesting improvements, finding bugs, and saying nice things about *rayshade* when I was all but ready to throw in the towel. Robert Skinner was kind enough to provide the *Noise()*, *DNoise()*, and other texturing functions and to allow them to be redistributed. Mark Podlipec provided the blob object and torus object, which uses Jochen Schwarze's cubic and quartic root-finding functions. Major Thanks to Rod Bogart for being willing to take the plunge and play such a large role in the development of version 4.0. I am most grateful to Benoit Mandelbrot for his support of this project and the inspiration he provided.

C. Kolb
January 10, 1992

Chapter 1

Introduction

This document describes *rayshade* in enough detail to enable the technical-minded to sit down and render some images. In its current form, it is truly a draft, and even then more of a reference manual than a proper user's guide.

This document does not provide any kind of thorough introduction to the basics of computer graphics or ray tracing. There are many other excellent sources for this kind of information. The technical and coding details of *rayshade* and its libraries will be documented elsewhere.

1.1 Getting Started

The best way to learn how to use *rayshade* is to dive right in and start making pictures. Study the example input files that are packaged with *rayshade*. Run them through *rayshade* to see what the images they produce look like. Change the input files; move the camera, change the field of view, modify surface properties, and see what differences your changes make, all the while referring to the appropriate portions of this document. Browse through the individual chapters to see what *rayshade* can and cannot do. The *rayshade* quick reference guide may also help you sort out syntactical nasties.

Throughout this text, the `typewriter` type style is used to indicate keywords and other items that should be passed directly to *rayshade*. Where appropriate, items in an *italic* style indicate places where you should provide an appropriate number or string.

Vectors, which consist of three numerical values, are indicated by an arrow over a name written in italic type style, e.g., $\overrightarrow{vector}$. Items enclosed between `[` and `]` characters indicate that specifying those items is optional. Complex constructions that are described elsewhere in the text, such as surface or object specification, are denoted by enclosing descriptive text between $<$ and $>$ characters.

1.2 A Simple Example

Because *rayshade* provides a default camera description, surface properties, and a default light source, it is easy to construct short input files that allow you to experiment with objects, textures, and transformations. If you haven't already run *rayshade* on one of the example input files, you might want to try producing an image using the following input:

```
sphere 2 0 0 0
```

If you are running *rayshade* on a UNIX[1]-like machine, the command:

```
echo "sphere 2 0 0 0" | rayshade > sphere.rle
```

should produce an image of a sphere.

[1]UNIX is a trademark of AT&T Bell Laboratories

Chapter 2

Running Rayshade

Rayshade can take anywhere from seconds to weeks to render an image. The exact time required is a function of the speed of the machine(s) on which you're working, the complexity of the scene, and how "good" you want the final image to be. [1]

Creating a finished ray-traced image is an iterative process. Usually, many test renderings are made at low resolution and with non-essential features turned off. After each test image is created, surface definitions might be modified, the eye or look positions may be slightly changed, or the intensity of a light source changed.

This chapter describes the basic operation of *rayshade* and some of the options that control that operation. Setting these options properly can greatly reduce rendering time, improve the quality of your images, and make you a better person.

Rayshade usually works as a filter, reading a description from the standard input and writing an image file to the standard output. As it is working, *rayshade* reports on the progress of the rendering by writing messages to the standard error.

[1] Appendix D describes some simple ways to accelerate the rendering process.

Operator	Use	Operation
^	a ^ b	Exponentiation
-	-a	Negation
*	a * b	Multiplication
/	a / b	Division
%	a % b	Remainder
+	a + b	Addition
-	a - b	Subtraction

Table 2.1: Operators, in order of decreasing precedence.

2.1 The Input File

The scene description read by *rayshade* consists of a number of keywords, each followed by a set of arguments. These keywords can be thought of as commands that direct *rayshade* to do various things, such as create objects, set the eye's position, and change an object's appearance.

Many of the keywords have related command-line options for turning on special features and setting values. These options override the values given in the input file, and are explained in detail in Appendix A.

Rayshade is "case sensitive," which means that typing `SPHERE` or `Sphere` instead of `sphere` won't work. *Rayshade* keywords are all lower-case. Many people choose to capitalize the first letter of names that they give to objects or surfaces in order to make then "stand out" in an input file.

Keywords, numbers and strings in the input file are separated by spaces, tabs, or new lines (carriage returns). These "whitespace" characters are handled identically by *rayshade*, which means that you can separate keywords from keywords, key words from arguments, and arguments from arguments using any combination of whitespace characters that you choose.

Numbers may be entered directly as reals or as parenthesized expressions. Reals may be written in exponential notation if you wish, and integers may be written with or without a trailing decimal point. When an integer value is called for and a real is given, the real value is truncated and the resulting integer is used. Table 2.1 lists the available operators available for use in expressions, in order of decreasing precedence.

The upshot of all this is that the strings `42`, `42.`, `(20 + 22)`, and `(7^2 - 7)` mean the same thing to *rayshade*.

Variables may also be defined and used in expressions. Several built-in functions are also provided. See Appendix B for further details.

Rayshade will automatically run the input it receives through the C preprocessor if it is available. This allows you to use C preprocessor directives, such as `#include`, `#define`, and `#ifdef` when designing your input files.

Comments may be included in *rayshade* input files by enclosing text between the strings `/*` and `*/`, as in the C programming language.

2.2 Images

The end result of running *rayshade* is an image file. Depending upon how it was installed, *rayshade* writes images in either the Utah Raster *RLE* format or a generic but easily-manipulated *mtv* format used by Mark VandeWettering in his *mtv* ray tracer. The *mtv* format consists of a header giving the resolution of the image followed by interleaved red-green-blue values for each pixel. The *RLE* format supports an arbitrary number of color channels, an alpha channel, comments, a history field, and the ability to treat images as windows into a larger image. As a result of this flexibility, a number of *rayshade*'s features are not supported if the *mtv* format is being used. You are thus strongly encouraged to obtain a copy of the Utah Raster Toolkit.

If the *mtv* format is used, the image will (and must) consist of three eight-bit color channels. If the *RLE* format is used, the image file consists of three eight-bit color channels plus an eight-bit alpha channel. *Rayshade* also documents in the *RLE* header the command line and *gamma* value used in creating the image.

If more than one frame is rendered, the resulting images are appended in turn to the image file. The various utilities provided by the Utah Raster Toolkit can be used to manipulate the resulting "movie" files. If the Toolkit is not being used, you will probably need to write utility programs to handle the decidedly non-standard multi-image *mtv* format files.

By default, *rayshade* writes the computed image to the standard output. The image file may be written to a file instead by specifying a file name in the input stream.

`outfile` *filename*
Write the computed image to the named file.

The output file name may also be specified on the command line by using the `-O` option.

The size of the output image is measured in pixels. The x size is the number of pixels left-to-right, while the y size is the number of pixels bottom-to-top.

`screen` *xsize ysize*
Use a screen *xsize* pixels wide by *ysize* pixels high.

The screen size may also be set on the command line through the `-R` option. The default screen size is 512 by 512 pixels.

When rendering an image, it is often advantageous to split the image into a number of disjoint windows, each of which is rendered on a different machine. One then combines the images corresponding to the windows into a final image.

`window` *minx maxx miny maxy*
Render the image in the given window.

The window must be properly contained within the screen, i.e., *minx* and *miny* must be greater than or equal to zero, while *maxx* and *maxy* must be less than *xsize* and *ysize*, respectively. The Utah Raster tool *rlecomp* is useful for reconstructing the full image from sub-images. By default, the window is equivalent to the entire screen.

It is also convenient to be able to render a small portion of the window by specifying a subregion using normalized coordinates.

`crop` *left right bottom top*
Crop the rendering window.

The rendering window is cropped by rendering the screen area that falls within $minx + left(maxx - minx)$ and $minx + right(maxx - minx)$ in the X direction, and similarly for the Y direction. *Left* and *bottom* must be greater than or equal to zero. *Right* and *top* must be less than or equal to one. If *left* is greater than *right*, the two values are swapped, and similarly for *bottom* and *top*.

Gamma correction may also be applied to the three output color channels. See Appendix A for more details.

