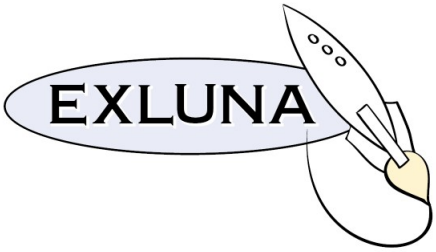
Entropy 3.1

Technical Reference



Exluna, Inc.

1900 Addison St., Suite 200
Berkeley, CA 94704

June 6, 2002

Contents

Chapter 1

Introduction

1.1 Welcome to Entropy

Entropy is a high-quality rendering system that incorporates the speed and low memory use of "scanline" renderers and the realism afforded by "global illumination" renderers. Entropy:

- employs an efficient "scanline" algorithm that can render very large scenes using much less time and memory than other rendering algorithms, and uses an "analytic" algorithm for hidden surface removal, providing superior quality antialiasing, especially for small or thin geometry such as hair;
- supports ray traced reflections and shadows, true area light sources, indirect "global illumination," and caustics;
- supports a wide variety of geometric primitives, including polygons and polygonal meshes, bilinear and bicubic patches and patch meshes, NURBS, subdivision surfaces, quadrics (such as spheres, cylinders, etc.), "blobby" implicit surfaces, 1D spline primitives (ideal for hair), and point primitives (ideal for particle systems);
- allows fully-programmable shading — users can supply "shaders" that specify in minute detail the appearance of surfaces, emission of light sources, attenuation of light through volumes, and modification of the shape of surfaces (displacement);
- produces high-quality images of RGB, alpha, depth, and arbitrary data computed by shaders, with no hard-coded resolution limits, and supporting 8-bit, 16-bit, and floating point output.
- allows user-programmable plugins for new image output types, procedural geometry, and calling C/C++ routines from inside shaders.

1.2 Compatibility, Ancestry, Versioning

Entropy uses some APIs that are very similar to those described in the RenderMan Interface 3.2 Specification published by Pixar Animation Studios. However, Entropy is not

associated with Pixar, and no claims are made that Entropy is in any way a compatible replacement for their PhotoRealistic RenderMan product. Despite this, you may find that most applications, scene files, and shaders written to conform to the RenderMan Interface may be used with Entropy without modification.

Entropy is "descended" from the Blue Moon Rendering Tools (BMRT), a shareware ray tracer that has been used on films such as *A Bug's Life*, *Stuart Little*, and *Hollow Man*. Entropy's graphics algorithms are new, its performance is vastly superior to BMRT, and it is fully supported as a production-ready product. Nonetheless, its heritage and compatibility with its ancestor are obvious, and also are reflected by its version numbering.

Entropy's version numbers are organized as *major.minor patch* (for example, 3.1 R1). The major release indicates a significant leap forward, possibly in incompatible ways. A minor release may include new functionality or enhanced performance, but will preserve backward-compatibility. A patch generally fixes bugs but does not add new features (except for features that, because of bugs, were nonfunctional). The major and minor version numbers of Entropy reflect the relationship and compatibility with BMRT releases (that's why the initial release of Entropy was 3.0, to show its relation to BMRT 2.x).

1.3 About This Manual

Organization

A thorough description of Entropy's use would cover scores of topics, all of which are interrelated. There is no way to arrange each chapter to require only knowledge of the preceeding sections, so we will not pretend to present the material in a linear fashion. Instead, this manual is organized into three main sections. You should not feel the need to read the chapters in order.

Part I presents the formal specifications of the two major API's: scene description files and shaders. This section is organized as a reference work, not as a tutorial. Beginning users may not need it right away, but as they become intermediate users they will have more frequent need to understand, and occasionally modify, scene files and shaders. Advanced users who routinely write their own shaders or debug scene files will find this section indispensable as the primary reference on those languages.

Part II contains the nitty-gritty details of how to actually use Entropy: invoking the renderer, compiling shaders, dealing with texture maps, writing moderately complex shaders, using ray tracing and global illumination, and so on. The material is presented in an expository, "how-to" format, organized by topic.

Part III is aimed squarely on the most advanced users and developers of modelers and third-party add-ons to Entropy. As a casual user, some knowledge of the scene description and shader API's of Part I will suffice. But Part III dscribes ways to extend the functionality of Entropy in even more advanced (and esoteric) ways. It describes procedural geometry, DSO/DLL's callable from your shaders, and image display servers. If you're not a software developer, you almost certainly don't need the material in Part III — don't feel bad about skipping it.

Margin Notes

NEW!

Deprecated

This manual has easy-to-spot margin notes (as seen here) indicating new features, commands, or language syntax that is present in Entropy but was not available in BMRT (or other compatible renderers), or whose meaning or performance has changed dramatically.

There are also some commands or language features that Entropy supports primarily for backward compatibility with scene files designed for use with BMRT or *PRMan*, but that we feel should be phased out and replaced with newer features or idioms. It's possible that someday support for these features may be eliminated altogether.[1] We have marked those commands as *deprecated* and have made it easy to spot such commands with margin notes, as done here.

[1] We promise that features will only disappear entirely with rarity, and only after the feature has been deprecated in previous releases.

Part I

API Specifications

Chapter 2

Scene Files

This chapter describes the formatting and commands for Entropy's input scene files. The scene files are plain ASCII, although if they are compressed by `gzip` Entropy will correctly uncompress them while reading the files.

Typically, scene files will be generated by a modeling system or an appropriate plugin or converter. Beginning users will generally not need to see the scene files themselves, and may remain blissfully unaware of their details. More intermediate and advanced users will find it useful to understand the format of these files for debugging purposes, clever modifications, and more fine control of renderer operations.

2.1 Lexical Elements

Entropy scene files are comprised of ASCII characters.

Whitespace includes blank spaces, tab characters, newlines, and carriage returns. Whitespace can delimit tokens, but the amount and type of whitespace has no syntactic or semantic significance.

The character '`#`' (when it is not inside a quoted string) indicates a comment. The `#` and all characters following it, until a newline character, are ignored by the renderer.

Numbers are integers or floating-point values. Integers consist of an optional sign ('`+`' or '`-`') followed by one or more decimal digits. Floating-point numbers consist of an optional sign, zero or more decimal digits, a decimal point ('`.`'), zero or more decimal digits, and an optional exponent (the letter '`e`' followed by one or more decimal digits).

Integers may be used for data that are expected to be floating-point (with the obvious conversion performed), but it is an error to use a floating-point number where an integer is expected.

Strings are delimited by double quotes (' `"` ') at their beginning and ending. As in C programs, strings may contain special escape sequences that begin with the backslash ('`\`') character: '`\n`' (newline), '`\r`' (carriage return), '`\t`' (tab), '`\b`' (backspace), '`\\`' (backslash character), '`\"`' (double quote character).

Names start with a letter ('`a`'-'`z`' or '`A`'-'`Z`') followed by one or more letters, digits, or the underscore ('`_`'). Names are case-sensitive; that is, upper-case and lower-case letters are different and refer to different variables or commands.

Arrays of data (strings or numbers) are delimited by square bracket characters ('[' and ']'). Strings and numbers may not be mixed in a single array. If an array of numbers has any floating-point elements, all elements will be assumed to be floating-point. An array of numbers with all integer values may be used as either an integer or a floating-point array.

2.2 Overall structure of scene files

Entropy scene files have a particular structure, embodied in the tree of the hierarchical graphics state. There are actually several calls that manipulate the hierarchical graphics state stack, some of which change the mode of the graphics state and others that push a subset of the attributes onto the stack. We describe the subtree of the hierarchy inside of a balanced pair of these commands as being a *block*. Being "in" a particular block means that one of the ancestors of that level of the hierarchy was created by that particular API call.

The scene description is divided into two phases: describing the *viewer*, and describing the *world*. The viewer description includes various parameters of the camera as well as parameters of the image file that is being generated. These parameters are called *options* and are global to the entire rendered image. The world description includes all of the geometry in the scene, with their material descriptions and other parameters that can be specific to individual objects in the scene. These parameters are called *attributes*. The division between describing "the viewer" and describing "the scene" occurs at `WorldBegin`, which starts a *world block*.

Rendering an animation can include the rendering of large numbers of frames, and each frame might require prerendering a certain number of images for use as texture maps or for other purposes. The scene description API allows for these logical divisions of the work to be specified, as well.

A scene description therefore proceeds as a series of nested blocks, in the following order:

```
FrameBegin
        Image and camera options
        WorldBegin
                Attributes, lights, primitives
        WorldEnd
FrameEnd
...
(optionally more frames)
```

These blocks are divided by the following commands:

FrameBegin *framenumber*
FrameEnd

In a scene file that contains the definitions of multiple frames, the individual frames are delimited by `FrameBegin` and `FrameEnd`. `FrameBegin`, which takes a single integer *framenumber* that isn't actually used for anything, saves the entire set of frame options. `FrameEnd` restores the options.

It is perfectly legal for a scene file to not have `FrameBegin` or `FrameEnd` statements, as long as it contains the definition of only one frame.

In a scene file that describes multiple frames, we strongly recommend that *no* commands be placed outside of `FrameBegin`/ `FrameEnd` pairs.

EXAMPLE:

```
FrameBegin 1
FrameEnd
```

WorldBegin
WorldEnd

At `WorldBegin`, the current transformation is marked as `"world"` space, the current transformation is reset to the identity (and further transformations are relative to `"world"` space, rather than `"camera"` space), and all the options (see Section 3.1) are fixed for the frame, and the entire attribute state is saved (just like with `AttributeBegin`).

Between `WorldBegin` and `WorldEnd`, the scene geometry (and the attributes that apply to the primitives) are set. No options may be set between `WorldBegin` and `WorldEnd`.

At `WorldEnd`, the scene is assumed to be complete, the entire frame is rendered, and the attribute state is restored to its values at `WorldBegin`.

EXAMPLE:

```
WorldBegin
WorldEnd
```

2.2.1 Types of scene commands

Broadly speaking, there are three types of commands in scene files. The first group of commands set *options*, that is, the properties that apply to the camera or the image as a whole. Examples include image resolution and camera placement. These commands all happen prior to `WorldBegin`. These are all covered in detail in Chapter 3.

The second group of commands set *attributes*, which are the properties that apply to individual geometric objects, and which may vary from object to object. Examples include object color and shader assignments. Included in the attributes is the current transformation (sometimes called the CTM), which defines the placement of objects. The renderer maintains a *graphics state* — the set of all attribute values and the current transformation. As attribute or transformation commands are executed in order, the graphics state is modified. Commands exist to save and restore the transformation or the entire attribute state.

The third group of commands add geometry to the scene. When a geometric primitive is added to the scene, that primitive gets a copy of the full graphics state – the current transformation and set of attribute values. We say that the primitive is *bound* to the attributes at that time. As it continues to be processed to make the image, the primitive will continue to remember the attribute values that were in effect when the primitive was created. Later

changes to the current transformation or attributes will only have an effect on subsequently-created primitives; once a primitive is declared, it cannot be changed by future alterations of the attribute state.

2.3 Parameter Lists and Declarations

Many routines — including all geometric primitives and shader declarations, and a number of other routines such as `Display`, `Option`, and `Attribute`— take a variable number of arguments as a way of extending their functionality to incorporate user-specified data. These routines typically take several fixed arguments, followed by a variable-length parameter list. The declarations of these routines in Chapters 3–4 indicate these optional arguments as: *...parameterlist...*.

These parameter lists are comprised of *token-value pairs* — alternating parameters of strings (tokens) supplying the name of a parameter, and arrays supplying the parameter's data value(s). The array may have only one element, if a single data value is needed.

Because the purpose of these parameter lists is to communicate user-designated data (for example, to pass it for use in shaders), the renderer will not know the nature of the data in advance. Therefore, the parameter name itself can have the type information built in, known as "in-line parameter declaration." The full format of a parameter name on a parameter list is:

"*class type name*"	for ordinary variables, and
"*class type name*[*length*]"	for arrays

The *name* is the name of the parameter. The *type* is one of: `float`, `color`, `point`, `vector`, `normal`, `matrix`, or `string` (the shading language types described in Section 5.2), and two additional types: `hpoint`, or `integer`. An `hpoint` is a 4-D homogeneous point (x, y, z, w) that is interpolated in 4-D and converted to 3-D by a homogeneous divide prior to providing their value to shaders. The meaning of `integer` should be obvious, but it is worth nothing that many parameters to `Option`, `Attribute`, and `Display` are integers, but shader parameters may not be `integer` (because there is no integer type in the shading language). Parameters may also be arrays of any of the basic types, as indicated by including the optional *length* in brackets.

The optional *class* is not used when passing data to shaders or to routines such as `Option`, `Attribute`, or `Display`. But when data is being passed on geometric primitives, the *class* may be one of: `constant`, `uniform`, `varying`, or `vertex`. The storage class indicates how much data is passed and how it should be interpolated across the primitive (this is explained in further detail in Chapter 4).

Following are some examples of token-value lists:

```
Displacement "dentmap" "string texturename" ["mydents.tx"]
Surface "planks" "float Kd" [0.75] "color darkwood" [.75 .5 .1]
Atmosphere "foggy" "float weights[5]" [.2 .95 .1 .1 1.25]
Patch "bilinear" "P" [0 0 0 .5 -.25 0 1 0 0 .5 2 0]
        "constant color paint" [.25 .25 .33]
        "vertex float specularity" [.5 .8 .6 .73]
```

Alternately, you can pass parameters without the full in-line declarations, relying on a global dictionary of name-type definitions. The `Declare` routine adds a name-type definition to the dictionary:

Declare *name declaration*

Adds a new parameter name to the global name-type dictionary. The parameter *name* will be used as the identifier in subsequent parameter lists. The parameter *type* defines the storage class and data type of the data. The syntax is similar to, but not identical to, variable declarations in the shading language — "*class type*".

class can be any of the four storage classes `constant`, `uniform`, `varying`, or `vertex`. *class* is optional, because it is only relevant to primitive variables, and defaults to `constant` if left out. *type* can be any of: `float`, `color`, `point`, `vector`, `normal`, `string`, `matrix`, `hpoint`, or `integer`, or can be a fixed-length array of any of those types by providing a trailing integer array length inside square brackets.

EXAMPLE:

```
Declare "woodcolor" "color"
Declare "weights" "vertex float[4]"
```

We recommend using "in-line declarations" rather than using `Declare`, due mainly to the inconvenience and potential confusion arising from the single global dictionary. The `Declare` syntax is mainly supported for backward-compatibility with older scene files or modeling programs.

2.4 Motion Blur

MotionBegin `[`$time_0$ ... $time_{n-1}$`]`
MotionEnd

Describes how objects move in the scene by specifying how a particular transformation changes, or primitive deforms, over time. The `MotionBegin` statement takes an array of *n* floating-point time values. Between the `MotionBegin` and `MotionEnd` statements are *n* scene commands of identical routine name, but different in parameter values. Each command corresponds to a time value in the list supplied to `MotionBegin`. Valid scene commands that can be motion blurred include any transformations (Section 3.2.3), or any geometric primitive (Chapter 4).

The motion block describes transformations or primitives change over time, but which time segment is captured by the camera is determined separately by `Shutter` (Section 3.1.1). If there is no `Shutter` statement, no motion blur will be apparent in the scene.

EXAMPLE:

```
MotionBegin [1.0 1.02]
 Translate 3 0 0
 Translate 3.5 0 0
MotionEnd
```

```
MotionBegin [1.0 1.02]
 Polygon "P" [0 0 0   1 0 0   .5 2 0]
 Polygon "P" [0 0 0   1.2 0 0   .4 2.1 0]
MotionEnd
```

Note: Entropy 3.1 currently supports only two motion times for each motion block. This will be extended in a future release.

2.5 External Resources

2.5.1 Archive Files

ReadArchive *filename*

Parse and interpret scene file commands in the file given by the string *filename*. This is similar to the behavior of the C (or shader) `#include` — it is as if the entire contents of *filename* was at this location in the file.

EXAMPLE:

```
ReadArchive "chair.rib"
```

Note: when `ReadArchive` is used to include a specific model that is stored in a separate file, it is much more efficient to use `Procedural "DelayedReadArchive"` (see Section 4.8) if it is possible to bound the geometry in the file.

2.5.2 Making Texture Maps

Although ordinary image files may be turned into textures, environment maps, and shadows using an external program (`mkmip`, see Section 10.1), it is occasionally useful to have scene files themselves contain commands to that will convert the files.

Following are the descriptions of the routines that can perform these tasks as scene file commands. For further explanation on the meanings of all of the parameters, please refer to Chapter 10.