2.3 Statistics Reporting

As it is working, *rayshade* informs you of its progress by writing messages to a "report file". By default, the report file is the standard error. The report itself consists of a number of progress report lines consisting of the number of eye rays traced, the total elapsed time, and the elapsed time since the last progress report. The end of the report contains detailed statistics about intersection tests performed, the number of rays traced, and the like.

`report` [`verbose`] [`quiet`] [*freq*] [*file*]
: Specify the kind of information included in the report and to which file it should be written. If `verbose` is specified, the report will also include a listing of the options selected, the bounding volumes of each aggregate, and the total number of primitives in each aggregate. `quiet` causes warning messages to be suppressed. *freq* specifies the frequency, in scanlines rendered, that progress reports are made. If given, *file* names a file to which the report will be written.

By default, a non-verbose, non-quiet report is written to the standard error once every 10 lines. The `-V` option may also be used to direct the report to a named file.

2.4 Antialiasing

Given a screen of a fixed size, creating an image is accomplished by sampling each pixel one or more times in order to determine what can be seen "through" that pixel by the camera. A pixel thus covers a square area of the image plane, not just a single point.

If a pixel is not sampled at the proper rate, aliasing will result. Aliasing usually appears as "jaggies" or "stair steps" in the image. In order to reduce these and other artifacts, *rayshade* provides an adaptive jittered antialiasing scheme that attempts to detect where increased sampling rates are needed. In jittered sampling, the location at which a sample is taken is perturbed by a random amount. This perturbation reduces aliasing but adds noise to the image. Appendix B describes how jittered time sampling is implemented in *rayshade*.

The adaptive sampling scheme implemented in *rayshade* begins by sampling each pixel on the current scanline once. For each pixel on the scanline, the contrast between it and its four immediate neighbors is computed. If this contrast is greater than a user-specified maximum in any color channel, the pixel and its neighbors are all supersampled by firing an additional $numsamples^2 - 1$ rays through those pixels that have not already been supersampled. This process is repeated for the current scanline until a pass is made without any pixel being supersampled.

`contrast` *redcont greencont bluecont*

Set the maximum allowed contrast between four color samples when adaptive supersampling is used. The contrast test is applied to each color channel separately.

The default maximum contrast values for the red, green, and blue channels are 0.25, 0.2, and 0.4, respectively.

`sample` *n* [`nojitter`]

Use n^2 samples when performing jittered sampling. The maximum legal value is 5. If `nojitter` is specified, sample locations and times will not be jittered.

By default, 3^2 jittered samples are taken.

A given set of sample values must be filtered in order to assign a color to a pixel. Ideally, when performing filtering for a specific pixel, the filter will consider samples from neighboring regions. In *rayshade*, the filtering applied to a pixel makes use of samples taken for that pixel alone. However, one may increase the size of the filter that is applied in order to approximate the results a more robust filtering scheme.

`filter` *type* [*width*]
: Use the indicated filter type with the given width, in pixels. Supported filter types are `gauss` (Gaussian) and `box` (the default).

The default filter width is 1.0 for a box filter, 1.8 for a Gaussian filter. The filter and pixel centers always coincide. When sampling a pixel, samples are taken over the area of the pixel filter, which is not necessarily the same as the area of the pixel itself.

Jittered sampling is used in *rayshade* to sample extended light sources as well. A total of $samples^2$ samples are taken of each extended light source in order to determine the extent of shadowing.

2.5 The Ray Tree

When ray tracing a scene, reflected or transmitted rays may strike other reflective or transparent objects. Further reflected or transmitted rays will be spawned, and so on. Taken together, such a family of rays is termed the *ray tree*. Care must be taken to control the depth of this tree: If it is allowed to grow too deeply, one may spend a great deal of time computing rays that contribute little to the final picture; if it is not allowed to grow far enough, this premature tree pruning may be evident in the image.

Rayshade provides two complementary methods for controlling the depth of the ray tree. One method sets an absolute maximum for the tree. The other allows one to adaptively prune a tree as it grows so that "unimportant" rays are not spawned.

`maxdepth` *level*
: Do not spawn rays deeper than those at the given *level*.

Rays from the eye are of depth zero. The default value for *level* is 15. This depth may also be set from the command line through the `-D` option.

`cutoff` *threshold*

Do not spawn rays whose contribution to the final color of the eye ray is less than *threshold* for each color channel. Threshold may be given as a single floating-point value, or as a red-green-blue triple.

The default value is 0.002. This threshold may also be set from the command line through the `-T` option.

Chapter 3

Specifying a View

When designing a *rayshade* input file, there are two main issues that must be considered. The first and more complex is the selection of the objects to be rendered and the appearances they should be assigned. The second and usually easier issue is the choice of viewing parameters. This chapter deals with the latter problem; the majority of the following chapters discuss aspects of objects and their appearances.

Rayshade uses a camera model to describe the geometric relationship between the objects to be rendered and the image that is produced. This relationship describes a perspective projection from world space onto the image plane.

The geometry of the perspective projection may be thought of as an infinite pyramid, known as the viewing *frustum*. The apex of the frustum is defined by the camera's position, and the main axis of the frustum by a "look" vector. The four sides of the pyramid are differentiated by their relationship to a reference "up" vector from the camera's position.

The image ultimately produced by *rayshade* may then be thought of as the projection of the objects closest to the eye onto a rectangular screen formed by the intersection of the pyramid with a plane orthogonal to the pyramid's axis. The overall shape of the frustum (the lengths of the top and bottom sides compared to left and right) is described by the horizontal and vertical fields of view.

3.1 Camera Position

The three basic camera properties are its position, the direction in which it is pointing, and its orientation. The keywords for specifying these values are described below. The default values are designed to provide a reasonable view of a sphere of radius 2 located at origin. If these default values are used, the origin is projected onto the center of the image plane, with the world x axis running left-to-right, the z axis bottom-to-top, and the y axis going "into" the screen.

`eyep` $\overrightarrow{pos}$
: Place the virtual camera at the given position.

The default camera position is (0, -8, 0).

`lookp` $\overrightarrow{pos}$
: Point the virtual camera toward the given position.

The default look point is the origin (0, 0, 0). The look point and camera position must not be coincident.

`up` $\overrightarrow{direction}$
: The "up" vector from the camera point is set to the given direction.

This up vector need not be orthogonal to the view vector, nor need it be normalized. The default up direction is (0, 0, 1).

Another popular standard viewing geometry, with the x axis running left-to-right, the y axis bottom-to-top, and the z axis pointing out of the screen, may be obtained by setting the up vector to (0, 1, 0) and by placing the camera on the positive z axis.

3.2 Field of View

Another important choice to be made is that of the field of view of the camera. The size of this field describes the angles between the left and right sides and top and bottom sides of the frustum.

`fov` *hfov* [*vfov*]
    Specify the horizontal and vertical field of view, in degrees.

The default horizontal field of view is 45 degrees. If *vfov* is omitted, as is the general practice, the vertical field of view is computed using the horizontal field of view, the output image resolution, and the assumption that a pixel samples a square area. Thus, the values passed via the `screen` keyword define the shape of the final image. If you are displaying on a non-square pixeled device, you must set the vertical field of view to compensate for the "squashing" that will result.

3.3 Depth of Field

Under many circumstances, it is desirable to render objects in the image such that they are in sharp focus on the image plane. This is achieved by using the default "pinhole' camera. In this mode, the camera's aperture is a single point, and all light rays are focused on the image plane.

Alternatively, one may widen the aperture in order to simulate depth of field. In this case, rays are cast from various places on the aperture disk towards a point whose distance from the camera is equal to the *focus distance*. Objects that lay in the focal plane will be in sharp focus. The farther an object is from the image plane, the more out-of-focus it will appear to be. A wider aperture will lead to a greater blurring of objects that do not lay in the focal plane. When using a non-zero aperture radius, it is best to use jittered sampling in order to reduce aliasing.

`aperture` *radius*
    Use an aperture with the given *radius*.

The default radius is zero, resulting in a pinhole camera model.

`focaldist` *distance*
    Set the focal plane to be *distance* units from the camera.

By default, the focal distance is equal to the distance from the camera to the look point.

3.4 Stereo Rendering

Producing a stereo pair is a relatively simple process; rather than simply rendering a single image, one creates two related images which may then be viewed on a stereo monitor, in a stereo slide viewer, or by using colored glasses and an appropriate display filter.