MakeTexture *sourceimage texturename swrap twrap filter swidth twidth ...parameterlist...*

Turns the image file named by the string *sourceimage* into a valid texture file to be stored with the name given by the string *texturename*. The strings *swrap* and *twrap* are the wrap modes, and may be any of: `"black"`, `"clamp"`, `"periodic"`, or `"mirror"` (these are explained in Section **??**). The string *filter* and floating-point parameters *swidth* and *twidth* determine the filter used for downsizing the texture to form a MIP-map. The *parameterlist* is an optional token-value list giving additional parameters.

EXAMPLE:

```
MakeTexture "scratch.tif" "scratch.tx" "black" "black" "box" 1 1
```

MakeCubeFaceEnvironment *px nx py ny pz nz texturename fov filter swidth twidth ...parameterlist...*

Turns the six image files named by the strings *px*, *nx*, *py*, *ny*, *pz*, *nz* into a valid cube-face environment map file to be stored with the name given by the string *texturename*. The floating-point parameter *fov* indicates the field of view (measured in degrees) of the six directional images, and should be at least 90. The string *filter* and floating-point parameters *swidth* and *twidth* determine the filter used for downsizing the texture to form a MIP-map. The *parameterlist* is an optional token-value list giving additional parameters.

EXAMPLE:

```
MakeCubeFaceEnvironment "px.tif" "nx.tif" "py.tif"
        "ny.tif" "pz.tif" "nz.tif" "refl.env" 90 "box" 1 1
```

MakeLatLongEnvironment *sourceimage texturename filter swidth twidth ...parameterlist...*

Turns the image file named by the string *sourceimage* into a valid latitude-longitude environment map file to be stored with the name given by the string *texturename*. The string *filter* and floating-point parameters *swidth* and *twidth* determine the filter used for downsizing the texture to form a MIP-map. The *parameterlist* is an optional token-value list giving additional parameters.

EXAMPLE:

```
MakeLatLongEnvironment "paintedenv.tif" "paintedenv.env" "box" 1 1
```

MakeShadow *depthmapname shadowmapname ...parameterlist...*

Turns the depth map file named by the string *depthmapname* into a valid shadow map texture file to be stored with the name given by the string *shadowmapname*. The *parameterlist* is an optional token-value list giving additional parameters.

EXAMPLE:

```
MakeShadow "light0.zfile" "light0.sm"
```

Chapter 3

Options and Attributes

This chapter describes the scene file routines that alter the *graphics state*. These routines fall into a two broad categories:

Options are properties that apply to the entire scene, such as image resolution.

Attributes are properties of individual object, and may vary from object to objects (an example is object color). Attributes include shader assignments and transformations (placement of objects in the scene).

The remainder of this chapter describes all of the options and attributes recognized by Entropy. Many of the common ones have dedicated commands to set them. More rarely-used options and attributes, or those that are very specific to Entropy's algorithms, are handled by a more general mechanism.

3.1 Options

Options are those properties that apply to the entire scene. Examples of scene options include image resolution and camera properties. The full list of "standard" options, each of which is set by a dedicated command, is given in Table 3.1. A number of more rarely-used options, generally related to Entropy's specific algorithms, are set through a more general mechanism and are listed in Table 3.2.

Because options apply to the entire scene and cannot vary from object to object, all option-setting commands must occur prior to `WorldBegin`. It is an error to set options between `WorldBegin` and `WorldEnd`.

3.1.1 Camera and Image Area Options

Clipping *hither yon*

Set the near and far clipping planes. Geometry whose z coordinate in camera space is less than *hither* or greater than *yon* will not be visible. There are also some computations in which `"camera"` space z values are normalized using the clip plane values

Option	Entropy defaults	
Camera options:		
Image resolution	640 x 480	(roughly video resolution)
Pixel aspect ratio	1.0	(square pixels)
Crop Window	(0,1,0,1)	(render entire image)
Frame Aspect Ratio	1.333	(determined by image resolution, if not set separately)
Screen Window	(-1.3333,1.3333,-1,1)	(determined by frame aspect ratio, if not set separately)
Camera projection	`"orthographic"`	
Near and far clipping planes	near=1e-6, far=1e30	
Other clipping planes	none	
Depth of field	fstop = ∞	(everything in focus)
Shutter time	open = close = 0	(no motion blur)
Image output options:		
Pixel samples	2 x 2	(4 regions per pixel)
Pixel filter	`"gaussian"` 2x2 pixels	
Exposure	gain=1, gamma=1	
Color (`"rgba"`) quantization	255 0 255 0.5	(output 8 bit color with dither)
Depth (`"z"`) quantization	0 0 0 0	(output floating point depths)
Display	`"ri.tif" "tiff" "rgb"`	(create an RGB TIFF file named `ri.tif`)
Imager shader	none	
Rendering options:		
Hider	`"hidden"`	(default rendering algorithm)

Table 3.1: Standard scene options and their default values.

(for example, `"screen"` space z or the return value of the shading language `depth()` function).

EXAMPLE:

```
Clipping .01 10000
```

NEW!

ClippingPlane *x y z nx ny nz*

Adds a user-defined clipping plane at position (x, y, z) and with normal (nx, ny, nz). The position and normal are in the coordinate system in effect at the time of the `ClippingPlane` statement. Geometry in the scene on the *positive* side (i.e., in the direction of the normal) will not be visible to the camera.

Note that this only applies to camera visibility. Objects that are culled from the camera view due to clipping planes will still be visible to ray tracing (reflection, shadows, indirect illumination, etc.). This may seem counter-intuitive, but is analogous

to `Clipping`— even objects outside the hither-yon range (from the P.O.V. of the camera) are visible in reflections.

EXAMPLE:

```
ClippingPlane 8 10 0 0 1 0
```

CropWindow *xmin xmax ymin ymax*

Designates a subregion ("crop window") of the image pixels to be rendered. The region is bounded by *xmin, xmax* horizontally and *ymin, ymax* vertically. The *xmin, xmax, ymin*, and *ymax* values are floats (ranging from 0 to 1) representing the portion of the image to be rendered, *not* raster (pixel) coordinates. The default is for the entire image to be rendered (`0 0 1 1`).

The pixel coordinates are rounded, and the border regions are computed in such a way that it is guaranteed that abutting crop windows will join together seamlessly.

EXAMPLE:

```
CropWindow 0 0.5 0 0.5
```

The example above causes just the upper left quadrant of the image to be rendered.

DepthOfField *fstop focallength focaldistance*

Simulates a camera lens with a particular focal length and f/stop, focused on objects at a given distance. The *focallength* and *focaldistance* parameters are measured in units of `"camera"` space. If *fstop* is `1e30` (effectively infinity), a pinhole camera will be used, resulting in a perfectly sharp image at all distances (this is the default behavior).

EXAMPLE:

```
DepthOfField 8 0.04 2.75
```

If the scene was modeled such that `"camera"` space had units of meters, the line above sets up an f/8, 40mm lens focused on objects 2.75m from the camera.

Format *xresolution yresolution pixelaspectratio*

Sets the full resolution (in pixels) of the image to be rendered. The *xresolution* and *yresolution* are integers giving the horizontal and vertical resolution, respectively. The *pixelaspectratio* is a float giving the aspect ratio (width/height) of the pixels (1.0 for square pixels).

EXAMPLE:

```
Format 640 480 1
```

FrameAspectRatio *aspectratio*

Sets the aspect ratio (width/height) of the image to be rendered. If `FrameAspectRatio` is not called, it will be determined from the `Format` parameters.

EXAMPLE:

```
FrameAspectRatio 1.333
```

Projection *projectionname ...parameterlist...*

Sets the projection used by the camera. The *projectionname* is a string that specifies the projection type. Entropy recognizes the projections `"orthographic"` (the default) and `"perspective"`. The optional *parameterlist* consists of token-value pairs supplying parameters specific to the projection. The `"perspective"` projection recognizes the float parameter `"fov"`, which specifies the field of view in degrees.

EXAMPLE:

```
Projection "perspective" "fov" [45]
```

ScreenWindow *left right bottom top*

Sets up the mapping from `"screen"` space (points projected onto the $z = 1$ plane in camera coordinates) to `"raster"` space (actual pixels in the final image). The $x = left$ line in `"screen"` space corresponds to the left edge of the raster image, $x = right$ to the right edge, $y = bottom$ to the lower edge, and $y = top$ to the upper edge.

The default values for `ScreenWindow` are:

$$(-frameaspectratio, frameaspectratio, -1, 1)$$

if $frameaspectratio \geq 1$. If $frameaspectratio < 1$, the default `ScreenWindow` coordinates are:

$$(-1, 1, -1/frameaspectratio, 1/frameaspectratio)$$

EXAMPLE:

```
ScreenWindow -1 1 -0.75 0.75
```

Shutter *opentime closetime*

Specifies the time range in which the camera's shutter is open, allowing moving objects to form a blurred image. Unlike a real camera, longer shutter times will not increase the amount of light exposure or change the brightness of the image. If $opentime = closetime$, the scene will be rendered with no motion blur.

EXAMPLE:

```
Shutter 0.0 0.016667
```

3.1.2 Image Formation and Display Options

Display *name format data ...parameterlist...*

Specifies an output stream for rendered image pixels. The *name* is a string that gives the name of an image file to write.

The *format* parameter is a string that specifies the type of file to write. Individual file formats or framebuffer types are implemented by *display drivers*. Entropy comes

with several display drivers (such as `"tiff"`), and also allows users or third parties to write additional display drivers to extend the set of formats that Entropy can write. A *format* of `"file"` indicates that the default file format (`tiff`) should be used, and `"framebuffer"` indicates that the default frame buffer display should be used.

NEW!

The *data* parameter is a string that is a comma-separated list of what data to output. Standard data fields include `"rgb"` (3-channel color), `"rgba"` (color and alpha), and `"z"` (depth information). Other values supplied indicate global variables (see Table 5.8) and *output variables* calculated by surface or atmosphere shaders. If non-standard data fields are specified, they should either have been pre-declared with `Declare`, or have "inline" type declarations.

The optional *paramlist* is a list of token-value pairs giving parameters specific to the particular display driver being used, and/or any of the following:

`"string filter"`	The name of the pixel filter to use.
`"float filterwidth[2]"`	The width (in x and y) of the pixel filter to use.
`"float quantize[4]"`	The *zero*, *one*, *min* and *max* quantization levels. See `Quantize` (p. 21) for their meanings.
`"float dither"`	The dither amplitude (0 for no dither).

Note that the `"filter"` and `"filterwidth"` override the `PixelFilter` options for this display output stream, and `"quantize"` and `"dither"` override the `Quantize` option for this display output stream. Any *paramlist* tokens other than the ones above will be passed along to the display driver. Check the documentation for the specific display driver to see what optional parameters it can take.

NEW!

If the first character of the *name* is `+` (the plus character), an *additional* display output stream will be created. There may be any number of display output streams, and each may have entirely different bit depths, quantization parameters, and pixel filters.

EXAMPLES:

```
    Display "frame0001.tif" "tiff" "rgba"
    Display "+frame0001.z" "zfile" "z"
    Display "+spec0001.tif" "file" "color specularpass" "quantize"
[0 65535 0 65535]
```

The above commands create three display output streams: (1) an TIFF file `frame0001.tif` containing the color and alpha of the image, using the pixel filter and quantization previously specified by `PixelFilter` and `Quantize`; (2) a `"camera"` space depth map `frame0001.z` (which because it's outputting z depth will use a `"box" 1 1` filter and the `"z"` `Quantize` values; (3) a color TIFF file `"spec0001.tif"` containing the `"specularpass"` surface shader output variable, as a 16-bit-per-channel file (overriding the `Quantize` values).

Exposure *gain gamma*

Indicates that the renderer should transform all colors by passing them through the following formula *prior to quantization*:

$$color = (color \cdot gain)^{1/gamma}$$

The default is $gain = 1, gamma = 1$.

EXAMPLE:

```
Exposure 1 2.2
```

Hider *name ...parameterlist...*

Sets the hidden surface algorithm to the one specified by the string *name*. A renderer may allow you to choose one of several different hidden surface algorithms available. The one named `"hidden"` indicates that the renderer should use the default algorithm. An optional *parameterlist* of token-value pairs may specify parameters specific to the algorithm being used.

EXAMPLE:

```
Hider "hidden"
```

Imager *shadername ...parameterlist...*

Sets the imager shader to *shadername*. The optional *paramlist* is a list of token-value pairs giving parameters specific to the particular shader being used. All parameters should either have their types declared "inline" or have been previously declared with `Declare`.

EXAMPLES:

```
Imager "clamptoalpha"
```

PixelSamples *xsamples ysamples*

Set the number of subdivisions per pixel in order to control antialiasing. Larger numbers product higher quality images, but take longer to render.

TIP: `PixelSamples 1 1` is good for fast previews. With Entropy, `4 4` is good enough for most final still images. If you are using lots of motion blur or depth of field, you may need `6 6` or higher. The "right" value depends greatly on the contents of the image. This is a basic time-versus-quality knob.

EXAMPLE:

```
PixelSamples 2 2
```

PixelFilter *filtername xwidth ywidth*

Final image pixels are produced by taking the results of many pixel subregions (determined by `PixelSamples`) and reconstructing a single pixel value that is a weight average of the contribution from the subregions, including those from nearby pixels. `PixelFilter` allows you to specify the weighting filter by name (`"box"`, `"triangle"`, `"catmull-rom"`, `"sinc"`, or `"gaussian"`) and the full width of the filter extent. The default is to use a `"gaussian"` filter with width 2 in each direction. This should be adequate for most images, but `PixelFilter` gives you the freedom to customize the filter and width.

EXAMPLE:

```
PixelFilter "gaussian" 2 2
```

Quantize *data one min max ditheramplitude*

Determines the quantization (scaling and method of converting to integer values) of the image data. The *one*, *min*, and *max* are integer values and *ditheramplitude* is a floating point value. Image data are scaled so that an original floating point pixel value of 1.0 gets a pixel value of *one*, a pseudo-random dither of amplitude *ditheramplitude* is added to eliminate banding artifacts, then the integer value is clamped to lie between *min* and *max*, respectively. More formally, the float-to-integer quantization is performed according to the following formula:

$$\begin{aligned} pixelval &= \mathrm{round}(one * floatval + ditheramplitude * \mathrm{random}()) \\ pixelval &= \mathrm{clamp}(pixelval, min, max) \end{aligned}$$

The range of *min* and *max* determine the *bit depth* of the resulting output image: if both are ≤ 255, 8-bit integer (per channel); if both are ≤ 65535, 16-bit integer; otherwise 32-bit integerr. If all four numeric parameters are 0, then no integer quantization is performed and a floating-point image is output. If *ditheramplitude* is 0 (as it should be for floating-point images), no dithering is performed.

The *data* parameter is a string that indicates which image output data fields should be quantized in this manner, and may be `"rgba"`, `"z"`.

EXAMPLES:

The proper settings for an 8-bit image (and the default settings for quantizing color data) are:

```
Quantize "rgba" 255 0 255 0.5
```

For full floating point output (the default for depth data):

```
Quantize "rgba" 0 0 0 0
```

3.1.3 Implementation Options

The previous sections have described RIB statements that set many commonly-used options. There are many additional options that do not have their own separate RIB statements, either because they are less commonly used or because they set parameters that are extremely specific to a particular rendering method. Implementation-specific options are set by using the extendible `Option` call:

Option *category ...parameterlist...*

Sets options of a particular *category* (a string representing an option category recognized by the renderer). Specific parameters in that category are supplied as a list of token-value pairs. Options apply to the entire scene and must be set prior to `WorldBegin`.

If *category* is `"user"`, the token and value are added to the option state even if it is not recognized as a standard option. This value can later be retrieved through the shading language function `option()`. User options *must* have their types declared, either through the use of "inline declares" or the `Declare` command.

NEW!

EXAMPLE:

```
Option "limits" "integer texturememory" [10000]
Option "user" "float temperature" [212]
```

The remainder of this section describes the implementation-specific options that are recognized by Entropy. In most cases, the new "inline declaration" syntax is used to clarify the expected data types, and the default values are provided as examples.

Option	**Entropy defaults**	
Rendering options:		
Spatial antialiasing quality	[1 1]	
Temporal antialiasing quality	4	
Minimum shadow bias	0.01	
Maximum ray depth	4	
Indirect illumination number of bounces	1	
Indirect illumination "save file"	none	
Indirect illumination "seed file"	none	
Texture cache memory	10000	(10 MB)
Default specular function	`"entropy"`	
Z threshold color	(1, 1, 1)	
Verbosity	`"normal"`	(some messages printed)
Statistics level	0	(no stats printed)
Statistics log file	none	(stats print to `stdout`)
Search paths:		
Archive search path	`""`	
Texture search path	`""`	
Shader search path	`""`	
Procedural search path	`""`	
Display driver search path	`""`	

Table 3.2: Implementation options and their default values.