Rayshade facilitates the rendering of stereo pairs by allowing you to specify the distance between the camera positions used in creating the two images. The camera position given in the *rayshade* input file defines the midpoint between the two camera positions used to generate the images. Generally, the remainder of the viewing parameters are kept constant.

`eyesep` *separation*
: Specifies the camera separation to be used in rendering stereo pairs.

There is no default value. The separation may also be specified on the command line through the *-E* option. The view to be rendered (left or right) must be specified on the command line by using the `-l` or `-r` options.

There are several things to keep in mind when generating stereo pairs. Firstly, those objects that lie in from of the focal plane will appear to protrude from the screen when viewed in stereo, while objects farther than the focal plane will recede into the screen. As it is usually easier to look at stereo images that recede into the screen, you will usually want to place the look point closer to the camera than the object of primary interest.

The degree of stereo effect is a function of the camera separation and the distance from the camera to the look point. Too large a separation will result in a hyperstereo effect that will be hard to resolve, while too little a value will result in no stereo effect at all. A separation equal to one tenth the distance from the camera to the look point is often a good choice.

Chapter 4

Light Sources

The lighting in a scene is determined by the number, type, and nature of the light sources defined in the input file. Available light sources range from simple directional sources to more realistic but computationally costly quadrilateral area light sources. Typically, you will want to use point or directional light sources while developing images. When final renderings are made, these simple light sources may be replaced by the more complex ones.

No matter what kind of light source you use, you will need to specify its intensity. In this chapter, an *Intensity* is either a red-green-blue triple indicating the color of the light source, or a single value that is interpreted as the intensity of a "white" light. In the current version of *rayshade*, the intensity of a light does not decrease as one moves farther from it.

If you do not define a light source, *rayshade* will create a directional light source of intensity 1.0 defined by the vector (1., -1., 1.). This default light source is designed to work well when default viewing parameters and surface values are being used.

You may define any number of light sources, but keep in mind that it will require more time to render images that include many light sources. It should also be noted that the light sources themselves will not appear in the image, even if they are placed in frame.

4.1 Light Source Types

The amount of ambient light present in a scene is controlled by a pseudo light source of type *ambient.*

`light` *Intensity* ambient
: Define the amount of ambient light present in the entire scene.

There is only one ambient light source; its default intensity is 1, 1, 1. If more than one ambient light source is defined, only the last instance is used. A surface's ambient color is multiplied by the intensity of the ambient source to give the total ambient light reflected from the surface.

Directional sources are described by a direction alone, and are useful for modeling light sources that are effectively infinitely far away from the objects they illuminate.

`light` *Intensity* `directional` $\overrightarrow{direction}$
: Define a light source with the given intensity that is defined to be in the given direction from every point it illuminates. The direction need not be normalized.

Point sources are defined as a single point in space. They produce shadows with sharp edges and are a good replacement for extended and other computationally expensive light source.

`light` *Intensity* `point` $\overrightarrow{pos}$
: Place a point light source with the given intensity at the given position.

Spotlights are useful for creating dramatic localized lighting effects. They are defined by their position, the direction in which they are pointing, and the width of the beam of light they produce.

`light` *Intensity* `spot` $\overrightarrow{pos}$ $\overrightarrow{to}$ α [θ_{in} θ_{out}]
: Place a spotlight at $\overrightarrow{pos}$, oriented as to be pointing at $\overrightarrow{to}$. The intensity of the light falls off as $(cosine\theta)^{\alpha}$, where θ is the angle between the

spotlight's main axis and the vector from the spotlight to the point being illuminated. θ_{in} and θ_{out} may be used to control the radius of the cone of light produced by the spotlight.

θ_{in} is the the angle at which the light source begins to be attenuated. At θ_{out}, the spotlight intensity is zero. This affords control over how "fuzzy" the edges of the spotlight are. If neither angle is given, they both are effectively set to 180 degrees.

Extended sources are meant to model spherical light sources. Unlike point sources, extended sources actually possess a radius, and as such are capable or producing shadows with fuzzy edges (*penumbrae*). If you do not specifically desire penumbrae in your image, use a point source instead.

`light` *Intensity* `extended` *radius* $\overrightarrow{pos}$

Create an extended light source at the given position and with the given *radius*.

The shadows cast by extended sources are modeled by taking samples of the source at different locations on its surface. When the source is partially hidden from a given point in space, that point is in partial shadow with respect to the extended source, and the sampling process is usually able to determine this fact.

Quadrilateral light sources are computationally more expensive than extended light sources, but are more flexible and produce more realistic results. This is due to the fact that an area source is approximated by a number of point sources whose positions are jittered to reduce aliasing. Because each of these point sources has shading calculations performed individually, area sources may be placed relatively close to the objects it illuminates, and a reasonable image will result.

`light` *Intensity* `area` $\overrightarrow{p1}$ $\overrightarrow{p2}$ *usamp* $\overrightarrow{p3}$ *vsamp*

Create a quadrilateral area light source. The u axis is defined by the vector from $\overrightarrow{p1}$ to $\overrightarrow{p2}$. Along this axis a total of *usamp* samples will be taken. The v axis of the light source is defined by the vector from $\overrightarrow{p1}$ to $\overrightarrow{p3}$. Along this axis a total of *vsamp* samples will be taken.

The values of *usamp* and *vsamp* are usually chosen to be proportional to the lengths of the u and v axes. Choosing a relatively high number of samples will result in a good approximation to a "real" quadrilateral source. However, because complete lighting calculations are performed for each sample, the computational cost is directly proportional to the product of *usamp* and *vsamp*.

4.2 Shadows

In order to determine the color of a point on the surface of any object, it is necessary to determine if that point is in shadow with respect to each defined light source. If the point is totally in shadow with respect to a light source, then the light source makes no contribution to the point's final color.

This shadowing determination is made by tracing rays from the point of intersection to each light source. These "shadow feeler" rays can add substantially to the overall rendering time. This is especially true if extended or area light sources are used. If at any point you wish to disable shadow determination on a global scale, there is a command-line option (`-n`) that allows you to do so. It is also possible to disable the casting of shadows onto given objects through the use of the `noshadow` keyword in surface descriptions. In addition, the `noshadow` keyword may be given following the definition of a light source, causing the light source to cast no shadows onto any surface.

Determining if a point is in shadow with respect to a light source is relatively simple if all the objects in a scene are opaque. In this case, one simply traces a ray from the point to the light source. If the ray hits an object before it reaches the light source, then the point is in shadow.

Shadow determination becomes more complicated if there are one or more objects with non-zero transparency between the point and the light source. Transparent objects may not completely block the light from a source, but merely attenuate it. In such cases, it is necessary to compute the amount of attenuation at each intersection and to continue the shadow ray until it either reaches the light source or until the light is completely attenuated.

By default, *rayshade* computes shadow attenuation by assuming that the index of refraction of the transparent object is the same as that of the medium through which the ray is traveling. To disable partial shadowing due to transparent objects, the `shadowtransp` keyword should be given somewhere in the input file.

`shadowtransp`
: The intensity of light striking a point is *not* affected by intervening transparent objects.

If you enclose an object behind a transparent surface, and you wish the inner object to be illuminated, you must not use the `shadowtransp` keyword or the `-o` option.

Chapter 5

Object Definition

Objects in *rayshade* are composed of relatively simple *primitive* objects. These primitives may be used by themselves, or they may be combined to form more complex objects known as *aggregates.* A special family of aggregate objects, *Constructive Solid Geometry* or CSG objects, are the result of a boolean operations applied to primitive, aggregate, or CSG objects.

This chapter describes objects from a strictly geometric point of view. Later chapters on surfaces, textures, and shading describe how object appearances are defined.

An *instance* is an object that has optionally been transformed and textured. They are the entities that are actually rendered by *rayshade*; when you specify that, for example, a textured sphere is to be rendered, you are said to be instantiating the textured sphere. An instance is specified as a primitive, aggregate, or CSG object that is followed by optional transformation and texturing information. Transformations and textures are described in Chapters 7 and 8 respectively.

5.1 The World Object

Writing a *rayshade* input file is principally a matter of defining a special aggregate object, the World object, which is a list of the objects in the scene. When writing a *rayshade* input file, all objects that are instantiated outside of object-definition blocks are added to the World object; you need not (nor should you) define the World object explicitly in the input file.