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 Entropy are:

`"archive"`	files included by `ReadArchive`.
`"texture"`	texture, shadow, and environment maps.
`"shader"`	compiled shaders.
`"procedural"`	DSO's and executables for `Procedural` calls.
`"display"`	DSO's for custom display drivers.

Search path types in Entropy 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 Entropy's search paths:

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

- `$ARCH` is replaced with the name of the machine architecture (`intelnt`, `linux`, or `sgi_m3`). This allows you to keep compiled software (like DSO's) for different platforms in different directories, without having to hard-code the platform name into your file.
- `$`*`VAR`*, `${`*`VAR`*`}`, `$(`*`VAR`*`)`, and `%`*`VAR`*`%` are replaced by the value of environment variable `VAR`, if it exists (for any environment variable). NEW!

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

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

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

Statistics and Messages

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

When nonzero, this option will cause Entropy 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 Entropy's statistics to be echoed to the given filename, rather than printed to `stdout`.

`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"`.

Global Illumination Controls

NEW!

```
Option "indirect" "integer maxbounce" [1]
```

Sets the maximum number of bounces of indirect lighting. Larger numbers will take more time to compute, but may have more correct results for scenes in which much of the light propagation is from complex multiple-bounce paths. The default (if indirect lighting is used at all) is one (1) bounce.

```
Option "indirect" "string savefile" [""]
```

If you specify this option with a non-empty string, 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" [""]
```

If you specify this option with a non-empty string, the irradiance data cache will 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.

Occlusion Information Controls

These options control aspects of the computations that are used by the `occlusion()` shader function (see Section 5.6.8).

NEW!

```
Option "occlusion" "string savefile" [""]
```

If you specify this option with a non-empty string, when rendering is done the contents of the occlusion 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 "occlusion" "string seedfile" [""]
```

If you specify this option with a non-empty string, the occlusion data cache will 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.

Rendering Options

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

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

```
Option "render" "string specularbrdf" ["entropy"]
```

NEW!

Specifies which of several specular highlight shapes is used in rendering. Entropy's default specular function, specified by `"entropy"`, is a modified Cook-Torrance function with Torrance-Sparrow distribution (see Roy Hall, *Illumination and Color in Computer Generated Imagery*, pp. 83 and 96). Also available are `"bmrt"` (a modified Blinn-Phong). The default specular function may be overridden by supplying the specular function name to the `specular()` function itself (see Section 5.6.8).

```
Option "limits" "color zthreshold" ["1 1 1"]
```

Specifies the minimum opacity for an object to be considered sufficiently opaque to show up in a shadow depth map. If an object's opacity (`Oi`) is not greater than this limit, it will not affect the "z" output channel (though it will still contribute to the pixel color and alpha).

Quality/Performance Tradeoff Options

```
Option "limits" "integer spatialquality[2]" [1 1]
Option "limits" "integer temporalquality" [4]
```

NEW!

Sets the quality level of the antialiasing. Unlike other renderers that use a single `PixelSamples` value to antialias in all dimensions at once, Entropy lets you set the required spatial antialiasing (for geometric edges and depth of field) separately from the additional amount of work needed to resolve motion blur. One advantage of this approach is that significant savings can be achieved for scenes in which only some objects are undergoing motion blur.

The default spatial quality is 1×1, which gives an appearance roughly like what you would get from `PixelSamples 2 2` in a point-sampling-based renderer like PRMan or BMRT. A spatial quality level of 2×2 is about as good as `PixelSamples 5 5` in BMRT or PRMan, and should be adequate for most images. Higher values may be needed for scenes with lots of thin geometry (like hair). Images with a significant amount of motion blur should also ensure that the `"temporalquality"` option is set (see below).

The default temporal quality is 4, which should be adequate for moderate amounts of motion blur. For scenes with significant motion blur, this can be raised up to 8. *Setting a high temporal quality does not increase rendering time for scenes that do not have motion blur, or for pixels that do not intersect with blurring objects.*

The `PixelSamples` x y command actually sets `temporalquality` to 4, and sets `spatialquality` to a total of $x/2 \times y/2$.

NEW! `Option "limits" "integer raydepth" [4]`

Sets the maximum depth of recursive rays that will be cast between reflectors and refractors. This has no effect if there are no ray traced reflections or refractions in the scene. (This option performs the same function as the deprecated BMRT `Option "render" "max_raylevel"`.)

`Option "limits" "integer texturememory" [10000]`

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 10000 (i.e., 10 MB). 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.

NEW! `Option "limits" "integer texturefiles" [100]`

Sets the maximum number of simultaneously-open texture file handles. On some systems, there is a maximum number of allowable open file handles. This option helps you to ensure that Entropy does not exceed that allotment. When the limit is reached, files that have not been accessed recently are closed to make room for newer file handles. The default of 100 simultaneously open files should be adequate for almost all systems and scenes.

`Option "limits" "integer bucketsize[2]" [32 32]`

Sets the size (in x and y pixels) of the screen buckets that represent units of work for the renderer. By default, Entropy will choose an appropriate bucket size based on the resolution and sampling rate of your image, unless you override by setting this option. There should be no reason to override the default for ordinary scenes; however, advanced users may wish to tune performance on problematic scenes by adjusting this option. Setting either the x or y bucket size to 0 reset to the default.

`Option "limits" "integer gridsize" [256]`

Sets the number of surface points that will be shaded at one time. The default is 256. There should be no reason to override the default for ordinary scenes.

`rgl`-Specific Options

```
Option "limits" "integer curvethinning" [1]
Option "limits" "integer curvethinthreshold" [1]
```

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. If the `"curvethinning"` frequency is set to zero, no curve thinning will take place at all.

3.2 Attributes

Attributes are properties that apply to individual objects in the scene, and may be different for each object. Examples of object attributes include color, object transformation (position/orientation), and shader assignments. The full list of "standard" attributes, each of which is set by a dedicated command, is given in Table 3.3. A number of more rarely-used attributes, generally related to Entropy's specific algorithms, are set through a more general mechanism and are listed in Table 3.4.

Attribute	**Entropy defaults**	
Shading attributes:		
Surface color	(1,1,1)	(white)
Surface opacity	(1,1,1)	(fully opaque)
Texture coordinates	(0, 0, 1, 0, 0, 1, 1, 1)	($s = u, t = v$)
Light Sources	none	
Area light source	none active	
Surface shader	`defaultsurface`	
Atmosphere shader	none	
Exterior volume shader	none	
Interior volume shader	none	
Shading rate	1	(approx. one shade per pixel)
Matte flag	0	(objects are not hold-out mattes)
Geometric attributes:		
Displacement shader	none	
Orientation	`"outside"`	(normals have the same handedness as the transformation)
Sides	2	(ok to view both sides of objects)
Cubic basis matrices	Bezier	
NURBS trim curves	none	(draw all of each NURBS surface)
Detail Range	$(0, 0, \infty, \infty)$	(draw all primitives)

Table 3.3: Standard attributes and their default values.

As it parses scene file commands, the renderer keeps track of the *current attribute state* — that is, the full set of attributes and their values. When a geometric primitive is declared, a copy of the current attribute state is *bound*, or permanently attached, to that geometric primitive. Thus, setting attribute values can affect the appearance of subsequently declared geometry, but does not change previously declared geometry. Because attributes may be changed for each object, is convenient to save the attribute state, modify attributes and declare geometric primitives, then restore the attribute state to its prior condition. A command is provided to perform this action:

AttributeBegin
AttributeEnd

Save and restore the attribute state, including the current transformation (see `TransformBegin`

and `TransformEnd`, section 3.2.3). Upon `AttributeEnd`, the current attribute set is replaced by the attribute set that was in effect at the corresponding `AttributeBegin`.

EXAMPLE:

```
AttributeBegin
AttributeEnd
```

It is perfectly legal to *nest* blocks delimited by `AttributeBegin`/ `AttributeEnd`. In other words, the renderer maintains a *stack* of attributes that is *pushed* by `AttributeBegin` and *popped* by `AttributeEnd`.

3.2.1 Standard Attributes

Color `[`*r g b*`]`

Sets the default surface color. This color is the value that the shader variable `Cs` has if it is not overridden by supplying a value on the geometric primitive.

EXAMPLE:

```
Color [1 0.5 0.5]
```

GeometricApproximation *type value*

Deprecated

This routine is obsolete, and in Entropy is exactly equivalent to:

```
Attribute "dice" "float type" [value]
```

EXAMPLE:

```
GeometricApproximation "motionfactor" 1
```

Matte *onoff*

If *onoff* is nonzero, subsequent geometry will be treated as a "hold-out matte." Matte objects block the view of objects behind them, but nonetheless have 0 opacity, thus leaving a "hole" in the final image. The default is for all geometry to be non-matte (ordinary geometry).

EXAMPLE:

```
Matte 0
```

Opacity `[`*r g b*`]`

Sets the default surface opacity (0 is transparent, 1 is opaque). This opacity is the value that the shader variable `Os` has if it is not overridden by supplying a value on the geometric primitive.

EXAMPLE:

```
Opacity [1 1 1]
```

Orientation *orient*

Sets the current orientation (that is, which of the two possible directions are chosen for the surface normal). The *orient* parameter is a string indicating one of four possible settings:

`"outside"`	same as the coordinate system's handedness
`"inside"`	opposite the coordinate system's handedness
`"lh"`	left handed orientation (regardless of CTM handedness)
`"rh"`	right handed orientation (regardless of CTM handedness)

EXAMPLE:

```
Orientation "outside"
```

ReverseOrientation

Switch the current orientation from its current state to the opposite state (i.e., from left handed to right handed, or from right handed to left handed).

EXAMPLE:

```
ReverseOrientation
```

ShadingRate *area*

Determines the spacing of shading calculations are performed on surfaces, so that the pixel area for each sample is *area*. For example, the default value of 1.0 indicates that surfaces should be shaded approximately once per pixel. A value of 4.0 would indicate that surfaces would be shaded about once every four pixels.

We strongly recommend using the default value of 1.0 for all ordinary high-quality rendering. Values smaller than 1 will make rendering take longer, but generally will not improve the final image. Larger values will render more quickly, but will have less detail and may appear blurry (but this may be desired for faster previews).

EXAMPLE:

```
ShadingRate 1
```

Sides *nsides*

If *nsides* is 2, subsequent geometry is "two-sided" and can be seen from either side. If *nsides* is 1, this indicates that the geometry forms a closed opaque object whose natural geometric normals always point to the outside of the object (or whichever side the camera will be on). `Sides 1`, therefore, is a hint to the renderer that if such geometry is pointed away from the camera, it must be on the "far side" of the object, and so will be found to be occluded, hence may be ignored.

EXAMPLE:

```
Sides 2
```

TextureCoordinates `[` *`s0 t0 s1 t1 s2 t2 s3 t3`* `]`

For parametric primitives (patches and patch meshes, NURBS, and quadrics), set the affine mapping between parametric (u, v) and texture coordinates (s, t). Specifically, $(u = 0, v = 0)$ will get texture coordinates (s_0, t_0), $(u = 1, v = 0)$ will get texture coordinates (s_1, t_1), $(u = 0, v = 1)$ will get texture coordinates (s_2, t_2), and $(u = 1, v = 1)$ will get texture coordinates (s_3, t_3). By default, $s = u$ and $t = v$. `TextureCoordinates` does not affect polygons or subdivision surfaces. Even on those primitives that do honor `TextureCoordinates`, any supplied `"s"` and `"t"` vertex coordinates take precedence.

EXAMPLE:

```
TextureCoordinates [0 0 1 0 0 1 1 1]
```

3.2.2 Implementation Attributes

The previous section described RIB statements that set many commonly-used attributes. There are many additional attributes that do not have their own separate RIB statements, either because they are less commonly used or because they set parameters that are extremely specific to a particular rendering method. The implementation-specific attributes are set by using the extendible `Attribute` call:

Attribute *`category ...parameterlist...`*

Sets attributes of a particular *category* (a string representing an attribute category recognized by the renderer). Specific parameters in that category are supplied as a list of token-value pairs. Attributes apply to specific pieces of geometry, and are saved and restored by the `AttributeBegin` and `AttributeEnd` commands.

NEW!

If *category* is `"user"`, the token and value are added to the attribute state even if it is not recognized as a standard attribute. This value can later be retrieved through the shading language function `attribute()`. User attributes honor the usual `AttributeBegin`/`AttributeEnd` scoping rules, but *must* have their types declared, either through the use of "inline declares" or the `Declare` command.

EXAMPLE:

```
Attribute "caustic" "color specularcolor" [0 0 0]
Attribute "user" "point refpoint" [3.14 2 1]
```

The remainder of this section describes the implementation-specific attributes that are recognized by Entropy. In most cases, the new "inline declaration" syntax is used to clarify the expected data types, and the default values are provided as examples. All of the attributes and their defaults are summarized in Table 3.4.

Attribute	Default	
Geometric attributes:		
Trim curve sense	`"inside"`	(keep the inside of the trim curves)
Displacement bound radius	0	(no displacement bound padding)
Displacement bound space	`"object"`	
Dicing and Subdivision attributes:		
Ray trace truly displaced geometry	1	(on)
Binary dicing	0	(off)
Motion factor	0	(no adjust of shading due to motion)
Curvature max angle	5	(degrees)
Curvature max refinement factor	1	(don't adjust)
Light Source attributes:		
Area light samples	1	
Caustic number of light photons	0	(do not photon map the light)
Rendering attributes:		
Camera visibility	1	(objects are visible to the camera)
Ray-traced reflection visibility	0	(objects do not appear in RT reflections)
Ray-traced shadow visibility	0	(objects do not appear in RT shadows)
Shadow ray opacity	`"opaque"`	(do not run the shader for shadow rays)
Trace against truly displaced objects?	1	(yes)
Indirect illum max error	0.25	
Indirect illum max pixel distance	20	
Indirect illum number of samples	256	
Caustic max pixel distance	16	
Caustic number of photons to gather	75	
Caustic specular color	(0,0,0)	(object not specularly reflective)
Caustic refraction color	(0,0,0)	(object not refractive)
Caustic refraction index	1	

Table 3.4: Implementation attributes and their default values.

Visibility of Primitives

NEW!

```
Attribute "visibility" "integer camera" [1]
Attribute "visibility" "integer reflection" [0]
Attribute "visibility" "integer shadow" [0]
```

Controls which rays may *see* an object. The default is for all objects to be visible in the camera view, but not to cast ray-traced shadows or to appear in ray-traced reflections. For any of the visibility types, a visibility of `0` indicates that it is not visible, and any nonzero value indicates that it is visible.

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. Objects which are only visible from the camera, but not from ray traced reflections or shadows, can be handled much more efficiently by the renderer.

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

Controls how surfaces shadow other surfaces. A value of `"opaque"` indicates that when hit by a shadow ray, the object is opaque. A value of `"Os"` indicates that the shadow ray should have a "density" indicated by `Os`. A value of `"shade"` indicates that the occluder's surface shader should be run at the intersection of the shadow ray in order to evaluate the specific `Oi` at that point.

The choices `"opaque"` (the default) and `"Os"` are much less expensive than `"shade"`, but sometimes you need `"shade"` if you want the shader to control the shadow opacity on a point-by-point basis. Of course, no matter what the setting of this attribute, the object will not cast shadows unless it has been included in the ray-traced shadow list by `Attribute "visibility" "shadow" [1]`.

Displacement and Subdivision Attributes

```
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" "integer tracedisplacements" [1]
```

Controls whether, for traced rays, the displacement shaders are run on entire grids (making many small pieces that need to be traced), or whether a bump-mapping approximation can be used. For objects where the amount of displacement is sufficiently small that a bump approximation is good enough (as seen in reflections or shadows), setting this attribute to 0 can *greatly* reduce the time and memory necessary to render the object. In either case, displacement shaders will truly move points to create ragged silhouettes as seen by the camera — this attribute only affects the appearance of objects in ray traced reflections or shadows.

```
Attribute "dice" "integer binary" [0]
```

When nonzero, causes dicing rates for patches to be rounded up to the next highest power of 2. This makes rendering a bit more expensive, but can be helpful in eliminating tiny cracks between adjacent pieces of geometry.

NEW!

`Attribute "dice" "float motionfactor" [0]`

Scales the effective shading rate of motion-blurred objects. The more an object moves, the coarser it will be shaded. This can speed up rendering of motion-blurred objects, which, due to blur, tend not to show much detail anyway. A value of `1` is typical. Larger values scale the shading rate more aggressively, smaller values more conservatively. The default is 0 (no adjustment of `ShadingRate` is performed for moving objects).

NEW! `Attribute "dice" "float maxscanlinecurvature" [120]`

Forces extra dicing every time a patch bends by an angle greater than this parameter (measured in degrees), even when the usual arc length metrics indicate that fewer divisions are needed. This keeps really thin features (like tubes) from degenerating. The default value (120 degrees) is the maximum, and should be adequate in nearly all cases. But very occasionally, it is useful to decrease the threshold if you are losing geometric detail or experiencing highlight aliasing on highly curved objects.

NEW! `Attribute "dice" "float maxraytracingcurvature" [5]`

Analogous to `"maxscanlinecurvature"`, but of course for ray tracing. Increasing this threshold will make ray tracing faster, but with possible loss of fine geometric detail in ray-traced reflections and shadows. Decreasing this threshold will capture more geometric detail in the ray tracing of highly curved geometry, but at some additional expense. In no case will this parameter ever cause the ray-traced geometry to dice more finely than it would have for scanline rendering (i.e., using the arc length and `"maxscanlinecurvature"` metrics).

NEW! `Attribute "dice" "integer keepcreases" [1]`

When nonzero (the default), creases (2nd order discontinuities or repeated knot values) are guaranteed to be preserved by having a splitting or dicing division at that position. When set 0, this parameter allows the renderer to "dice over" a crease. This can come in handy for faster rendering of very small (but creased) geometry.

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"`.

Light Source Attributes

Unlike all of the other attributes that bind to subsequent geometric primitives, the following attributes bind to subsequent light sources.

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

Sets the number of times to sample a particular light source for each shading calculation. This is especially useful for reducing the noise in area light soft shadows. By increasing the number of samples, you can reduce the noise in the shadows.

`Attribute "light" "integer motionrays" [0]` NEW!

If this attribute is passed a nonzero integer value, any `shadow()` calls made in the light shader that cause rays to be traced will attempt to motion blur those shadows. This works very well if the visible object (the one the shadow is being cast upon) is still, but the occluding object is moving. It can have artifacts if the visible object (upon which the shadow falls) is also moving, but in most cases this will not be objectionable. If it is, you can always turn the attribute off. Note that a light source's ray-traced shadows will also be motion blurred if the surface has its `Attribute "render" "motionrays"` turned on; in other words, either the light or the surface may trigger motion blurred shadow rays.

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

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 ]
```

Note, however, that we consider this attribute deprecated, and strongly prefer that shadows be controlled in the light source shader by a `shadow()` call. Since Entropy extends `shadow()` to allow ray traced shadows, it should be easy to write light shaders that are flexible enough to support shadow maps or ray tracing with minimal modification.

Miscellaneous

`Attribute "render" "integer motionrays" [0]` NEW!

If this attribute is passed a nonzero integer value, subsequent objects will attempt to motion blur any shadows or reflections that fall upon them. This works very well if the visible object is still, but the reflected (or occluding) object is moving. It may have artifacts if the visible object (upon which the reflections or shadow fall) is also moving, but in most cases this will not be objectionable. If it is, you can always turn the attribute off. Note that for ray traced shadows, motion blur can also be turned on for the light using `Attribute "light" "motionrays"`.

`Attribute "identifier" "string name" [""]`

Sets the name of subsequent object. This is used, among other things, for printing more helpful error messages by indicating which object had the error.

Global Illumination Controls

`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. Smaller numbers are higher quality, but use more memory and take longer to render.

`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. It should be obvious how this is used. Use as low a number that gives an acceptable appearance, as the amount of time spent calculating indirect illumination is directly proportional to this.

Caustic Controls

`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 that we want the light to shoot and store in order to calculate caustics. The light may actually shoot many more photons in order to get this number of photons reflected/refracted and stored. 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).

`rgl`-Specific Attributes

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

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

3.2.3 Transformations

As it parses scene file commands, the renderer keeps track of the *current transformation* (sometimes called the *CTM* for "current transformation matrix." When a geometric primitive is declared, a copy of the CTM is *bound*, or permanently attached, to that geometric primitive. More specifically, the geometric primitive adopts the CTM as its `"object"` space, and therefore any point locations (such as vertex positions `"P"`) are relative the CTM

that was in effect at the time that the geometric primitive command was encountered. Thus, transformation commands affect the appearance of subsequently declared geometry, but do not change previously declared geometry.

Because each object has its own position, and because complex models are often described as hierarchies, it is convenient save the transformation state, modify the transformation and declare geometric primitives, then restore the transformation to its prior condition. A command is provided to perform this action:

TransformBegin
TransformEnd

Save and restore the current transformation. Upon `TransformEnd`, the current transformation is set to the transformation that was in effect at the corresponding `TransformBegin`.

EXAMPLE:

```
TransformBegin
TransformEnd
```

Remember that the current transformation is actually part of the attribute state, therefore the CTM is also saved and restored (along with the rest of the attribute state) by `AttributeBegin` and `AttributeEnd`.

A variety of commands are available to replace or modify the current transformation. The two most fundamental (and upon which all others are based) are `Transform` and `ConcatTransform`.

Transform `[` *transform* `]`

Replace the current transformation with the 4x4 matrix supplied.

EXAMPLE:

```
Transform [1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]
```

ConcatTransform `[` *transform* `]`

Concatenate the given 4x4 transformation matrix onto the current transformation.

EXAMPLE:

```
ConcatTransform [1 0 0 0 0 1 0 0 0 0 1 0 3 0 0 1]
```

The `Transform` and `ConcatTransform` routines, which replace and concatenate the CTM, respectively, are fully general. For several of the most useful and common transformations, there are specific routines that have a more compact, simpler syntax:

Identity

Set the current transformation to the identity transformation. This is identical to the call:

```
Transform [1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]
```

Translate *x y z*

Prepend the current transformation with the given translation. This is idenical to the call:

```
ConcatTransform [1 0 0 0 0 1 0 0 0 0 1 0 x y z 1]
```

EXAMPLE:

```
Translate 2 0 0
```

Rotate *angle x y z*

Prepend the current transformation with a rotation of *angle* degrees about the axis defined by (x, y, z).

EXAMPLE:

```
Rotate 30 0 0 1
```

Scale *sx sy sz*

Prepend the current transformation with a scale factor of (*sx, sy, sz*). This is idenical to the call:

```
ConcatTransform [sx 0 0 0 0 sy 0 0 0 0 sz 0 0 0 0 1]
```

EXAMPLE:

```
Scale 2 2 2
```

Skew *angle x1 y1 y2 x2 y2 z2*

Prepend the current transformation with a matrix that skews by shifting points along lines parallel to (*x2, y2, z2*). Point are by an amount that maps points on the axis (*x1, y1, z1*) to an axis that forms an angle of *angle* with (*x1, y1, z1*).

EXAMPLE:

```
Skew [30 0 0 1 0 1 0]
```

Perspective *fov*

Prepends the current transformation matrix with a perspective matrix having the given field of view (in degrees). Perspective matrices have nasty singularities that make some regions of space unable to transform correctly.

EXAMPLE:

```
Perspective 90
```

Several coordinate systems have pre-declared names: `"world"`, `"camera"`, `"screen"`, `"NDC"`, `"raster"`. The user may tag other coordinate systems with names; these names are then added to a list of named coordinate systems. Two routines exist to name and recall coordinate systems:

CoordinateSystem *name*

Create a named alias for the current transformation. The *name* may be used with `CoordSysTransform` or as the name of a transformation in the various shader routines that can specify named transformation. The mapping between names and coordinate systems is global and is not saved and restored with the rest of the attribute state. If there are two calls to `CoordinateSystem` with the same *name*, the second one will simply overwrite the first. This routine does not modify the current transformation in any way.

EXAMPLE:

```
CoordinateSystem "leftarm"
```

CoordSysTransform *name*

Replace the current transformation with the named transformation, which may either be a standard coordinate system name (such as `"world"`, `"camera"`, etc.) or a name defined by `CoordinateSystem`.

EXAMPLE:

```
CoordSysTransform "leftarm"
```

3.2.4 Shaders and Lights

Every object has a collection of shaders bound to it: displacement, surface, atmosphere, interior, and exterior. Only the surface shader is required; the other shader types may be set to nothing. For ordinary objects as viewed by the main camera, shaders are run in the following order: displacement (if it exists), surface, atmosphere (if it exists).

For ray traced reflections or refractions, the sequence is similar, except that no atmosphere shader is run. Rather, either the interior or exterior shader is run, depending on the direction of the traced ray compared to the normal of the surface.

The `LightSource` command creates a light. The `AreaLightSource` creates an area light, to which all subsequently-declared geometric primitives will be added, until the enclosing attribute block is exited (via `AttributeEnd`).

The attribute state includes a list of currently active lights. The `LightSource` and `AreaLightSource` commands add their newly-created lights to the active list. Note that the active light list is part of the attribute state and will be saved and restored by `AttributeBegin` and `AttributeEnd`. Thus, if a light is declared inside an Attribute block, it will be turned off for all primitives declared after the surrounding `AttributeEnd`. But the `Illuminate` command can override this behavior by turning individual lights on and off for subsequently declared primitives. Note also that a light cannot possibly illuminate a geometric primitive that is declared earlier in the scene file. For this reason, lights are typically declared immediately after `WorldBegin`, before any ordinary scene geometry is created.

Following are descriptions of the commands that alter shader assignments and lights:

Surface *shadername ...parameterlist...*

Sets the surface shader to *shadername*. Surface shaders are responsible for determining the color (`Ci`) and opacity (`Oi`) of a point on the geometric primitive, as it appears from a viewing direction.

The optional *paramlist* is a list of token-value pairs giving parameters specific to the particular shader being used. All parameters should either have their types declared "inline" or have been previously declared with `Declare`.

EXAMPLES:

```
Surface "wood" "float Kd" [0.75] "color darkwood" [.1 .08 .03]
```

Displacement *shadername ...parameterlist...*

Sets the displacement shader to *shadername*. Displacement shaders are run prior to the surface shader, and may be used to make small-scale adjustments (such as scratches, bumps, or dents) to the shape of the geometric primitive.

The optional *paramlist* is a list of token-value pairs giving parameters specific to the particular shader being used. All parameters should either have their types declared "inline" or have been previously declared with `Declare`.

EXAMPLES:

```
Displacement "dented" "float dentamp" [1.3]
```

Atmosphere *shadername ...parameterlist...*

Sets the atmosphere shader to a volume shader *shadername*. Atmosphere shaders run after the surface shader, to account for how `Ci` and `Oi` are altered as the light from the surface travels through the participating media to arrive at the camera. Atmosphere shaders are not called on ray-traced reflections or refractions.

The optional *paramlist* is a list of token-value pairs giving parameters specific to the particular shader being used. All parameters should either have their types declared "inline" or have been previously declared with `Declare`.

EXAMPLES:

```
    Atmosphere "thickfog" "float density" [0.01] "color fogcolor"
[1 .9 .3]
```

Interior *shadername ...parameterlist...*

Sets the interior volume shader to *shadername*. Interior shaders are used to adjust the attenuation of light of refracted rays (much as atmosphere shaders adjust light that travels from a surface to the camera).

The optional *paramlist* is a list of token-value pairs giving parameters specific to the particular shader being used. All parameters should either have their types declared "inline" or have been previously declared with `Declare`.

EXAMPLES:

```
Interior "glassint" "float extinction" [.01]
```

Exterior *shadername ...parameterlist...*

Sets the interior volume shader to *shadername*. Exterior shaders are used to adjust the attenuation of light of reflected rays (much as atmosphere shaders adjust light that travels from a surface to the camera).

The optional *paramlist* is a list of token-value pairs giving parameters specific to the particular shader being used. All parameters should either have their types declared "inline" or have been previously declared with `Declare`.

EXAMPLES:

```
    Exterior "thickfog" "float density" [0.01] "color fogcolor" [1
.9 .3]
```

LightSource *shadername lightid ...parameterlist...*

Make a light source that uses the light shader *shadername* and add the light to the list of currently active (illuminated) lights. The light source's identifier is *lightid*, which may be either an integer or a quoted string. The identifier may be used to turn the light on and off with `Illuminate`.

The optional *paramlist* is a list of token-value pairs giving parameters specific to the particular light shader being used. All parameters should either have their types declared "inline" or have been previously declared with `Declare`.

EXAMPLES:

```
    LightSource "barnlight" 13 "color lightcolor" [.5 1 .5] "string
shadowname" ["lgt5.sm"]
```

AreaLightSource *shadername lightid ...parameterlist...*

Make an area light source that uses the light shader *shadername* and add the light to the list of currently active (illuminated) lights. The light becomes the "active area light," so any subsequent geometry is added to that area light, until we hit `AttributeEnd`. The light source's identifier is *lightid*, which may be either an integer or a quoted string. The identifier may be used to turn the light on and off with `Illuminate`.

The optional *paramlist* is a list of token-value pairs giving parameters specific to the particular light shader being used. All parameters should either have their types declared "inline" or have been previously declared with `Declare`.

EXAMPLES:

```
    LightSource "arealight" 21 "float intensity" [100]
```

Illuminate *lightid onoff*

Add or remove a light that has already been declared with `LightSource` or `AreaLightSource`. The *lightid* is corresponds to the identifier of a previously declared light (and thus may be either an integer or quoted string, depending on how the light was declared). If *onoff* is 0, the light is removed from the active light list, and therefore does not

shine on subsequently declared geometry). If *onoff* is 1, the light is added to the active light list (if it not already active), and therefore will shine on subsequently declared geometry.

EXAMPLE:

```
Illuminate 13 1
```

Chapter 4

Geometric Primitives

4.1 Primitive Overview

Entropy supports several types of geometric primitives:

- Polygons — objects made of flat facets.
- Parametric patches — curved rectilinear parametric surfaces defined as bilinear, bicubic, or NURBS patches.
- Subdivision surfaces — curved surfaces defined by a control hull and refinement rules.
- 1-D curves and 0-D points — ideal for describing hair, fur, and small particles.
- Quadrics — simple shapes such as spheres and cylinders.
- Implicit ("blobby") primitives.

All of the primitive types except for quadrics define their shape in terms of a collection of *control points*, passed on their parameter lists as `"P"` or `"Pw"` data. `"P"` values are ordinary 3D points defining the control hull of a nonrational surface; `"Pw"` values are 4D homogeneous points defining a rational surface. The quadrics are defined in canonical positions and do not need `"P"` or `"Pw"` values.

In addition to the positional information required for most primitives, the parameter lists may also contain user-supplied data that will be passed along to the shader, overriding shader parameter data of the same name passed through shader assignments. All primitives allow this arbitrary data to have one of several *storage classes*:

`constant` A single data value is supplied for the entire geometric primitive.

`vertex` The number of data elements is identical to the number of control points (i.e., the length of the `"P"` or `"Pw"` arrays), and the values are interpolated across the primitive in the same manner as the position `"P"` (that is, bicubically or linearly or as NURBS, etc., depending on the primitive type).

In addition, some primitives can also accept data with storage class `uniform`, `varying`, and `facevarying` (which have meanings specific to each primitive).

`uniform` One value per face, per subpatch, or per curve, and whose value does not change over each face/subpatch/curve. For quadrics, `uniform` is identical to `constant`.

`varying` One value per face or subpatch corner, or per curve end, and whose value is linearly or bilinearly interpolated over that subpatch. Note that for polygons and quadrics, `varying` is identical to `vertex`.

NEW! `facevarying` A special case for polygon and subdivision meshes, for which the values are specified in the same order as the vertex index array – in other words, allowing for multiple values at each vertex, one per face that borders the vertex.

Data supplied as `constant` will replace the shader parameter default, regardless of whether the shader parameter was defined as `uniform` or `varying` (the value will be replicated if necessary), and data supplied as `uniform` will do so on a face-by-face basis. Data supplied as `vertex`, `varying`, or `facevarying` may only be used to override shader parameters that were declared as `varying`. It is an error to pass `vertex`, `varying`, or `facevarying` data when a `uniform` shader parameter was expected.

For the majority of primitives that do not have special meanings for these classes, `uniform` is identical to `constant`, and `varying` and `facevarying` are identical to `vertex`. Even for primitives where they are meaningful, `varying` and `uniform` primitive variables are confusing and rarely useful; we advise avoiding them and using only `vertex` or `constant`.

To recap, it may help to think of `constant` as "per primitive," `vertex` as "per control point," `uniform` as "per piece" (for certain primitives that are made from multiple pieces), and `varying` as "linearly interpolating across pieces" (with `facevarying` being a slight modification of that for some primitives). We apologize for the awkward terminology, especially the confusing similarity to the entirely separate concept of `uniform` and `varying` values in shaders.

It is also possible to use the parameter lists to override certain geometric values or values from the attribute state — in particular, `"Cs"` (the surface color ordinarily set by `Color`), `"Os"` (the surface opacity set by `Opacity`), `"s"` and `"t"` (the texture coordinates), and `"N"` (the shading normal). For example, here is the declaration of a triangle that attaches user-supplied normals, such as might be interpolated on a polygonal mesh:

```
Polygon "P" [0 0 0 1 0 0 .5 2 0]
        "vertex normal N" [-0.2 -0.2 1 0 .25 1 0 0 1]
```

The result will be that the renderer interpolates the `"N"` values across the surface before supplying them to the shader as variable `N`, giving the appearance of a smoothed surface instead of a faceted one.

The remainder of this chapter presents the commands for all of the geometric primitive types, plus the facilities for making procedural primitives.

4.2 Polygons and Polygon Meshes

Polygon *...parameterlist...*

Create a single simple, closed, convex polygon. The *parameterlist* is a list of token-value pairs that must contain position data (`"P"`) and may optionally contain other primitive variables. The number of vertices comprising the polygon is determined by the number of points contained in the `"P"` data.

EXAMPLE:

```
Polygon "P" [0 0 0   1 0 0   .5 2 0]
```

GeneralPolygon `[`*nverts*`]` *...parameterlist...*

Create a single polygon that may be concave and have multiple loops. If there is more than one loop, the first loop is the outer boundary of the polygon, and the other loops are "holes." *nverts* is an array of integers whose length (number of entries) is the number of loops, and whose data are the number of vertices in each loop.

The *parameterlist* is a list of token-value pairs that must contain position data (`"P"`) and may optionally contain other primitive variables. The length of the vertex point list (`"P"`) should be the sum of the numbers of vertices in all loops.

EXAMPLE:

```
GeneralPolygon [4 3] "P" [0 0 0 0 1 0 1 1 0 1 0 0 .5 .5 0 .75
.75 0 .5 .75 0]
```

The above declares a polygon with a quadrilateral outer boundary and a triangular hole.

PointsPolygons `[`*nverts*`]` `[`*verts*`]` *...parameterlist...*

Create many simple, convex polygons, possibly with shared vertex data. The length of the integer array *nverts* is the number of polygonal faces, and the data in *nverts* is the number of vertices in each face. The integer array *verts*, whose length should be the sum of all entries in *nverts*, contains the vertex indices for each face, in order.

The *parameterlist* is a list of token-value pairs that must contain position data (`"P"`) and may optionally contain other primitive variables. The length of the vertex point list (`"P"`) must be at least as long as the highest vertex index in *verts* plus one. (Indices, like C arrays, begin with 0).

`PointsPolygons` objects will accept primitive variables of class `constant` (one value for the whole mesh), `uniform` (one value per face), `vertex` and `varying` (both taking one value per `"P"` vertex entry), and `facevarying` (similar to `varying`, but given in the same order as the *verts* array – i.e., with each face's vertex values given separately, thus allowing a variable to take on a separate value for each face, even at a shared vertex).

EXAMPLE:

```
PointsPolygons [3 3] [0 1 2 0 2 3] "P" [0 1 0 1 0 0 0 -1 0 -1
0 0]
```

PointsGeneralPolygons [*nloops*] [*nverts*] [*verts*] *...parameterlist...*

Create many polygons, possibly with shared vertex data, and which may be concave and/or have holes. The length of the integer array *nloops* is the number of general polygons, and the data in *nloops* is the number of loops in each polygon. The integer array *nverts*, whose length must be the total number of loops in all polygons (i.e., the sum of all entries in *nloops*), contains the number of vertices in each loop (in polygon order). The integer array *verts*, whose length should be the sum of all entries in *nverts*, contains the vertex indices for each loop, for each face, in order.

The *parameterlist* is a list of token-value pairs that must contain position data (`"P"`) and may optionally contain other primitive variables. The length of the vertex point list (`"P"`) must be at least as long as the highest vertex index in *verts* plus one. (Indices, like C arrays, begin with 0.)

`PointsGeneralPolygons` objects will accept primitive variables of class `constant` (one value whole mesh), `uniform` (one value per face), `vertex` and `varying` (both taking one value per `"P"` vertex entry), and `facevarying` (similar to `varying`, but given in the same order as the *verts* array – i.e., with each face's vertex values given separately, thus allowing a variable to take on a separate value for each face, even at a shared vertex).

EXAMPLE:

```
    PointsGeneralPolygons [1 1] [3 3] [0 1 2 0 2 3] "P" [0 1 0 1 0
0 0 -1 0 -1 0 0]
```

4.3 Patches, Meshes, and NURBS

Basis *ubasisname ustep vbasisname vstep*
Basis [*ubasismatrix*] *ustep* [*vbasismatrix*] *vstep*

This routine does not create a geometric primitive, but rather sets attributes specifying the u and v basis matrices for cubic `Patch`, `PatchMesh`, and `Curves` primitives, and the u and v step sizes for cubic `PatchMesh` and `Curves` primitives. The basis matrices and steps are ordinary attributes and may be saved and restored with `AttributeBegin` and `AttributeEnd`.

The basis matrices are either specified by name (one of `"bezier"`, `"bspline"`, `"catmull-rom"`, or `"hermite"`), or as a matrix of 16 floats. The step sizes are integers and should be 3 for Bezier, 1 for B-spline or Catmull-Rom, or 2 for Hermite.

EXAMPLE:

```
    Basis "bezier" 3 "bezier" 3
```

Patch *type ...parameterlist...*

Creates a single bilinear patch (if the string *type* is `"bilinear"`) or bicubic patch (if *type* is `"bicubic"`). In the case of a bicubic patch, the bicubic basis will be the one

specified in the attribute state by the `Basis` function. A `Patch` is parameterized by u and v parameters each ranging from 0 to 1.

The *parameterlist* is a list of token-value pairs that must contain control vertex positions, and may optionally contain other primitive variables that will be interpolated and made available to shaders. The control vertex positions may be given as ordinary 3-points (`"P"`), or as homogeneous 4-D coordinates (`"Pw"`). A bilinear patch requires 4 (2×2) control vertex positions, while a bicubic patch requires 16 (4×4) control vertex positions.

Primitive `vertex` variables require the same number of data entries as the control vertex positions (4 for bilinears, 16 for bicubics), `varying` variables require 4 data entries and are interpolated bilinearly across the patch (even if it is a bicubic patch), and `uniform` and `constant` variables both require a single data entry.

EXAMPLE:

```
Patch "bilinear" "P" [0 0 0 1 0 0 0 1 0 1 1 0]
```

PatchMesh *type nu uwrap nv vwrap ...parameterlist...*

Creates a rectangular mesh of bilinear or bicubic patches, depending on whether the string *type* is `"bilinear"` or `"bicubic"`. In the case of a bicubic patch mesh, the bicubic basis and the mesh step size will be the ones specified in the attribute state by the `Basis` function. A `PatchMesh` is parameterized by u and v parameters each ranging from 0 to 1.

The *uwrap* and *vwrap* are strings taking the value of either `"nonperiodic"` or `"periodic"`, indicating whether or not the u and v directions wrap all the way around to form a continuous ring.

The *parameterlist* is a list of token-value pairs that must contain control vertex positions, and may optionally contain other primitive variables that will be interpolated and made available to shaders. The control vertex positions may be given as ordinary 3D points (`"P"`), homogeneous 4D coordinates (`"Pw"`), or 1D z values (`"Pz"`). In the case of 1D `"Pz"` values, the corresponding x and y values in `"object"` space will be equal to the patch's (u, v) coordinates.

The control vertices form a rectangular array of $nu \times nv$ points. The number of individual patches forming the mesh can be computed based on *nu*, *nv*, and the basis steps:

$$\text{bilinear } nupatches = \begin{cases} nu & \text{if } uwrap = \text{periodic} \\ nu - 1 & \text{if } uwrap = \text{nonperiodic} \end{cases}$$

$$\text{bicubic } nupatches = \begin{cases} \left(\frac{nu}{nustep}\right) & \text{if } uwrap = \text{periodic} \\ \left(\frac{nu-4}{nustep}\right) + 1 & \text{if } uwrap = \text{nonperiodic} \end{cases}$$

The analogous computation for v yields *nvpatches*. The number of *patch corners* can then be computed:

uwrap	*vwrap*	*ncorners*
`"nonperiodic"`	`"periodic"`	$(nupatches + 1) \cdot nvpatches$
`"nonperiodic"`	`"nonperiodic"`	$(nupatches + 1) \cdot (nvpatches + 1)$
`"periodic"`	`"nonperiodic"`	$nupatches \cdot (nvpatches + 1)$
`"periodic"`	`"periodic"`	$nupatches \cdot nvpatches$

Primitive `vertex` variables require the same number of data entries as the control vertex positions ($nu \times nv$); `varying` variables require one data entry per patch corner (*ncorners*); `uniform` variables require one data entry per patch ($nupatches \times nvpatches$); and `constant` variables require a single data entry.

EXAMPLE:

```
    PatchMesh "bicubic" 2 "nonperiodic" 1 "nonperiodic" "P" [-1 -0.7
0.7 -0.33 -0.7 -0.33 -1 -0.1 -0.33 -0.33 -0.1 -0.33 ]
```

NuPatch *nu uorder* [*uknot*] *umin umax*
nv vorder [*vknot*] *vmin vmax ...parameterlist...*

Create a NURBS (non-uniform rational B-spline) mesh. The integer value *nu* gives the number of points making up the control hull in the u direction. The integer *uorder* is the order (i.e., the degree of the polynomial plus one) of the patch in the u direction. The float array *uknot* is the knot vector for the u direction, and should have length $nu + uorder$. The *umin* and *umax* (both floating point numbers) give the u parametric boundaries of the patch. The *nv*, *vorder*, *vknot*, *vmin*, and *vmax* are the analogous parameters describing the v direction of the patch mesh.

The *parameterlist* is a list of token-value pairs that must contain control vertex positions, and may optionally contain other primitive variables that will be interpolated and made available to shaders. The control vertex positions may either be given as ordinary 3D points (`"P"`) denoting a nonrational patch, or as homogeneous 4D coordinates (`"Pw"`) denoting a rational patch. The total number of control vertices must be $nu \times nv$.

A `NuPatch` may also be thought of as being composed of $(nupatches) \times (nvpatches)$ individual patches, where $nupatches = (1 + nu - uorder)$ and $nvpatches = (1 + nv - vorder)$.

Any `vertex` primitive variables should have $nu \times nv$ entries; `varying` primitive variables should have $(nupatches + 1) \times (nvpatches + 1)$ entries (one per "subpatch corner"; `uniform` variables should have $(nupatches) \times (nvpatches)$ entries (one per "subpatch"); and `constant` variables should have just one data item.

If there is a *trim curve* in effect in the attribute state at the time that a `NuPatch` is declared, parts of the patch mesh may be trimmed when the patch is rendered.

EXAMPLE:

```
    NuPatch 2 2 [0 0 1 1] 0 1 2 2 [0 0 1 1] 0 1 "P" [-3 -1 4 -3 1
4 -1 -1 4 -1 1 4]
```

TrimCurve *ncurves order knot min max n u v w*

Sets the trim curve, which determines which parts of any subsequently-declared `NuPatch` primitives are trimmed. The trim curve is comprised by a number of closed NURBS curve loops.

The number of closed loops is equal to the length of the integer array *ncurves*, whose entries are the number of curve segments in each loop. The integer array *order*, whose length is the total number of curves (i.e., the sum of all entries in *ncurves*), contains the order (polynomial degree + 1) of each curve. The floating-point array *knot* contains the knot vectors of all the curves, in order. The floating point arrays *min* and *max* contain the parametric minimum and maximum value for each of the curves, respectively, in order. The integer array *n* contains, for each curve, the number of control points comprising the curve. And the floating-point arrays *u*, *v*, and *w* contain the homogeneous parametric coordinates (on the `NuPatch`) of all the control vertices of all the NURBS curve segments. The meanings of all these parameters is analogous to the similarly named parameters of `NuPatch`.

Note that `TrimCurve` is an attribute, and may be saved and restored along with the rests of the attribute state with `AttributeBegin` and `AttributeEnd`.

EXAMPLE:

```
    TrimCurve [1] [4] [-.75 -.5 -.25 0 .25 .5 .75 1 1.25 1.5 1.75
] [0] [1] [7] [.9 .5 .01 .5 .9 .5 .01] [.5 .01 .5 .9 .5 .01 .5] [1
1 1 1 1 1 1]
```

4.4 Subdivision Surfaces

SubdivisionMesh *scheme* [*nverts*] [*vertices*]
[*tags*] [*nargs*] [*intargs*] [*floatargs*] *...parameterlist...*

Create a subdivision surface mesh. The *scheme* is a string specifying the name of the subdivision method (currently only `"catmull-clark"` is recognized). Much like `PointsPolygons`, the length of the integer array *nverts* is the number of faces, and the data in *nverts* is the number of vertices in each face. The integer array *vertices*, whose length should be the sum of all entries in *nverts*, contains the vertex indices for each face, in order.

The *parameterlist* is a list of token-value pairs that must contain position data (`"P"`) and may optionally contain other primitive variables. The length of the vertex point list (`"P"`) must be at least as long as the highest vertex index in *verts* plus one. (Indices, like C arrays, begin with 0.)

`SubdivisionMesh` objects will accept primitive variables of class `constant` (one value for whole mesh), `uniform` (one value per face), `vertex` (one per `"P"` vertex entry, interpolating just like position), `varying` (also one value per vertex, but interpolating linearly across each face), and `facevarying` (similar to `varying`, but

given in the same order as the *verts* array – i.e., with each face's vertex values given separately, thus allowing a variable to take on a separate value for each face, even at a shared vertex).

Faces, edges, and vertices may be tagged with additional properties. The *tags* parameter is an array of strings giving the tag names. The integer array *nargs* has length $ntags \times 2$, and for each tag contains the number of integer arguments, followed by the number of floating-point arguments, for that tag. The integer array *intargs* and the floating-point array *floatargs* contain all of the integer and float arguments, respectively (the length of the *intargs* should be the sum of all the even elements of *nargs*, and the length of the *floatargs* should be the sum of all the odd elements of *nargs*). Tags may include:

`"hole"` Specifies certain faces are holes. The tag has *n* integer arguments, which specify the face numbers that are holes, and no floating-point arguments.

`"crease"` Specifies a chain of edges that form a crease. The tag has *n* integer arguments that specify a list of vertices that make up the crease, and one floating point argument giving the "sharpness" of the crease (larger values are sharper).

`"corner"` Marks vertices as sharp corners. The tag has *n* integer arguments that specify the vertices which are corners, and either *n* or 1 floating-point arguments that specify the "sharpness" of the corners (larger values are sharper). If only one sharpness value is given, all of the designated corners have the same sharpness.

`"interpolateboundary"`
If this tag is present, it indicates that the subdivision surface should interpolate boundary faces all the way to their edges. It requires no integer and no floating-point arguments.

NOTE: Entropy 3.1 only supports the `"hole"` tag, and ignores the other tags. This is expected to be fixed in a future release.

EXAMPLE:

```
    SubdivisionMesh "catmull-clark" [4 4 4 4 4 4] [0 2 3 1 4 6 7 5
5 1 3 4 2 0 7 6 6 4 3 2 1 5 7 0] [] [] [] [] "P" [ 25 -25 -25 25 25
-25 25 -25 25 25 25 25 -25 25 25 -25 25 -25 -25 -25 25 -25 -25 -25]
```

4.5 Curves and Points

Curves *type* `[`*nvertices*`]` *wrap* *...parameterlist...*

Draws a number of curve primitives, which may appear as tubes (like hair) or ribbons.

The *type* is a string, either `"linear"` or `"cubic"`, indicating whether the individual curves are piecewise linear or piecewise cubic. Piecewise cubic curves use the v basis matrix set by `Basis`. The integer array *nvertices* has length equal to the total

number of individual curves, and its data are the number of vertices in each curve. The string *wrap* is either `"periodic"` or `"nonperiodic"`, describing whether or not the individual curves wrap end-to-end.

The *parameterlist* is a list of token-value pairs that must contain position data (`"P"`) and may also contain other primitive variables. The total number of control points `"P"` must be the total vertices in all the curves (i.e., the sum of all entries in the *nvertices* array).

Like patch meshes, curves obey the basis matrix and basis step. Therefore, the number of segments in an individual curve with *nv* vertices, and therefore the number of varying data items on that curve, is

$$nsegments = \begin{cases} nv-1 & \text{for nonperiodic linear curves} \\ nv & \text{for periodic linear curves} \\ (\frac{nv-4}{vstep})+1 & \text{for nonperiodic cubic curves} \\ \frac{nv}{vstep} & \text{for periodic cubic curves} \end{cases}$$

$$nvarying = \begin{cases} nsegments+1 & \text{for nonperiodic curves} \\ nsegments & \text{for periodic curves} \end{cases}$$

Any `vertex` primitive variables must have the same number of data entries as the number of points (i.e., they must have the same length as `"P"`). Any `uniform` primitive variables contain one data entry per individual curve. Any `varying` primitive variables have one entry for every segment boundary, that is, a total of $\sum nvarying_i$. Any `constant` primitive variables have only one data item.

If the *parameterlist* contains a primitive variable `"width"`, of type `varying float`, the width values (one per curve segment end) will be used as the `"object"` space sizes of the particles. Alternately, if there is a primitive variable `"constantwidth"`, of type `constant float`, the single value supplied will be used as the `"object"` space sizes of all the particles. If neither `"width"` nor `"constantwidth"` are supplied, particles will all be 1 unit in diameter (measured in `"object"` space).

`Curves` primitives actually produce ribbons. The `"P"` control points obviously define the "spline" of the curve. If primitive variable normals (`"N"`) are supplied, these normals are used to orient the ribbon surface to always be perpendicular to `N`. If no `"N"` values are supplied, the curves are oriented "toward the camera" (or toward the ray origin for ray tracing), so they will appear as thin tubes like hair or spaghetti.

EXAMPLE:

```
    Curves "cubic" [4] "nonperiodic" "P" [ 0 1 0 -1 0 0 -1 -1 0 0
-1 0] "varying float width" [.1 .04]
```

Points *...parameterlist...*

Draws point-like particles. The *parameterlist* is a list of token-value pairs that must contain position data (`"P"`) and may also contain other primitive variables. The total number of particles is determined by the length of the `"P"` array.

If the *parameterlist* contains a primitive variable `"width"`, of type `vertex float`, the width values (one per point) will be used as the `"object"` space sizes of the particles. Alternately, if there is a primitive variable `"constantwidth"`, of type `constant float`, the single value supplied will be used as the `"object"` space sizes of all the particles. If neither `"width"` nor `"constantwidth"` are supplied, particles will all be 1 unit in diameter (measured in `"object"` space).

EXAMPLE:

```
Points "P" [0 0 0  1 0 0  1 1 0  0 1 0] "constantwidth" [0.1]
```

4.6 Quadrics

Six quadrics, plus the torus, are supported as geometric primitives. The quadrics all share several important properties. Most notably, unlike all of the previously-described primitives, quadrics are not described by a mesh of control vertices, and therefore do not require `"P"` values to be supplied. Rather, each quadric is defined parametrically, using trigonometric equations that sweep it out as a function of two parameters.

The quadrics are all created by sweeping a curve around the z-axis in its local coordinate system, so z is always "up." The sweep angle, *thetamax*, is given in degrees (360 being a closed, fully-swept shape). A $thetamax < 0$ creates a quadric that is inside-out; The quadrics all have simple controls for sweeping a partial quadric, using ranges of z or the parametric angles. Quadrics are defined relative to their `"object"` space coordinate systems, and are placed by using a transformation, since they have no built-in translation or rotational controls.

Quadric parameter lists are used solely for applying primitive variables, and so do not affect the shape of the primitives. All quadrics require four data values for any `vertex` or `varying` parameters (one for each parametric corner) and one data value for any `uniform` or `constant` parameter.

The quadric primitives are illustrated in Figure **??**. The individual quadric API routines are explained below.

Cone *height radius thetamax ...parameterlist...*

Creates a cone with an open base on the *x-y* plane and apex at $(0, 0, height)$. The equation of the surface is:

$$\begin{aligned} \theta &= u \cdot thetamax \\ x &= radius \cdot (1 - v) \cdot \cos\theta \\ y &= radius \cdot (1 - v) \cdot \sin\theta \\ z &= v \cdot height \end{aligned}$$

EXAMPLE:

```
Cone 3.0 1.0 360.0
```

Cylinder *radius zmin zmax thetamax ...parameterlist...*

Creates a cylinder with the given *radius*. The cylinder is parallel to (and centered upon) the z-axis and extends from $z = zmin$ to $z = zmax$. The equation of the surface is:

$$\begin{aligned} \theta &= u \cdot thetamax \\ x &= radius \cdot \cos\theta \\ y &= radius \cdot \sin\theta \\ z &= v \cdot (zmax - zmin) \end{aligned}$$

EXAMPLE:

```
Cylinder 1.0 -0.5 1.2 360.0
```

Disk *height radius thetamax ...parameterlist...*

Creates a disk parallel to the *x-y* plane with $z = height$. The equation of the surface is:

$$\begin{aligned} \theta &= u \cdot thetamax \\ x &= radius \cdot (1 - v) \cdot \cos\theta \\ y &= radius \cdot (1 - v) \cdot \sin\theta \\ z &= height \end{aligned}$$

EXAMPLE:

```
Disk 0 2 360
```

Hyperboloid *x1 y1 z1 x2 y2 z2 thetamax ...parameterlist...*

Create a hyperboloid by sweeping the line segment joining points (x_1, y_1, z_1) and (x_2, y_2, z_2) about the z-axis with the given sweep angle *thetamax*.

The hyperboloid is actually quite a flexible superset of some of the other primitives. For example, if these points have the same x- and y-coordinates, and differ only in z, this will create a cylinder. If the points both have the same z coordinate, it will make a planar ring (a disk with a hole cut out of the center). If the points are placed so that they have the same angle with the x-axis (in other words, are on the same radial line if looked at from the top), they will create a truncated cone. In truth, some of these special cases are more useful for geometric modeling than the general case that creates the "familiar" hyperboloid shape.

The equation of the surface is:

$$\begin{aligned} \theta &= u \cdot thetamax \\ x_r &= (1 - v)x_1 + v \cdot x_2 \\ y_r &= (1 - v)y_1 + v \cdot y_2 \\ z_r &= (1 - v)z_1 + v \cdot z_2 \end{aligned}$$

$$\begin{aligned} x &= x_r \cdot \cos\theta - y_r \cdot \sin\theta \\ y &= x_r \cdot \sin\theta + y_r \cdot \cos\theta \\ z &= z_r \end{aligned}$$

EXAMPLE:

```
Hyperboloid 1 0 0 0 0.5 2 360
```

Paraboloid *topradius zmin zmax thetamax ...parameterlist...*

Creates a partial paraboloid swept around the z-axis. The paraboloid is defined as having its minimum at the origin and has radius *topradius* at height *zmax*, and only the portions above *zmin* are drawn. The equation of the surface is:

$$\begin{aligned} \theta &= u \cdot thetamax \\ z &= v \cdot (zmax - zmin) \\ r &= topradius \cdot \sqrt{z/zmax} \\ x &= r \cdot \cos\theta \\ y &= r \cdot \sin\theta \end{aligned}$$

EXAMPLE:

```
Paraboloid 3.0 0 6 360
```

Sphere *radius zmin zmax thetamax ...parameterlist...*

Creates a partial or full sphere with the given *radius*, centered at the origin. The *zmin* and *zmax* parameters can cut off the top and bottom of the sphere if they are not equal to $\pm radius$. The equation of the surface is:

$$\begin{aligned} \phi_{min} &= \begin{cases} \text{asin}\left(\frac{zmin}{radius}\right) & \text{if} \quad zmin > -radius \\ -90.0 & \text{if} \quad zmin \leq -radius \end{cases} \\ \phi_{max} &= \begin{cases} \text{asin}\left(\frac{zmax}{radius}\right) & \text{if} \quad zmax < radius \\ 90.0 & \text{if} \quad zmax \geq radius \end{cases} \\ \phi &= \phi_{min} + v \cdot (\phi_{max} - \phi_{min}) \\ \theta &= u \cdot thetamax \\ x &= radius \cdot \cos\theta \cdot \cos\phi \\ y &= radius \cdot \sin\theta \cdot \cos\phi \\ x &= radius \cdot \sin\phi \end{aligned}$$

EXAMPLE:

```
Sphere 2.0 -2.0 2.0 360
```

Torus *majorradius minorradius phimin phimax thetamax ...parameterlist...*

Creates a quartic "donut" surface (technically not a quadric). The cross section of a torus is a circle of radius *minorradius* on the *x*-*z* plane, and the angles *phimin* and *phimax* define the arc of that circle. It will be swept around z at a distance of *majorradius* to create the torus. Thus, $majorradius + minorradius$ defines the outside radius of the entire torus (its maximum size), while $majorradius - minorradius$ defines the radius of the hole. The equation of the surface is:

$$\begin{aligned} \theta &= u \cdot thetamax \\ \phi &= phimin + (phimax - phimin) \\ r &= minorradius \cdot \cos\phi \\ z &= minorradius \cdot \sin\phi \\ x &= (majorradius + r) \cdot \cos\theta \\ y &= (majorradius + r) \cdot \sin\theta \end{aligned}$$

EXAMPLE:

```
Torus 3 0.5 0 360 360
```

4.7 Implicit Surfaces

Blobby `[` *code* `]` `[` *floats* `]` `[` *strings* `]` *...parameterlist...*

NEW!

The `Blobby` documentation is currently incomplete. Sorry about that.

Entropy's `Blobby` primitives are incompletely implemented. Currently, they only support ellipsoid blobs (not segments or planes), and only support the add, multiply, max, min, and subtract operations.

EXAMPLE:

```
Blobby 2 [1001 0 1001 16 0 2 0 1]
[0.5 0 0 0 0 0.5 0 0 0 0 0.5 0 0.6 0.1 0.05 1
0.65 0 0 0 0 0.64 0 0 0 0 0.63 0 0.81 0.12 0.74 1] [ ]
"varying color Cs" [1 0.3 0.389422 0.2 1 0.378140]
```

4.8 Procedural Geometry

A *procedural primitive* is a collection of geometry for which you only tell the renderer its location and spatial extent as a bounding box. Only if (and when) the renderer needs to know the what is inside the box, the renderer will invoke a procedure (such as running a program or reading another file) to generate the geometry inside the box.

Extensive use of procedural primitives is an excellent way to reduce the memory consumption of very large scenes. Procedurals that are outside the camera view or occluded by

other objects may never get expanded, and thus use almost no memory whatsoever. Even those procedurals that do get expanded will delay doing so until they are absolutely needed, further reducing the "working set" of geometry needed at any time by the renderer.

Procedural *procname* [*procargs*] [*boundingbox*]

Adds a "procedural primitive" to the scene. The string parameter *procname* is the type of procedural primitive. The *procargs* parameter is an array of strings whose meaning is specific to the type of procedural. The *boundingbox* (expressed as an array of six floats: *xmin*, *xmax*, *ymin*, *ymax*, *zmin*, *zmax*) describes the spatial extent of the procedural primitive, relative to the CTM coordinate system.

There are three procedural types supported by Entropy:

Procedural "DelayedReadArchive" [*filename*] [*boundingbox*]

The DelayedReadArchive procedural takes a single string array argument: the *filename* of a RIB file containing the geometry comprising the procedural primitive. When the contents of the procedural's bounding box are needed, the file will be read to provide the geometry to the renderer.

Any time you are inclined to use ReadArchive, you should try instead to use DelayedReadArchive if it is at all possible to bound the contents of the file.

EXAMPLE:

```
Procedural "DelayedReadArchive" ["chair.rib"] [-20 20 -30 30 0
100]
```

Procedural "RunProgram" [*programname datablock*] [*boundingbox*]

The RunProgram procedural takes a two arguments in its array of string parameters: the *programname* is the name of an executable program to run (including any command line arguments), and the *datablock* is a string giving arguments or other data that is meaningful to the procedural program (it can be an empty string if the program does not require any further information). When the contents of the procedural's bounding box are needed, the program will be run with the given arguments, providing the geometry to the renderer.

EXAMPLE:

```
Procedural "RunProgram" ["makechair -height 30" ""] [-20 20 -30
30 0 100]
```

Procedural "DynamicLoad" [*dsoname paramdata*] [*boundingbox*]

The DynamicLoad procedural takes a two arguments in its array of string parameters: the *dsoname* is the name of a shared object file (called DSO's or DLL's, depending on your operating system), and *paramdata* is a string giving arguments or other data that is meaningful to the procedural program. When the contents of the procedural's bounding box are needed, the program will be run with the given arguments, providing the geometry to the renderer.

Procedural DSO's must have three public interface routines:

```
void * ConvertParameters (const char * initialdata);
void Subdivide (void * blinddata, float detailsize);
void Free (void * blinddata);
```

The `ConvertParameters` routine takes a pointer to the *paramdata*, creates any internal representation needed for the procedural, and returns a `(void *)` blind pointer to the procedural data. The `Subdivide` routine, which is only called if the procedural needs to be expanded, takes that same blind pointer that was created by `ConvertParameters`. The `Subdivide` routine is responsible for generating the contents of the procedural (using the C API for describing scenes). Finally, the `Free` routine, which also takes the same blind pointer, is called at the end of the frame, and should free any resources associated with the procedural.

EXAMPLE:

```
Procedural "DynamicLoad" ["chair.so" ""] [-20 20 -30 30 0 100]
```

Because the whole point of procedurals is that the renderer does not need to know the contents of the procedural until exactly when it is needed, it is impossible for the renderer to know about any attributes set inside the procedural. For ray tracing, this is very important — the `Attribute "visibility"` (see Section 3.2.2, p. 32) determines whether or not primitives appear at all in reflections or shadows. For efficiency reasons, you don't want the renderer to have to fully expand the procedural just to find out if anything inside will require ray tracing.

Therefore, `Attribute "visibility" "reflection"` or `Attribute "visibility" "shadow"` must be set when the procedural is declared (i.e., as if the procedural itself was a primitive binding to the attribute state), or Entropy will assume that nothing inside the procedural will be needed for ray tracing. It's okay if the contents of the procedural turn reflection or shadow visibility off again — but if anything inside will be visible in ray tracing, the entire procedural must be marked as potentially visible in traced rays.

Chapter 5

Shading Language

This chapter gives a terse, yet complete, specification for Entropy's shading language.

5.1 Preliminaries

5.1.1 Shader Types

There are four types of shaders supported by Entropy:

surface
: Surface shaders describe the appearance of surface geometry as viewed by the camera or as it appears in a reflection or shadow ray. Surface shaders are bound to geometry with the `Surface` call.

displacement
: Displacement shaders can actually deform geometry in order to add surface detail. Displacement shaders are bound to geometry with the `Displacement` call.

volume
: Volume shader describe how light is attenuated as it passes through volumes of participating media. Volume shaders may be bound to geometry through the `Atmosphere`, `Interior`, or `Exterior` calls.

light
: Light shaders determine how much light from a particular light source arrives at a point in space. Light shaders are bound to lights in the scene using the `LightSource` and `AreaLightSource` calls.

5.1.2 Overall shader structure

A shader is organized as follows:

[optional function definitions]

shadertype shadername ([*params*])

{
    statements
}

The *shadertype* is one of `surface`, `displacement`, `volume`, `light`, or `imager`. The *shadername* is the name of the shader, chosen by the user. The optional *params* is a list of parameter declarations.

type paramname = *defaultvalue* { , *paramname* = *defaultvalue* } { ; *moreparams* }

The *type* is one of the shading language data types (described in Section 5.2). The *paramname* is the name of the parameter, and it is given a *defaultvalue*, which will be the value of the parameter if another value is not supplied in the scene file. Several parameters of the same type may be declared together, separated by commas. Several parameter sets of different type may be declared in the parameter list, separated by semicolons. Below is an example shader declaration:

```
surface pitted ( float Ka=1, Kd=1, Ks=0.5;
                 float angle = radians(30);
                 color splotcolor = 0;
                 color stripecolor = color (.5, .5, .75);
                 string texturename = "";
                 string dispmapname = "mydisp.tx";
                 vector up = vector "shader" (0,0,1);
                 varying point Pref = point (0,0,0);
                 output varying normal surfNorm = 0;
               )
{
  ...
}
```

The storage class (see Section 5.2.6) for shader parameters is assumed to be `uniform`, but you may use the `varying` keyword to force a parameter to be `varying`. You must declare a parameter as `varying` if you intend to pass `varying` or `vertex` data on the geometric primitive for that variable.

Shader parameters are treated as read-only — your shader may not modify their values — unless you declare them using the `output` keyword. Declaring a parameter as `output` makes it writable, and by writing new values your shader can communicate with other shaders attached to the same geometry or specify new data to be incorporated into the image output.

Prior to the definition of the shader itself, there may be any number of functions (subroutines) declared. Those functions may then be called by the shader.

5.1.3 Identifiers

Identifiers, used for names of variables, functions, and shaders, consist of one or more letters, numerals, or the underscore character. The first character of an identifier may not be a digit.

The following are valid identifiers in the shading language:

```
i
Foo19
_bar
really_long_name
```

The following are not valid identifiers:

`7seas`	(can't start with a numeral)
`wow!`	(can't include punctuation)
`my variable`	(definitely can't contain spaces!)

Identifiers are *case sensitive*; in other words, the identifiers `index` and the variable `Index` are completely different variables.

5.1.4 Comments

As in C, any text enclosed by `/*` and `*/` is considered a comment, and will be ignored. As with C++, any text following `//`, until the end of the line, is considered a comment and will be ignored.

NEW!

5.1.5 Preprocessor

Shaders are assumed to be filtered by a preprocessor, just like C programs. The following preprocessor directives are supported and perform the same functionality as in the C language:

```
#include
#define
#ifdef
#ifndef
#if
#endif
#else
```

5.2 Data Types

Entropy's shading language provides several built-in data types for performing computations inside your shader, as shown in Table 5.1.

Table 5.1: Shading Language data types.

`float`	Scalar floating-point data (numbers)

`point`
`vector`
`normal`
	Three-dimensional positions, directions, and surface orientations

`color`	Spectral reflectivities and light energy values
`matrix`	4×4 transformation matrices
`string`	Character strings (such as filenames)

5.2.1 float

The `float` type is used for all scalar numeric values (including integer values), and is nearly identical to the equivalent data type in the C language. `float` constants are constructed the same way as in C, for example: `1`, `1.14`, `1.0e-6`. The named constant `PI` is predefined to be the value to π (to the accuracy of the `float` data type).

5.2.2 color

The `color` data type is used to represent 3-component spectral reflectivities and light energies (such as radiance).[1] The components of colors are referent to a particular *color space*. Colors are by default represented as RGB triples `"rgb"` space. You can assemble a color out of three floats, either representing an RGB triple or some other color space known to the renderer, for example:

```
color (0, 0, 0)             /* black */
color "rgb" (.75, .5, .5)   /* pinkish */
color "hsv" (.2, .5, .63)   /* specify in "hsv" space */
```

All three of these expressions above return colors in `"rgb"` space. Even the third example returns a color in `"rgb"` space — specifically, the RGB value of the color that is equivalent to hue 0.2, saturation 0.5, and value 0.63. In other words, when assembling a color from components given relative to a specific color space in this manner, there is an implied transformation to `"rgb"` space. Table 5.2 lists the color spaces that Entropy knows by name.

The `+`, `-`, `*`, and `/` operators can be applied to two colors, performing the operations component-by-component. Colors may be compared using the `==` and `!=` boolean operators. All of these operations may be performed between a `color` and a `float`, treating the `float` as if it were a `color` with all three components being identical. Section **??** lists built-in functions that manipulate colors, including functions to retrieve and set individual color components and that transform colors among different color spaces.

[1] Note that reflectivities are only physically meaningful of each component is between 0 and 1, whereas radiance values may have unbounded positive values. Entropy makes no attempt to police either the values you store in `color`'s or your semantic use of reflectivity versus radiance.

Table 5.2: Names of color spaces.

`"rgb"`	The coordinate system that all colors start out in, and in which the renderer expects to find colors that are set by your shader (such as `Ci`, `Oi`, and `Cl`).
`"hsv"`	hue, saturation, and value.
`"hsl"`	hue, saturation, and lightness.
`"YIQ"`	the color space used for the NTSC television standard.
`"xyz"`	CIE *XYZ* coordinates.
`"xyY"`	CIE *xyY* coordinates.

5.2.3 point, vector, normal

A `point` is a position in 3D space. A `vector` has a length and direction, but does not exist in a particular location. A `normal` is a vector that is perpendicular to a surface, and thus describes the surface's orientation. All three of these types are internally represented by three floating-point numbers.

This manual will often refer to these types collectively as "point-like" types (or, without loss of generality, simply as points). Note that points, vectors, and normals transform between coordinate systems using different transformation rules, so it is important that you choose the right types and use the right transformation routines on them.

All points, vectors, and normals are described relative to some coordinate system. All data provided to a shader (surface information, graphics state, parameters, and vertex data) are relative to one particular coordinate system that we call the `"current"` coordinate system. The `"current"` coordinate system is one that is convenient for the renderer's shading calculations.

You can construct a point-like type out of three floats using a constructor, for example:

```
point (0, 2.3, 1)
vector (a, b, c)
normal (0, 0, 1)
```

These expressions are interpreted as a point, vector, and normal whose three components are the floats given, relative to `"current"` space .

As with colors, you may also specify the coordinates relative to some other coordinate system:

```
Q = point "object" (0, 0, 0);
```

This example assigns to `Q` the point at the origin of `"object"` space. However, this statement does *not* set the components of `Q` to (0,0,0)! Rather, `Q` will contain the `"current"` space coordinates of the point that is at the same location as the origin of `"object"` space. In other words, the point constructor that specifies a space name implicitly specifies a transformation to `"current"` space. This type of constructor also can be used for vectors and normals.

Table 5.3 lists the coordinate systems that Entropy recognizes by name. The RIB statement `CoordinateSystem` may be used to give additional names to user-defined coordinate

systems. These names may also be referenced inside your shader to designate transformations.

The `+`, `-`, `*`, and `/` operators can be applied to two point-like values, performing the operations component-by-component. Point-like values may be compared using the `==` and `!=` boolean operators. All of these operations may be performed between a point (or vector or normal) and a `float`, treating the `float` as if it were a `point` with all three components being identical. The shading language also defines the `^` operator to be cross-product (taking two vector operands and returning a vector) and the `.` operator to be dot-product (taking two vector operands and returning a float).

Table 5.3: Names of predeclared geometric spaces.

`"current"`	The coordinate system that all points start out in and the one in which all lighting calculations are carried out.
`"object"`	The local coordinate system of the graphics primitive (sphere, patch, etc.) that we are shading.
`"shader"`	The coordinate system active at the time that the shader was declared (by the `Surface`, `Displacement`, or `LightSource` statement).
`"world"`	The coordinate system active at `WorldBegin`.
`"camera"`	The coordinate system with its origin at the center of the camera lens, x-axis pointing right, y-axis pointing up, and z-axis pointing into the screen.
`"screen"`	The *perspective-corrected* coordinate system of the camera's image plane. Coordinate (0,0) in `"screen"` space is looking along the z-axis of `"camera"` space.
`"raster"`	The 2D, projected space of the final output image, with units of pixels. Coordinate (0,0) in `"raster"` space is the upper left corner of the image, with x and y increasing to the right and down, respectively.
`"NDC"`	Normalized device coordinates — like raster space, but normalized so that x and y both run from 0 to 1 across the whole image, with (0,0) being at the upper left of the image, and (1,1) being at the lower right (regardless of the actual aspect ratio).

5.2.4 matrix

The `matrix` type represents a matrix required to transform points and vectors between one coordinate system and another. Matrices are represented internally by 16 floats (a 4×4 homogeneous transformation matrix).

A `matrix` can be constructed from a single float or 16 floats. For example:

```
matrix zero = 0;   /* makes a matrix with all 0 components */
matrix ident = 1;  /* makes the identity matrix */

/* Construct a matrix from 16 floats */
matrix m = matrix (m00, m01, m02, m03, m10, m11, m12, m13,
                   m20, m21, m22, m23, m30, m31, m32, m33);
```

Assigning a single floating-point number x to a matrix will result in a matrix with diagonal components all being x and other components being zero (i.e., x times the identity matrix). Constructing a matrix with 16 floats will create the matrix whose components are those floats, in row-major order.

Similar to point-like types, a `matrix` may be constructed in reference to a named space:

```
/* Construct matrices relative to something other than "current" */
matrix q = matrix "shader" 1;
matrix m = matrix "world" (m00, m01, m02, m03, m10, m11, m12, m13,
                           m20, m21, m22, m23, m30, m31, m32, m33);
```

The first form creates the matrix that transforms points from `"current"` space to `"shader"` space. Transforming points by this matrix is identical to calling `transform("shader",...)`. The second form prepends the current-to-world transformation matrix onto the 4×4 matrix with components $m_{0,0}...m_{3,3}$. Note that although we have used `"shader"` and `"world"` space in our examples, any named space is acceptable.

Matrix values can be compared with `==` and `!=`. Also, the `*` operator between matrices denotes matrix multiplication, while `m1 / m2` denotes multiplying `m1` by the inverse of matrix `m2`. Thus, a matrix can be inverted by writing `1/m`.

5.2.5 string

The `string` type may hold character strings. The main application of strings is to provide the names of files where textures may be found. Strings can be compared using `==` and `!=`.

String constants are denoted by surrounding the characters with double quotes, as in `"I am a string literal"`. As in C programs, strings may contain special escape sequences that begin with the backslash (‘\’) character: ‘`\n`’ (newline), ‘`\r`’ (carriage return), ‘`\t`’ (tab), ‘`\\`’ (backslash character), ‘`\"`’ (double quote character).

5.2.6 Storage classes: `uniform` and `varying`

In addition to a data *type,* all variables and values in the shading language have a *storage class*, which may be either `uniform` or `varying`. A `uniform` variable is one whose value is the same at all points on a surface (or at least all the surface points that are shaded at any one time). A `varying` variable is one that can take on different values at different positions on the surface. Expressed another way, `uniform` values are *per object* whereas `varying` values are *per surface point*.

It is important to remember that `uniform` does not mean “read only.” `Uniform` variables are perfectly free to be reassigned or to take different values at different times; they are just prohibited from taking different values at different locations in space.

If no storage class is explicitly specified in a shader parameter declaration, it will be assumed to be `uniform`. If no storage class is specified in a local variable declaration, it will be assumed to be `varying`. Strings may be `uniform` only (even if they are declared as local variables), but all other data types may be either `uniform` or `varying`.

You should declare as many of your parameters and variables as `uniform` as possible — `uniform` values take up much less storage and can be computed much more rapidly than

`varying` values. It is especially helpful (in terms of optimizing your shaders) to be sure that you use `uniform` values and variables for controlling conditionals and loops.

5.3 Language Syntax

The bodies of shaders and shader functions are composed of *statements*. Statements in the shading language are terminated by semicolons (`;`). The types of statements allowed are:

Variable Declaration A local variable may be declared, and optionally, an initial value may be assigned to it.

Assignment A previously declared variable may have its value set to the evaluation of an expression.

Procedure call A previously declared function or a built-in function may be invoked.

Conditional The shading language supports `if` and `if-else` statements, much like the C language.

Loop The shading language supports `for` and `while` loops, much like the C language.

Loop Modifier The `break` statement causes a loop to terminate, and the `continue` statement skips to the next iteration. Both statements may be called only within a loop.

Lighting Statement The shading language allows light shaders to generate light with the `solar` and `illuminate` statements, and for surface shaders to gather light with the `illuminance` statement.

Function Declaration A new function may be declared and defined, able to be called and invoked later within the same lexical scope.

New scope A new naming scope may be declared by enclosing statements with curly braces: { }

5.3.1 Variable Declarations and Assignments

Local variables are those that you, the shader writer, declare for your own use. They are analogous to local variables in C or any other general-purpose programming language. The syntax for declaring a variable in the shading language is (items in brackets are optional):

[*class*] *type variablename* [= *initializer*]

where

- The optional *class* specifies one of `uniform` or `varying`. If *class* is not specified, it defaults to `varying` for local variables.
- *type* is one of the basic data types, described earlier.

- *variablename* is the name of the variable you are declaring.
- If you wish to give your variable an initial value, you may do so by assigning an *initializer*.

Arrays are also supported, declared as follows:

class type variablename `[` *arraylen* `]` = { *init0*, *init1 ...* }

Arrays must have a constant length; they may not be dynamically sized. Also, only 1D arrays are allowed. Other than that, however, the syntax of array usage is largely similar to C.

Some examples of variable declarations are

```
float a;          /* Declare; current value is undefined */
uniform float b;  /* Explicitly declare b as uniform */
float c = 1;      /* Declare and assign */
float d = b*a;    /* Another declaration and assignment */
float e[10];      /* The variable e is an array */
```

When you declare local variables, you will generally want them to be `varying`. But be on the lookout for variables that take on the same value everywhere on the surface (for example, loop control variables), because declaring them as `uniform` may allow Entropy to take shortcuts that allow your shaders to execute more quickly and use less memory.

5.3.2 Procedure Calls

Just like in C and many other programming languages, you may call functions by specifying their name, followed by parentheses, optionally with a comma-separated argument list:

functionname `(` `)`

functionname `(` *arg1* , *...* , *argn* `)`

If the function does not return any value (a `void` function) or if you wish to ignore the return value, a bare procedure call as above is a valid program statement. The function's return value (if it has one) may also be assigned to a variable, used in an expression, or passed as an argument to another function.

5.3.3 Conditionals

Conditionals in Entropy's shading language work much as in C:

`if` `(` *condition* `)`
    truestatement

and

```
if ( condition )
     truestatement
else
     falsestatement
```

The statements can also be entire blocks, surrounded by curly braces. The condition may be one of the following Boolean operators: `==`, `!=` (equality and inequality); $<$, $<=$, $>$, $>=$ (less-than, less-than or equal, greater-than, greater-than or equal). Conditions may be combined using the logical operators: `&&` (and), `||` (or), `!` (not).

Unlike C, in shaders it is not legal to use a `float`, `point`, or `color` directly as the condition (it must be a *comparison* between two values), nor may a boolean expression be assigned to a `float`.

5.3.4 Loops

Two types of loop constructs work nearly identically to their equivalents in C. Repeated execution of statements for as long as a condition is true is possible with a `while` statement:

```
while ( condition )
     truestatement
```

Also, C-like `for` loops are also allowed:

```
for ( init ; condition ; loopstatement )
     body
```

As with `if` statements, loop conditions must be relations, not floats. As with C, you may use `break` and `continue` statements to terminate a loop altogether or skip to the next iteration, respectively.

5.3.5 Lighting Statements

The shading language has some special syntactic structures for emitting light in light shaders, and for gathering light in surface or volume shaders.

Emission of Light: `solar` and `illuminate`

Within light shaders, the `solar` and `illuminate` statements are available to control the emission of light from infinitely far and finite positions, respectively. Both statements set the `L` variable appropriately, and both expect that its statements will set the `Cl` variable to the amount of light arriving at surface position `Ps`.

```
solar ( vector axis ; float spreadangle ) {
     statements ;
}
```

The effect of the `solar` statement is to send light to every `Ps` from the same direction, given by *axis*. The `solar` statement sets the `L` variable to its first argument. The result is that rays from such a light are parallel, as if the source was infinitely far away (like the sun).

The *spreadangle* parameter is usually set to zero, indicating that the source subtends an infinitesimal angle and that the rays are truly parallel. Values for *spreadangle* greater than zero indicate that a plethora of light rays arrive at each `Ps` from a range of directions, instead of a single ray from a particular direction. Such lights are known as *broad solar lights* and are analogous to very distant but very large area lights (for example, the sun actually subtends a 1/2 degree angle when seen from Earth).

For lights that have a definite, finitely close position, there is another construct to use:

```
illuminate ( point from ) {
     statements;
}

illuminate ( point from; vector axis; float angle ) {
     statements;
}
```

The first form of the `illuminate` statement indicates that light is emitted from position `from`, and is radiated in all directions. The `illuminate` statement implicitly sets `L = Ps - from`. The second form of `illuminate` also specifies a particular cone of light emission, given by an axis and angle. If `Ps` does not fall within the cone, the body of the `illuminate` statement will not be executed.

Gathering of light: `illuminance`

Surface and volume shaders may gather available light using another statement: `illuminance`.