5.2 Primitives

Primitive objects are the building box with which other objects are created. Each primitive type has associated with it specialized methods for creation, intersection with a ray, bounding box calculation, surface normal calculation, ray enter/exit classification, and for the computation 2D texture coordinates termed u-v coordinates. This latter method is often referred to as the *inverse mapping* method.

While most of these methods should be of little concern to you, the inverse mapping methods will affect the way in which certain textures are applied to primitives. Inverse mapping is a matter of computing normalized u and v coordinates for a given point on the surface of the primitive. For planar objects, the u and v coordinates of a point are computed by linear interpolation based upon the u and v coordinates assigned to vertices or other known points on the primitive. For non-planar objects, uv computation can be considerably more involved.

This section briefly describes each primitive and the syntax that should be used to create an instance of the primitive. It also describes the inverse mapping method, if any, for each type.

`blob` *thresh st r* $\overrightarrow{p}$ [*st r* $\overrightarrow{p}$ ...]
: Defines a blob with consisting of a threshold equal to *thresh*, and a group of one or more metaballs. Each metaball is defined by its position $\overrightarrow{p}$, radius r, and strength *st*.

The metaballs affect each other according to a superimposed density distribution:

$$F(x, y, z) = \sum_{i=0}^{n} b_i e^{-d_i} - T = 0$$

There is no inverse mapping method for blobs.

`box` $\overrightarrow{corner1}$ $\overrightarrow{corner2}$

Creates an axis-aligned box which has $\overrightarrow{corner1}$ and $\overrightarrow{corner2}$ as opposite corners.

Transformations may be applied to the box if a non-axis-aligned instance is required. There is no inverse mapping method for boxes.

`sphere` *radius* $\overrightarrow{center}$

Creates a sphere with the given *radius* and centered at the given position.

Note that ellipsoids may be created by applying the proper scaling to a sphere. Inverse mapping on the sphere is accomplished by computing the longitude and latitude of the point on the sphere, with the u value corresponding to longitude and v to latitude. On an untransformed sphere, the z axis defines the poles, and the x axis intersects the sphere at $u = 0$, $v = 0.5$. There are degeneracies at the poles: the south pole contains all points of latitude 0., the north all points of latitude 1.

`torus` *rmajor rminor* $\overrightarrow{center}$ $\overrightarrow{up}$

Creates a torus centered at $\overrightarrow{center}$ by rotating a circle with the given minor radius around the center point at a distance equal to the major radius.

In tori inverse mapping, the u value is computed using the angle of rotation about the up vector, and the v value is computing the angle of rotation around the tube, with $v = 0$ occuring on the innermost point of the tube.

`triangle` $\overrightarrow{p1}$ $\overrightarrow{p2}$ $\overrightarrow{p3}$

Creates a triangle with the given vertices.

`triangle` $\overrightarrow{p1}$ $\overrightarrow{n1}$ $\overrightarrow{p2}$ $\overrightarrow{n2}$ $\overrightarrow{p3}$ $\overrightarrow{n3}$

Creates a Phong-shaded triangle with the given vertices and vertex normals.

For both Phong- and flat-shaded triangles, the u axis is the vector from $\overrightarrow{p1}$ to $\overrightarrow{p2}$, and the v axis the vector from $\overrightarrow{p1}$ to $\overrightarrow{p3}$. There is a degeneracy at

$\overrightarrow{p3}$, which contains all points with $v = 1.0$. This default mapping may be modified using the `triangleuv` primitive described below.

`triangleuv` $\overrightarrow{p1}\ \overrightarrow{n1}\ \overrightarrow{uv1}\ \overrightarrow{p2}\ \overrightarrow{n2}\ \overrightarrow{uv2}\ \overrightarrow{p3}\ \overrightarrow{n3}\ \overrightarrow{uv3}$
: Creates a Phong-shaded triangle with the given vertices, vertex normals. When performing texturing, the uv given for each vertex are used instead of the default values.

When computing uv coordinates within the interior of the triangle, linear interpolation of the coordinates associated with each triangle vertex is used.

`poly` $\overrightarrow{p1}\ \overrightarrow{p2}\ \overrightarrow{p3}\ [\overrightarrow{p4}\ \ldots]$
: Creates a polygon with the given vertices. The vertices should be given in counter-clockwise order as one is looking at the "front" side of the polygon. The number of vertices in a polygon is limited only by available memory.

Inverse mapping for arbitrary polygons is problematical. *Rayshade* punts and equates u with the x coordinate of the point of intersection, and v with the y coordinate.

`heightfield` *file*
: Creates a height field defined by the altitude data stored in the named *file*. The height field is based upon perturbations of the unit square in the $z = 0$ plane, and is rendered as a surface tessellated by right isosceles triangles.

See Appendix C for a discussion of the format of a height field file. Height field inverse mapping is straight-forward: u is the x coordinate of the point of intersection, v the y coordinate.

`plane` $\overrightarrow{point}\ \overrightarrow{normal}$
: Creates a plane that passes through the given point and has the specified normal.

Inverse mapping on the plane is identical to polygonal inverse mapping.

`cylinder` *radius* $\overrightarrow{bottom}$ $\overrightarrow{top}$

Creates a cylinder that extends from $\overrightarrow{bottom}$ to $\overrightarrow{top}$ and has the indicated *radius*. Cylinders are rendered *without* endcaps.

The cylinder's axis defines the v axis. The u axis wraps around the cylinder, with $u = 0$ dependent upon the orientation of the cylinder.

`cone` rad_{bottom} $\overrightarrow{bottom}$ rad_{top} $\overrightarrow{top}$

Creats a (truncated) cone that extends from $\overrightarrow{bottom}$ to $\overrightarrow{top}$. The cone will have a radius of rad_{bottom} at $\overrightarrow{bottom}$ and a radius of rad_{top} at $\overrightarrow{top}$. Cones are rendered *without* endcaps.

Cone inverse mapping is analogous to cylinder mapping.

`disc` *radius* $\overrightarrow{pos}$ $\overrightarrow{normal}$

Creates a disc centered at the given position and with the indicated surface normal.

Discs are useful for placing endcaps on cylinders and cones. Inverse mapping for the disc is based on the computation of the normalized polar coordinates of the point of intersection. The normalized radius of the point of intersection is assigned to u, while the normalized angle from a reference vector is assigned to v.

5.3 Aggregate Objects

An aggregate is a collection of primitives, aggregate, and CSG objects. An aggregate, once defined, may be instantiated at will, which means that copies that are optionally transformed and textured may be made. If a scene calls for the presence of many geometrically identical objects, only one such object need be defined; the one defined object may then be instantiated many times.

An aggregate is one of several possible types. These aggregate types are differentiated by the type of ray/aggregate intersection algorithm (often termed an *acceleration technique* or *efficiency scheme*) that is used.

Aggregates are defined by giving a keyword that defines the type of the aggregate, followed by a series of object instantiations and surface definitions, and terminated using the `end` keyword. If a defined object contains no instantiations, a warning message is printed.

The most basic type of aggregate, the *list*, performs intersection testing in the simplest possible way: Each object in the list is tested for intersection with the ray in turn, and the closest intersection is returned.

`list` ... `end`
: Create a List object containing those objects instantiated between the `list`/`end` pair.

The *grid* aggregate divides the region of space it occupies into a number of discrete box-shaped voxels. Each of these voxels contains a list of the objects that intersect the voxel. This discretization makes it possible to restrict the objects tested for intersection to those that are likely to hit the ray, and to test the objects in nearly "closest-first" order.

`grid` *xvox yvox zvox* ... `end`
: Create a Grid objects composed of *xvox* by *yvox* by *zvox* voxels containing those objects instantiated between the `grid`/`end` pair.

It is usually only worthwhile to "engrid" rather large, complex collections of objects. Grids also use a great deal more memory than List objects.

5.4 Constructive Solid Geometry

Constructive Solid Geometry is the process of building solid objects from other solids. The three CSG operators are Union, Intersection, and Difference. Each operator acts upon two objects and produces a single object result. By combining multiple levels of CSG operators, complex objects can be produced from simple primitives.

The union of two objects results in an object that encloses the space occupied by the two given objects. Intersection results in an object that encloses the space where the two given objects overlap. Difference is an order dependent operator; it results in the first given object minus the space where the second intersected the first.

5.4.1 CSG in *Rayshade*

CSG in *rayshade* will generally operate properly when applied to conjunction with on boxes, spheres, tori, and blobs. These primitives are by nature consistent, as they all enclose a portion of space (no hole from the "inside" to the "outside"), have surface normals which point outward (they are not "inside-out"), and do not have any extraneous surfaces.

CSG objects may also be constructed from aggregate objects. These aggregates contain whatever is listed inside, and may therefore be inconsistent. For example, an object which contains a single triangle will not produce correct results in CSG models, because the triangle does not enclose space. However, a collection of four triangles which form a pyramid does enclose space, and if the triangle normals are oriented correctly, the CSG operators should work correctly on the pyramid.

CSG objects are specified by surrounding the objects upon which to operate, as well as any associated surface-binding commands, by the operator verb on one side and the `end` keyword on the other:

`union` <*Object*> <*Object*> [<*Object*> ...] `end`
Specify a new object defined as the union of the given objects.

`difference` <*Object*> <*Object*> [<*Object*> ...] `end`
Specify a new object defined as the difference of the given objects.

`intersect` <*Object*> <*Object*> [<*Object*> ...] `end`
Specify a new object defined as the intersection of the given objects.

Note that the current implementation does not support more that two objects in a CSG list (but it is planned for a future version).

5.4.2 Potential CSG Problems

A consistent CSG model is one which is made up of solid objects with no dangling surfaces. In *rayshade*, it is quite easy to construct inconsistent models, which will usually appear incorrect in the final images. In *rayshade*, CSG is implemented by maintaining the tree structure of the CSG operations. This tree is traversed, and the operators therein applied, on a per-ray basis. It is therefore difficult to verify the consistency of the model "on the fly."

One class of CSG problems occur when surfaces of objects being operated upon coincide. For example, when subtracting a box from another box to make a square cup, the result will be wrong if the tops of the two boxes coincide. To correct this, the inner box should be made slightly taller than the outer box. A related problem that must be avoided occurs when two coincident surfaces are assigned different surface properties.

It may seem that the union operator is unnecessary, since listing two objects together in an aggregate results in an image that appears to be the same. While the result of such a short-cut may appear the same on the exterior, the interior of the resulting object will contain extraneous surfaces. The following examples show this quite clearly.

```
difference
  box -2 0 -3  2 3 3
  union  /* change to list; note bad internal surfaces */
    sphere 2 1 0 0
    sphere 2 -1 0 0
  end
end rotate 1 0 0 -40  rotate 0 0 1 50
```

The visual evidence of an inconsistent CSG object varies depending upon the operator being used. When subtracting a consistent object from and inconsistent one, the resulting object will appear to be the union of the two objects, but the shading will be incorrect. It will appear to be inside-out in places, while correct in other places. The inside-out sections indicate the areas where the problems occur. Such problems are often caused by polygons with incorrectly specified normals, or by surfaces that exactly coincide (which appear as partial "Swiss cheese" objects).

The following example illustrates an attempt to subtract a sphere from a pyramid defined using an incorrectly facing triangle. Note that the resulting image obviously points to which triangle is reversed.

```
name pyramid list
    triangle 1 0 0  0 1 0  0 0 1
```

```
    triangle 1 0 0  0 0 0  0 1 0
    triangle 0 1 0  0 0 0  0 0 1
    triangle 0 0 1  1 0 0  0 0 0  /* wrong order */
end

difference
    object pyramid scale 3 3 3 rotate 0 0 1 45
        rotate 1 0 0 -30 translate 0 -3.5 0
    sphere 2.4 0 0 0
end
```

By default, cylinders and cones do not have end caps, and thus are not consistent primitives. One must usually add endcaps by listing the cylinder or cone with (correctly-oriented) endcap discs in an aggregate.

5.5 Named Objects

A name may be associated with any primitive, aggregate, or CSG object through the use of the `name` keyword:

`name` *objname* <*Instance*>
: Associate *objname* with the given object. The specified object is not actually instantiated; it is only stored under the given name.

An object thus named may then be instantiated (with possible additional transforming and texturing) via the `object` keyword:

`object` *objname* [<Transformations>] [<Textures>]
: Instantiate a copy of the object associated with *objname*. If given, the transformations and textures are composed with any already associated with the object being instantiated.

Chapter 6

Surfaces and Atmospheric Effects

Surfaces are used to control the interaction between light sources and objects. A surface specification consists of information about how the light interacts with both the exterior and interior of an object . For non-closed objects, such as polygons, the “interior” of an object is the “other side” of the object’s surface relative to the origin of a ray.

Rayshade usually ensures that a primitive’s surface normal is pointing towards the origin of the incident ray when performing shading calculations. Exceptions to this rule are transparent primitives, for which *rayshade* uses the direction of the surface normal to determine if the incident ray is entering or exiting the object. All non-transparent primitives will, in effect, be double-sided.

6.1 Surface Description

A surface definition consists of a number of component keywords, each of which is usually followed by either a single number or a red-green-blue color triple. Each of the values in the color triple are normalized, with zero indicating zero intensity, and one indicating full intensity.

If any surface component is left unspecified, its value defaults to zero, with the exception of the index of refraction, which is assigned the default index of refraction (normally 1.0).

Surface descriptions are used in *rayshade* to compute the color of a ray that strikes the surface at a point $\vec{P}$. The normal to the surface at $\vec{P}$, $\vec{N}$, is also computed.

`ambient` $\overrightarrow{color}$
: Use the given *color* to approximate those surface-surface interactions (e.g., diffuse interreflection) not modeled by the ray tracing process.

A surface's ambient color is always applied to a ray. The color applied is computed by multiplying the ambient color by the intensity of the ambient light source.

If $\vec{P}$ is in shadow with respect to a given light source, that light source makes no contribution to the shading of $\vec{P}$.

`diffuse` $\overrightarrow{color}$
: Specifies the diffuse color.

The diffuse contribution from each non-shadowed light source at $\vec{P}$ is equal to the diffuse color of the surface scaled by the cosine of the angle between $\vec{N}$ and the vector from $\vec{P}$ to the light source.

`specular` $\overrightarrow{color}$
: Specifies the base color of specular reflections.

`specpow` *exponent*
: Controls the size of the specular highlight. The larger the *exponent*, the smoother the apparent finish.

The intensity of specular highlights from light sources are scaled by the specular color of the surface.

`reflect` *reflectivity*

Specifies the specular reflectivity of the surface. If non-zero, reflected rays will be spawned.

The intensity of specularly reflected rays will be proportional to the specular color of the surface scaled by the reflectivity.

`transp` *transparency*

Specifies the specular transmissivity of the surface. If non-zero, transmitted (refracted) rays will be spawned.

`body` $\overrightarrow{color}$

Specifies the body color of the object. The body color affects the color of rays that are transmitted through the object.

`extinct` *coefficient*

Specifies the extinction coefficient of the interior of the object.

The extinction coefficient is raised to a power equal to the distance the transmitted ray travels through the object. The overall intensity of specularly transmitted rays will be proportional to this factor multiplied by the surface's body color multiplied by the transparency of the object.

`index` N

Specifies the index of refraction. The default value is equal to the index of refraction of the atmosphere surrounding the eye.

`translucency` *translu* $\overrightarrow{color}$ *stexp*

Specifies the translucency, diffusely transmitted color, and Phong exponent for transmitted specular highlights.

If a light source illuminates a translucent surface from the side opposite that from which a ray approaches, illumination computations are performed, using the given color as the surface's diffuse color, and the given exponent as the Phong highlight exponent. The resulting color is then scaled by the surface's translucency.

6.2 Atmospheric Effects

Any number of atmospheric effects may be associated with the default medium ("air").

`fog` $\overrightarrow{\textit{color}}$ $\overrightarrow{\textit{thinness}}$
: Add exponential fog with the specified *thinness* and *color*.

Fog is simulated by blending the color of the fog with the color of each ray. The amount of fog color blended into a ray color is an exponential function of the distance from the ray origin to the point of intersection divided by the specified *thinness* for each color channel. If the distance is equal to *thinness*, a ray's new color will be half of the fog color plus half its original color.

`mist` $\overrightarrow{\textit{color}}$ $\overrightarrow{\textit{thinness}}$ *zero scale*
: Add global low-altitude mist of the specified color. The color of a ray is modulated by a fog with density that varies linearly with the difference in z coordinate[1] between the ray origin and the point of intersection. The thinness values specify the transmissivity of the fog for each color channel. The base altitude of the mist is given by *zero*, and the apparent height of the mist can be modulated using *scale*, which scales the difference in altitude used to compute the fog.

`fogdeck` *altitude offset* $\overrightarrow{\textit{scale}}$ *chaoscale* $\overrightarrow{\textit{color}}$ $\overrightarrow{\textit{thinness}}$
: Add low-altitude fog, with transmissivity modulated by a chaotic function.

6.3 The Default Medium

The default medium is the medium which surrounds and encompasses all of the objects in the scene; it is the "air" through which eye rays usually travel before hitting an object. The properties of the default medium may be modified through the use of the `atmosphere` keyword.

[1] This all but assumes that the default up vector (0, 0, 1) is being used.

`atmosphere` [*N*] [*atmospheric effects*]

If given, *N* specifies the index of refraction of the default medium. The default is 1.0. Any atmospheric effects listed are applied to rays that are exterior to every object in the scene (e.g., rays emanating from the camera).

```
/*
 * Red sphere on a grey plane, with fog.
 */
eyep 0. -10. 2.
atmosphere fog  .8 .8 .8 14. 14. 14.
plane 0 0 0  0 0 1
sphere diffuse 0.8 0 0   1.5  0 0 1.5
```

6.4 Surface Specification

Rayshade provides a number of ways to define surfaces and to bind these surfaces to objects. The most straight-forward method of surface specification is to simply list the surface properties to be used. Alternatively, one may associate a name with a given surface. This name may subsequently be used to refer to that surface.

`surface` *name* <*Surface Definition*>

Associate the given collection of surface attributes with the given name.

The binding of a collection of surface properties to a given object is accomplished in a bottom-up manner; the surface that "closest" in the modeling tree to the primitive being rendered is the one that is used to give the primitive its appearance.

An object that has no surface bound to it is assigned a default surface that gives the appearance of white plastic.

The most direct way to bind a surface to a primitive is to specify the surface when the the primitive instantiated. This is accomplished by inserting a list of surface attributes or a surface name after the primitive's type keyword and before the actual primitive data.

```
/*
 * A red 'mud' colored sphere reseting on a
 * white sphere. To the right is a sphere with
 * default surface attributes.
 */
surface mud ambient .03 0. 0.  diffuse .7 .3 0.
sphere  ambient .05 .05 .05 diffuse .7 .7 .7   1. 0 0 0
sphere mud  1. 0 0 2
sphere 1. 1.5 0 0
```

Here, we define a red surface named "mud". We then instantiate a sphere, which has a diffuse white surface bound to it. The next line instantiates a sphere with the defined "mud" surface bound to it. The last line instantiates a sphere with no surface bound to it; it is assigned the default surface by *rayshade*.

The applysurf keyword may be used to set the default surface characteristics for the aggregate object currently being defined.

applysurf <*Surface Specification*>
: The specified surface is applied to all following instantiated objects that do not have surfaces associated with them. The scope of this keyword is limited to the aggregate currently being defined.

```
/*
 * Mirrored ball and cylinder sitting on 'default' plane.
 */
surface mirror ambient .01 .01 .01
        diffuse .05 .05 .05
        specular .8 .8 .8 specpow 20  reflect 0.95
plane 0 0 0  0 0 1
applysurf mirror
sphere 1 0 0 0
cylinder 1  3 0 0  3 0 3
```

For convenience, the name cursurf may be used to refer to the current default surface.

The utility of bottom-up binding of surfaces lies in the fact that one may be as adamant or as noncommittal about surface binding as one sees fit when defining objects. For example, one could define a king chess piece consisting of triangles that have no surface bound to them, save for the cross on top, which has a gold-colored surface associated with it. One may then instantiate the king twice, once applying a black surface, and once applying a white surface. The result: a black king and a white king, each adorned with a golden cross.

```
surface white ...
surface black ...
surface gold  ...
...
define cross
        box x y z  x y z
        ...
defend
define king
        triangle x y z  x y z  x y z
        ...
        object gold cross
defend

object white king translate 1. 0 0
object black king
```

Chapter 7

Transformations

Rayshade supports the application of linear transformations to objects and textures. If more than one transformation is specified, the total resulting transformation is computed and applied.

`translate` $\overrightarrow{delta}$
: Translate (move) by *delta*.

`rotate` $\overrightarrow{axis}$ θ
: Rotate counter-clockwise about the given axis by θ degrees.

`scale` $\overrightarrow{v}$
: Scale by *v*.

All three scaling components must be non-zero, else degenerate matrices will result.

`transform` $\overrightarrow{row1}$ $\overrightarrow{row2}$ $\overrightarrow{row3}$ $[\overrightarrow{delta}]$
: Apply the given 3-by-3 transformation matrix. If given, *delta* specifies a translation vector.

Transformations should be specified in the order in which they are to be applied immediately following the item to be transformed. For example:

```
/*
 * Ellipsoid, rotated cube
 */
sphere 1. 0 0 0   scale 2. 1. 1. translate 0 0 -2.5
box 0 0 0 .5 .5 .5
   rotate 0 0 1 45 rotate 1 0 0 45 translate 0 0 2.5
```

Transformations may also be applied to textures:

```
plane 0 0 -4  0 0 1
  texture checker red scale 2 2 2 rotate 0 0 1 45
```

Note that transformation parameters may be specified using animated expressions, causing the transformations themselves to be animated. See Appendix B for further details.

Chapter 8

Texturing

Textures are used to modify the appearance of an object through the use of procedural functions. A texture may modify any surface characteristic, such as diffuse color, reflectivity, or transparency, or it may modify the surface normal ("bump mapping") in order to give the appearance of a rough surface.

Any number of textures may be associated with an object. If more than one texture is specified, they are applied in the order given. This allows one to compose texturing functions and create, for example a tiled marble ground plane using the *checker* and *marble* textures.

Textures are associated with objects by following the object specification by a number of lines of the form:

`texture` *name* <*Texturing Arguments*> [*Transformations*]

Transformations may be applied to the texture in order to, for example, shrink or grow feature size, change the orientation of features, and change the position of features.

Several of the texturing functions take the name of a colormap as an argument. A colormap is 256-line ASCII file, with each line containing three space-separated values ranging from 0 to 255. Each line gives the red, green, and blue values for a single entry in the colormap.

8.1 Texturing Functions

`blotch` *BlendFactor surface*

Produces a mildly interesting blotchy-looking surface. *BlendFactor* is used to control the interpolation between the default surface characteristics and the characteristics of the given surface. A value of 0 results in a roughly 50-50 mix of the two surfaces. Higher values result in a great portion of the default surface characteristics.

`bump` *scale*

Apply a random bump map. The point of intersection is passed to *DNoise()*. The returned normalized vector is weighted by *scale* and the result is added to the normal vector at the point of intersection.

Using an image texture applied to the "bump" component offers a more direct way to control the modification of surface normals (see below).

`checker` *<Surface>*

Applies a 3D checkerboard texture. Every point that falls within an "even" unit cube will be assigned the characteristics of the named surface applied to it, while points that fall within "odd" cubes will have its usual surface characteristics. Be wary of strange effects due to roundoff error that occur when a planar checkered surface lies in a plane of constant integral value (e.g., $z = 0$) in texture space. In such cases, simply translate the texture to ensure that the planar surface is not coincident with an integral plane in texture space (e.g., `translate 0 0 0.1`).

`cloud` *scale H* λ *octaves cthresh lthresh tscale*

This texture is a variant on Geoff Gardner's ellipsoid-texturing algorithm. It should be applied to unit spheres centered at the origin. These spheres may, of course, be transformed at will to form the appropriately-shaped cloud or tree.

A sample of normalized *fBm* (see the *fbm* texture) is generated at the point of intersection. This sample is used to modulate the surface transparency. The final transparency if a function of the sample value, the the proximity of the point of intersection to the edge of the sphere (as seen from the ray origin), and three parameters to control the overall "density." The proximity of the point to the sphere edge is determined by evaluating a *limb* function, which varies from 0 on the limb to 1 at the center of the sphere.

$$transp = 1. - \frac{fBm - cthresh - (lthresh - cthresh)(1 - limb)}{tscale}$$

`fbm` *offset scale* H λ *octaves thresh* [*colormap*]

Generate a sample of discretized fractional Brownian motion (fBm) and uses it to scale the diffuse and ambient component of an object's surface. *Scale* is used to scale the value returned by the fBm function. *Offset* allows one to control the minimum value of the fBm function. H is the *Holder exponent* used in the fBm function (a value of 0.5 works well). λ is used to control *lacunarity*, and specifies the the frequency difference between successive samples of the fBm basis function (a value of 2.0 will suffice). *Octaves* specifies the number of octaves (samples) to take of the fBm basis function (in this case, Noise()). Between five and seven octaves usually works well. *Thresh* is used to specify a lower bound onthe output of the fBm function. Any value lower than *thresh* is set to zero.

If a *colormap* is named, a 256-entry colormap is read from the named file, and the sample of fBm is scaled by 255 and is used as an index into the colormap. The resulting colormap entry is used to scale the ambient and diffuse components of the object's surface.

`fbmbump` *offset scale* H λ *octaves*

Similar to the *fbm* texture. Rather than modifying the color of a surface, this texture acts as a bump map.

`gloss` *glossiness*

Gives reflective surfaces a glossy appearance. This texture perturbs the object's surface normal such that the normal "samples" a cone of unit height with radius $1. - glossiness$. A value of 1 results in perfect mirror-like reflections, while a value of 0 results in extremely fuzzy reflections. For best results, jittered sampling should be used to render scenes that make use of this texture.

`marble` [*colormap*]

Gives a surface a marble-like appearance. The texture is implemented as roughly parallel alternating veins of marble, each of which is separated by 1/7 of a unit and runs perpendicular to the Z axis. If a colormap is named, the surface's ambient and diffuse colors will be scaled using the RGB values in the colormap. If no colormap is given, the diffuse and ambient components are simply scaled by the value of the marble function. One may transform the texture to control the density and orientation of the marble veins.

`sky` *scale H* λ *octaves cthresh ltresh*

Similar to the *fbm* texture. Rather than modifying the color of a surface, this texture modulates its transparency. *cthresh* is the value of the *fBm* function above which the surface is totally opaque. Below *lthresh*, the surface is totally transparent.

`stripe` <*Surface*> *size bump* <Mapping>

Apply a "raised" stripe pattern to the surface. The surface properties used to color the stripe are those of the given surface. The width of the stripe, as compared to the unit interval, is given by *size*. The magnitude of *bump* controls the extent to which the bump appears to be displaced from the rest of the surface. If negative, the stripe will appear to sink into the surface; if positive, it will appear to stand out of the surface.

Mapping functions are described below.

`wood`

Gives a surface a wood-like appearance. The feature size of this texture is approximately 0.01 of a unit, making it often necessary to scale the texture in order to achieve the desired appearance.

8.2 Image Texturing

Rayshade also supports an `image` texture. This texture allows you to use images to modify the characteristics of a surface. You can use three-channel images to modify the any or all of the ambient, diffuse, and specular colors of a surface. If you are using the Utah Raster Toolkit, you can also use single-channel images to modify surface reflectance, transparency, and the specular exponent. You can also use a single-channel image to apply a bump map to a surface.

In all but the bump-mapping case, a component is modified by multiplying the given value by the value computed by the texturing function. When using the Utah Raster Toolkit, surface characteristics are modified in proportion to the value of the *alpha* channel in the image. If there is no *alpha* channel, or you are not using the Utah Raster Toolkit, *alpha* is assumed to be everywhere equal to 1.

`component` <*Component*>
: The named component will be modified.

Possible surface components are: `ambient` (modify ambient color), `diffuse` (modify diffuse color), `specular` (modify specular color), `specpow`, (modify specular exponent), `reflect`, (modify reflectivity), `transp` (modify transparency), `bump`, (modify surface normal). The `specpow`, `reflect`, `transp`, and `bump` components require the use of a single-channel image.

`range` *high low*
: Specify the range of values to which the values in the image should be mapped. An value of 1 will be mapped *high*, 0 to *low*. Intermediate values will be linearly interpolated.

`smooth`
: When given, pixel averaging will be performed in order to smooth the sampled image. If not specified, no averaging will occur.

`textsurf` <*Surface Specification*>
: For use when modifying surface colors, this keyword specifies that the given surface should be used as the base to be modified when the *alpha* value in the image is non-zero. When *alpha* is zero, the object's unmodified default surface characteristics are retained.

The usual behavior is for the object's default surface properties to be used.

`tile` *un vn*

: Specify how the image should be tiled (repeated) along the u and v axes. If positive, the value of *un* gives the number of times the image should be repeated along the u axis, starting from the origin of the texture, and positive *vn* gives the number of times it should be repeated along the v axis. If either value is zero, the image is repeated infinitely along the appropriate axis.

Tiling is usually only a concern when planar mapping is being used, though it may also be used if image textures are being scaled. By default *un* and *vn* are both zero.

A mapping function may also be associated with an image texture.

8.3 Mapping Functions

Mapping functions are used to apply two-dimensional textures to surfaces. Each mapping functions defines a different method of transforming a three dimensional point of intersection to a two dimensional $u - v$ pair termed texturing coordinates. Typically, the arguments to a mapping method define a center of a projection and two non-parallel axes that define a local coordinate system.

The default mapping method is termed $u - v$ mapping or *inverse mapping*. Normally, there is a different inverse mapping method for each primitive type (see chapter 5). When inverse mapping is used, the point of intersection is passed to the *uv* method for the primitive that was hit.

`map uv`

: Use the *uv* (inverse mapping) method associated with the object that was intersected in order to map from 3D to determine texturing coordinates.

The inverse mapping method for each primitive is described in Chapter 5.

`map planar` $[\overrightarrow{origin}\ \overrightarrow{vaxis}\ \overrightarrow{uaxis}]$

: Use a planar mapping method. The 2D texture is transformed so that its u axis is given by $\overrightarrow{uaxis}$ and its v axis by $vaxis$. The texture is projected along the vector defined by the cross product of the u and v axes, with the (0,0) in texture space mapped to $\overrightarrow{origin}$.

`map cylindrical` $[\overrightarrow{origin}\ \overrightarrow{vaxis}\ \overrightarrow{uaxis}]$

Use a cylindrical mapping method. The point of intersection is projected onto an imaginary cylinder, and the location of the projected point is used to determine the texture coordinates. If given, $\overrightarrow{origin}$ and $\overrightarrow{vaxis}$ define the cylinder's axis, and $\overrightarrow{uaxis}$ defines where $u = 0$ is located.

See the description of the inverse mapping method for the cylinder in Chapter 5. By default, the point of intersection is projected onto a cylinder that runs through the origin along the z axis, with $\overrightarrow{uaxis}$ equal to the x axis.

`map spherical` $[\overrightarrow{origin}\ \overrightarrow{vaxis}\ \overrightarrow{uaxis}]$

Use a spherical mapping method. The intersection point is projected onto an imaginary sphere, and the location of the projected point is used to determine the texturing coordinates in a manner identical to that used in the inverse mapping method for the sphere primitive. If given, the center of the projection is $\overrightarrow{origin}$, $\overrightarrow{vaxis}$ defines the sphere axis, and the point where the non-parallel $\overrightarrow{uaxis}$ intersects the sphere defines where $u = 0$ is located.

By default, a spherical mapping projects points towards the origin, with $\overrightarrow{vaxis}$ defined to be the z axis and $\overrightarrow{uaxis}$ defined to be the x axis.

Appendix A

Options

This appendix describes the command-line arguments accepted by *rayshade*. These options override defaults as well as any values or flags given in the input file, and are thus useful for generating test and other unusual, "non-standard" renderings.

The general form of a *rayshade* command line is:

`rayshade` [*Options*] [*filename*]

If given, the input file is read from *filename*. By default, the input file is read from the standard input. Recall that, by default, the image file is written to the standard output; you will need to redirect the standard output if you have not chosen to write the image to a file directly. The name of the input file may be given anywhere on the command line.

Command-line options fall into two broad categories: those that set numerical or other values and thus must be followed by further arguments, and those that simply turn features on and off. *Rayshade*'s convention is to denote the value-setting arguments using capital letters, and feature-toggling arguments using lower-case letters.

`-A` *frame*
: Begin rendering (action) on the given frame.

The default starting frame is number zero.

`-a`

Toggle writing of alpha channel.

This option is only available when the Utah Raster Toolkit is being used.

`-C` *R G B*

Set the adaptive ray tree pruning color. If all channel contributions falls below the given cutoff values, no further rays are spawned.

Overrides the value specified via the `cutoff` keyword.

`-c`

Continue an interrupted rendering.

When given, this option indicates that the image file being written to contains a partially-completed image. *Rayshade* will read the image to determine the scanline from which to continue the rendering. This option is only available with the Utah Raster Toolkit. The `-O` option must also be used.

`-D` *depth*

Set maximum ray tree depth.

Overrides the value specified in the input file via the `maxdepth` keyword.

`-E` *separation*

Set eye separation for rendering of stereo pairs.

Overrides the value specified via the `eyesep` keyword.

`-e`

Write exponential RLE file.

This option is only available for use with the Utah Raster Toolkit. See the Utah Raster Toolkit's *unexp* manual page for details on exponential RLE files.

`-F` *freq*

Set frequency of status report.

Overrides the value given using the `report` keyword.

`-f`

Flip all computed polygon (and triangle) normals.

This option should be used when rendering polygons defined by vertices given in *clockwise* order, rather than counter-clockwise order as expected by *rayshade*.

`-G` *gamma*

Use given gamma correction exponent writing writing color information to the image file.

The default value for *gamma* is 1.0.

`-g`

Use a Gaussian pixel filter.

Overrides the filter selected through the use of the `filter` keyword.

`-h`

Print a short use message.

`-j`

Toggle the use of jittered sampling to perform antialiasing. If disabled, a fixed sampling pattern is used.

`-l`

Render the left stereo pair image.

`-m`

Write a sampling map to the alpha channel.

Rather than containing coverage information, the alpha channel values will be restricted to zero, indicating no supersampling, and full intensity, indicating supersampling. This option is only available if the Utah Raster Toolkit is being used.

`-N` *frames*
: Set the total number of frames to be rendered.

This option overrides any value specified through the use of the `frames` keyword. By default, a single frame is rendered.

`-n`
: Do not render shadows.

`-O` *outfile*
: Write the image to the named file.

This option overrides the name given with the `outfile` keyword, if any, in the input file.

`-o`
: Toggle the effect of object opacity on shadows.

This option is equivalent to specifying `shadowtransp` in the input file. By default, *rayshade* traces shadow rays through non-opaque objects.

`-P` *cpp-arguments*
: Specify the options that should be passed to the C preprocessor.

The C preprocessor, if available, is applied to all of the input passed to *rayshade*.

`-p`
: Perform preview-quality rendering.

This option is equivalent to `-n -S 1 -D 0`.

`-q`

Do not print warning messages.

`-R` *xsize ysize*

Produce an image *xsize* pixels wide by *ysize* pixels high.

This option overrides any screen size set by use of the `screen` keyword.

`-r`

Render the right stereo pair image.

`-S` *samples*

Use $samples^2$ jittered samples.

This option overrides any value set through the use of the `samples` keyword in the input file.

`-s`

Disable caching of shadowing information.

It should never be necessary to use this option.

`-T` *r g b*

Set the contrast threshold in the three color channels for use in adaptive supersampling.

This option overrides any value given through the use of the *contrast* keyword.

`-u`

Toggle the use of the C preprocessor.

Rayshade usually feeds its input through a C preprocessor if one is available on your system. If this option is given, unadulterated input files will be used.

`-V` *filename*
: Write verbose output to the named file.

This option overrides any file named through the use of the `report` keyword.

`-v`
: Write verbose output.

When given, this option causes information about the options selected and the objects defined to be included in the report file.

`-W` *minx maxx miny maxy*
: Render the specified window.

The window must be properly contained within the screen. This option overrides any window specified using the *window* keyword in the input file.

`-X` *left right bottom top*
: Crop the rendering window using the given normalized values.

This option is provided to facilitate changing and/or examining a small portion of an image without having to re-render the entire image.

Appendix B

Animation

Rayshade provides basic animation animation support by allowing time-varying transformations to be associated with primitives and aggregate objects. Commands are provided for controlling the amount of time between each frame, the speed of the camera shutter, and the total number of frames to be rendered.

By default, *rayshade* renders a single frame, with the shutter open for an instant (0 units of time, in fact). The shutter speed in no way changes the light-gathering properties of the camera, i.e., frames rendered using a longer exposure will not appear brighter than those with a shorter exposure. The only change will be in the potential amount of movement that the frame "sees" during the time that the shutter is open.

Each ray cast by *rayshade* samples a particular moment in time. The time value assigned to a ray ranges from the starting time of the current frame to the starting time plus the amount of time the shutter is open. When a ray encounters an object or texture that possesses an animated transformation, the transformed entity is moved into whatever position is appropriate for the ray's current time value before intersection, shading, or texturing computations are performed.

The starting time of the current frame is computed using the length of each frame the current frame number, and the starting time of the first frame.

`shutter` *t*

> Specifies that the shutter is open for t units of time for each exposure.

A larger value of t will lead to more motion blur in the final image. Note that t may be greater than the actual length of a frame. By default, t is zero, which prevents all motion blur.

`framelength` *frameinc*

> Specifies the time increment between frames.

The default time between frames is 1 unit.

`starttime` *time*

> Specifies the starting time of the first frame.

By default, *time* is zero.

Variables may be defined thorugh the use of the `define` keyword:

`define` *name value*

> Associate *name* with the given *value.* Value may be a constant or a parenthesized expression.

The variable *name* may thereafter be used in expressions in the input file.

An animated transformation is one for which animated expressions have been used to define one or more of its parameters (e.g. the angle through which a rotation occurs). An animated expression is one that makes use of a time-varying ("animated") variable or function.

There are two supported animated variables. The first, `time`, is equal to the current time. When a ray encounters an animated transformation defined using an expression containing `time`, the ray substitutes its time value into the expression before evaluation. Using the `time` variable in an animated expression is the most basic way to create blur-causing motion.

The second animated variable, `frame`, is equal to the current frame number. Unlike the `time` variable, `frame` takes on a single value for the duration of each frame. Thus, transforms animated through the use of the `frame` variable will not exhibit motion blurring.

Also supported is the `linear` function. This function uses `time` implicitly to interplate between two values.

`linear (` *Stime, Sval, Etime, Eval* `)`
: Linearly interpolate between *Sval* at time *Stime* and *Eval* at time *Etime*. If the current time is less than *Stime*, the function returns *Sval*. If the current time is greater than *Etime*, *Eval* is returned.

The following example shows the use of the `time` variable to animate a sphere by translating it downwards over five frames. Note thet the `shutter` keyword is used to set the shutter duration in order to induce motion blurring.

```
frames 5
shutter 1
sphere 1 0 0 2 translate 0 0 (-time)
```

Further examples of animation may be found in the Examples directory of the *rayshade* distribution.

Appendix C

Height Field Files

This appendix describes the format of the files that store data for the height field primitive. The format is an historical relic; a better format is needed.

Height field data is stored in binary form. The first record in the file is a 32-bit integer giving the square root of number of data points in the file. We'll call this number the size of the height field.

The size is followed by altitude (z) values stored as 32-bit floating point values. The 0th value in the file specifies the z coordinate of the lower-left corner of the height field (0, 0). The next specifies the Z coordinate for $(1/(size-1), 0)$. The last specifies the coordinate for $(1., 1.)$. In other words, the i^{th} value in the heightfield file specifies the z coordinate for the point whose x coordinate is $(i\%size)/(size-1)$, and whose y coordinate is $(i/size)/(size-1)$. Non-square height fields may be rendered by specifying altitude values less than or equal to the magic value -1000. Triangles that have any vertex less than or equal in altitude to this value are not rendered.

While this file format is compact, it sacrifices portability for ease of use. While creating and handling height field files is simple, transporting a height field from one machine to another is problematical due to the fact that differences in byte order and floating-point format between machines is not taken into account.

These problems could be circumvented by writing the height field file in a fixed-point format, taking care to write the bytes that encode a given value in a consistent way from machine to machine. An even better idea would be to write a set of tools for manipulating arbitrary 2D arrays of floating-point values in a compact, portable way, allowing for comments and the like in the file...