```
illuminance ( point position ) {
     statements;
}

illuminance ( point position; vector axis; float angle ) {
     statements;
}
```

The `illuminance` statement loops over all light sources visible from a particular *position*. In the first form, all lights are considered, and in the second form, only those lights whose directions are within *angle* of *axis* (typically, *angle*=$\pi/2$ and *axis*=`N`, which indicates that all light sources in the visible hemisphere from `P` should be considered). For each light source, the *statements* are executed, during which two additional variables are defined: `L` is the vector that points to the light source, and `Cl` is the color representing the incoming energy from that light source.

5.3.6 Scoping and Function Definitions

Even though the shading language provides many useful functions, you will probably want to write your own, just as you would in any other programming language. Defining your own functions is similar to doing it in C:

returntype functionname (*params*)
{
    ... *do some computations* ...
    `return` *return_value* ;
}

However, in many ways shader function definitions are not quite like C:

- Only one `return` statement is allowed per function. The exception to this rule is for `void` functions, which have no `return` statement.
- All function parameters are passed by reference. In other words, unlike C, the function does not have private copies of its parameters that can be modified without affecting their originals. Rather, function parameters are merely new names for the original variables (much like using the `&` reference parameters in C++).
- You may not compile functions separately from the body of your shader. The functions must be declared prior to use and in the same compilation pass as the rest of your shader (though you may place them in a separate file and use the `#include` mechanism).

Valid return types for functions are the same as variable declarations: `float`, `color`, `point`, `vector`, `normal`, `matrix`, `string`. You may declare a function as `void`, indicating that it does not return a value. You may not have a function that returns an array.

Although parameters are passed by reference, any parameters you want to modify must be declared with the `output` keyword, as in the following example:

```
float myfunc (float f;          /* you can't assign to f */
              output float g;) /* but you can assign to g */
```

Entropy expands functions inline, instead of compiling them separately and calling them as subroutines. This means that there is no overhead associated with the call sequence. The downside is increased size of compiled code and the lack of support for recursion.

Functions obey standard variable lexical scope rules. Functions may be declared outside the scope of the shader itself, as you do in C. By default, shader functions may only access their own local variables and parameters. However, this can be extended by use of an `extern` declaration — global variables may be accessed by functions if they are accessed as `extern` variables. You may also define functions inside shaders or other functions — that is, functions may be defined anyplace where a local variable might be declared. In the case of local functions, variables declared in the outer lexical scope may be accessed if they are redeclared using the `extern` keyword. Following is an example:

```
float x, y;

float myfunc (float f)
{
    float x;        /* local hides the one in the outer scope */
    extern float y; /* refers to the y in the outer scope */
    extern point P; /* Refers to the global P */
    ...
}
```

5.4 Expressions

The expressions available in the shading language include the following:

- constants: floating-point (e.g. `1.0`, `3`, `-2.35e4`), string literals (e.g., `"hello"`), and the named constant `PI`
- point, vector, normal, or matrix constructors, for example:
  `point "world" (1,2,3)`
- variable references (by name)
- function calls
- unary and binary operators on other expressions, for example (in order of precedence):

`-` *expr*	(negation)
expr `.` *expr*	(vector dot product)
expr `/` *expr*	(division)
expr `*` *expr*	(multiplication)
expr `^` *expr*	(vector cross product)
expr `+` *expr*	(addition)
expr `-` *expr*	(subtraction)

  The operators `+`, `-`, `*`, `/`, and the unary `-` (negation) may be used on any of the numeric types. For multicomponent types (colors, vectors, matrices), these operators combine their arguments on a component-by-component basis.

  The `^` and `.` operators only work for vectors and normals and represent cross product and dot product, respectively.

  The only operators that may be applied to the `matrix` type are `*` and `/`, which respectively denote matrix-matrix multiplication and matrix multiplication by the inverse of another matrix.

- relations between variables (all lower precedence than the numeric operators):

expr == *expr*	(equal)
expr != *expr*	(not equal)
expr < *expr*	(less than)
expr <= *expr*	(less than or equal to)
expr > *expr*	(greater than)
expr >= *expr*	(greater than or equal to)

The == and != comparisons may be performed between any two values of equal type, and are performed component-by-component for multicomponent types. The relative ordering comparisons (such as < and >) may only be performed between two `float`'s.

Note that relations produce Boolean (true/false) values. Boolean values may not be assigned to variables, but may be used in conditionals and loops.

- logical combinations of Boolean expressions:

expr && *expr*	(and)
expr || *expr*	(or)
! *expr*	(not)

- another expression enclosed in parenthesis: `( )`. Parenthesis can be used to guarantee associativity of operations.
- type casts, specified by simply having the type name in front of the value to cast:

```
vector P            /* cast a point to a vector */
point f             /* cast a float to a point */
color P             /* cast a point to a color! */
```

The three-component types (`point`, `vector`, `normal`, `color`) may be cast to other three-component types. A `float` may be cast to any of the three-component types (by placing the float in all three components) or to a `matrix` (which makes a matrix with all diagonal components being the `float`). Obviously, there are some type casts that are not allowed because they make no sense, like casting a `point` to a `float`, or casting a `string` to a numerical type.

- ternary operator, just like C: *condition* ? *expr1* : *expr2*
  If *condition* is true, *expr1* is returned, but if *condition* is false, *expr2* is returned.

5.5 Global variables

In addition to shader parameters and locally-defined variables, the shading language makes information about the surface available through pre-declared "global" variables. Somewhat different sets of global variables are available in different types of shaders. Table 5.8 lists all of the predefined global variables, with information about how their meanings differ among shader types. The global variables are read-only except where noted as "writable."

5.5.1 Surface shaders

The purpose of surface shaders is to modify the `Ci` and `Oi` variables to be the color and opacity of the surface at the point being shaded. If the shader does not explicitly set `Ci`, it will retain its default of 0 (black). If the shader does not explicitly set `Oi`, it will retain its default value of 1 (fully opaque).

Variable		Description
`varying color`	**Cs**	Surface color
`varying color`	**Os**	Surface opacity
`varying point`	**P**	Surface position (WRITABLE*)
`varying vector`	**dPdu**, **dPdv**	Partial derivatives $\partial P/\partial u$ and $\partial P/\partial v$ of the surface.
`varying normal`	**N**	Surface shading normal (WRITABLE)
`varying normal`	**Ng**	Surface geometric normal
`varying float`	**s**, **t**	Surface texture coordinates
`varying float`	**u**, **v**	Surface parameters
`varying float`	**du,dv**	Change in surface parameters between adjacent shading points
`varying vector`	**L**	Incoming light ray direction (only available inside `illuminance` statements)
`varying color`	**Cl**	Incoming light ray color (only available inside `illuminance` statements)
`varying point`	**E**	Position of the eye
`varying vector`	**I**	Incident ray direction
`uniform float`	**time**	Current shutter time
`uniform float`	**dtime**	The amount of time covered by this shading sample.
`varying vector`	**dPdtime**	How the surface position P is changing per unit time, as described by motion blur in the scene.
`varying color`	**Ci**	Incident ray color (WRITABLE)
`varying color`	**Oi**	Incident ray opacity (WRITABLE)

(*) Writing to P in surface shaders will not result in true displacement, only bump mapping.

Table 5.4: Surface Shader “Global” Variables

5.5.2 Displacement shaders

Displacement shaders have access to most of the same global variables as surface shaders. They may not write to `Ci` or `Oi` or interrogate light sources, but if displacement shaders modify `P`, the surface will deform (displace).

Variable		Description
`varying point`	**P**	Surface position (WRITABLE*)
`varying vector`	**dPdu**, **dPdv**	Partial derivatives $\partial P/\partial u$ and $\partial P/\partial v$ of the surface.
`varying normal`	**N**	Surface shading normal (WRITABLE)
`varying normal`	**Ng**	Surface geometric normal
`varying float`	**s**, **t**	Surface texture coordinates
`varying float`	**u**, **v**	Surface parameters
`varying float`	**du,dv**	Change in surface parameters between adjacent shading points
`varying point`	**E**	Position of the eye
`varying vector`	**I**	Incident ray direction
`uniform float`	**time**	Current shutter time
`uniform float`	**dtime**	The amount of time covered by this shading sample.
`varying vector`	**dPdtime**	How the surface position P is changing per unit time, as described by motion blur in the scene.

(*) Writing to P in displacement shaders will deform the surface.

Table 5.5: Displacement Shader "Global" Variables

5.5.3 Volume shaders

Volume shaders attenuate light as it passes through volumes of participating media. At the beginning of volume shader execution, `Ci` and `Oi` variables contain the color and opacity of whatever was "behind" the volume. The volume shader should modify `Ci` and `Oi` as necessary to reflect what happens to that light as it passes through the volume.

Variable		Description
`varying color`	**Ci**, **Oi**	Color and opacity of light on the far side of the volume.
`varying point`	**P**	Far volume endpoint.
`varying vector`	**I**	Incident ray through the volume.
`varying point`	**E**	Position of the eye
`varying vector`	**L**	Incoming light ray direction (only available inside `illuminance` statements)
`varying color`	**Cl**	Incoming light ray color (only available inside `illuminance` statements)
`uniform float`	**time**	Current shutter time
`uniform float`	**dtime**	The amount of time covered by this shading sample.
`varying color`	**Ci**, **Oi**	Incident light color and opacity as it exits from the near side of the volume.

Table 5.6: Volume Shader "Global" Variables

5.5.4 Light shaders

Light shaders are supposed to set `L` and `Cl` to reflect the direction and color of light arriving at point `Ps`.

Variable		Description
`varying point`	**Ps**	Position of the surface being illuminated
`varying point`	**P**	Position on the light source
`varying vector`	**dPdu**, **dPdv**	Partial derivatives $\partial P/\partial u$ and $\partial P/\partial v$ of the area light source surface
`varying normal`	**N**	Surface shading normal on the light
`varying normal`	**Ng**	Surface geometric normal on the light
`varying normal`	**Ns**	Normal on the surface that caused the light to be run. Be very careful using this – its value is undefined when lights are run at positions other than the surface's `P` (for example, in volume marching).
`varying float`	**s**, **t**	Surface texture coordinates on the light
`varying float`	**u**, **v**	Surface parameters on the light
`varying float`	**du,dv**	Change in surface parameters between adjacent shading points
`varying vector`	**L**	Outgoing light ray direction (only available inside `illuminate` and `solar` statements)
`varying color`	**Cl**	Outgoing light ray color (WRITABLE)
`uniform float`	**time**	Current shutter time
`uniform float`	**dtime**	The amount of time covered by this shading sample.

Table 5.7: Light Shader "Global" Variables

Variable		Description
`varying color`	**Cs**, **Os**	Input surface color and opacity (surface shaders only)
`varying point`	**P**	Surf: surface position Disp: surface position (WRITABLE to displace) Vol: position at far end of the volume Light: position on the area light geometry
`varying vector`	**dPdu**, **dPdv**	Surf/disp: Partial derivatives $\partial P/\partial u$ and $\partial P/\partial v$ of the surface. Light: Partial derivatives of the area light source geometry.
`varying normal`	**N**	Surf/disp: shading normal (WRITABLE) Light: shading normal of the area light geometry
`varying normal`	**Ng**	Surf/disp: true geometric normal Light: true geometric normal of area light geometry
`varying float`	**s**, **t**	Texture coordinates
`varying float`	**u**, **v**	Geometric parameters
`varying float`	**du,dv**	Change in surface parameters between adjacent shading points
`varying vector`	**L**	Inside `illuminance`: incoming light direction Inside lights (WRITABLE): outgoing light direction
`varying color`	**Cl**	Inside `illuminance`: incoming light color Inside lights (WRITABLE): light arriving at `Ps`.
`varying vector`	**I**	Surf/disp: Incident ray direction Vol: ray through the volume
`varying point`	**E**	Position of the eye
`varying vector`	**dPdtime**	How the surface position P is changing per unit time, as described by motion blur in the scene.
`varying color`	**Ci**, **Oi**	Incident ray color and opacity (WRITABLE)
`varying point`	**Ps**	Lights: Position of the surface being illuminated
`uniform float`	**ncomps**	Number of color components
`uniform float`	**time**	Current shutter time
`uniform float`	**dtime**	The amount of time covered by this shading sample.

Table 5.8: Shader "Global" Variables summary table.

5.6 Built-in Library Functions

Entropy's shading language provides a variety of built-in functions. For brevity, functions that are identical to those found in the standard C library are presented with minimal elaboration. Note that some functions are *polymorphic*, that is, they can take arguments of several

different types. In some cases we use the shorthand *ptype* to indicate a type that could be any of the point-like types `point`, `vector`, or `normal`.

5.6.1 Mathematical Functions

`float` **radians** `(float d)`
`float` **degrees** `(float r)`

Convert degrees to radians, and radians to degrees, respectively.

`float` **sin** `(float angle)`
`float` **cos** `(float angle)`
`float` **tan** `(float angle)`
`float` **asin** `(float f)`
`float` **acos** `(float f)`
`float` **atan** `(float y, x)`
`float` **atan** `(float y_over_x)`

Basic trigonometry. Angles, as in C, are assumed to be expressed in radians.

`float` **pow** `(float x, float y)`
`float` **exp** `(float x)`
`float` **log** `(float x)`
`float` **log** `(float x, b)`

Exponentials: return x^y, e^x, $\ln x$, and $\log_b x$, respectively.

`float` **sqrt** `(float x)`
`float` **inversesqrt** `(float x)`

Return $\sqrt{x}$ and $1/\sqrt{x}$, respectively.

`float` **abs** `(float x)`

Return $|x|$.

`float` **sign** `(float x)`

Returns 1 if $x > 0$, -1 if $x < 0$, 0 if $x = 0$.

`float` **floor** `(float x)`
`float` **ceil** `(float x)`
`float` **round** `(float x)`

Return the highest integer less than or equal to x, the lowest integer greater than or equal to x, or the closest integer to x, respectively.

`float` **mod** `(float a, b)`

Just like the *fmod* function in C, returns $a - b * \text{floor}(a/b)$.

type **min** (*type* a, b, ...)
type **max** (*type* a, b, ...)
type **clamp** (*type* x, minval, maxval)

The `min` and `max` functions return the minimum or maximum, respectively, of a list of two or more values. The `clamp` function returns

`min(max(x,minval),maxval)`,

that is, the value x clamped to the specified range. The *type* may be any of `float`, `point`, `vector`, `normal`, or `color`. The variants that operate on colors or point-like objects operate on a component-by-component basis (i.e., separately for x, y, and z).

float **mix** (float x, y; float alpha)
color **mix** (color x, y; float alpha)
point **mix** (point x, y; float alpha)
vector **mix** (vector x, y; float alpha)
normal **mix** (normal x, y; float alpha)

The `mix` function returns a linear blending of any simple *type* (any of `float`, `point`, `vector`, `normal`, or `color`): $x * (1 - \alpha) + y * (\alpha)$

5.6.2 Geometric and Color Functions

float **comp** (color c; float i)
float **comp** (ptype p; float i)

Return the i^{th} component of a color or point-like value.

void **setcomp** (output color c; float i, float x)
void **setcomp** (output ptype p; float i, x)

Modify color `c` (or point-like `p`) by setting its i^{th} component to value `x`.

float **xcomp** (ptype p)
float **ycomp** (ptype p)
float **zcomp** (ptype p)

Return the x, y, z, or simply the i^{th} component of a point-like variable.

void **setxcomp** (output ptype p; float x)
void **setycomp** (output ptype p; float x)
void **setzcomp** (output ptype p; float x)

Set the x, y, z, or simply the i^{th} component of a point-like type. These routines alter their first argument but do not return any value.

`color` **ctransform** `(string tospacename; color c_rgb)`
`color` **ctransform** `(string fromspacename, tospacename; color c_from)`

Transform a color from one color space to another. The first form assumes that `c_rgb` is already an "rgb" color and transforms it to another named color space. The second form transforms a color between two named color spaces.

`float` **length** `(vector V)`
`float` **length** `(normal V)`

Returns the length of a vector or normal.

`float` **distance** `(point P0, P1)`

Returns the distance between two points.

`float` **ptlined** `(point P0, P1, Q)`

Returns the distance from `Q` to the closest point on the line segment joining `P0` and `P1`.

`vector` **normalize** `(vector V)`
`vector` **normalize** `(normal V)`

Return a vector in the same direction as V but with length 1, that is, `V / length(V)`

`vector` **faceforward** `(vector N, I, Nref)`
`vector` **faceforward** `(vector N, I)`

If `Nref . I` < 0, returns `N`, otherwise returns `-N`. For the version with only two arguments, `Nref` is implicitly `Ng`, the true surface normal. The point of these routines is to return a version of `N` that faces towards the camera — in the direction "opposite" of `I`.

To further clarify the situation, here is the implementation of `faceforward` expressed in Shading Language:

```
vector faceforward (vector N, I, Nref)
{
   return (I.Nref > 0) ?  -N : N;
}

vector faceforward (vector N, I)
{
   extern normal Ng;
   return faceforward (N, I, Ng);
}
```

`vector` **reflect** `(vector I, N)`

For incident vector `I` and surface orientation `N`, returns the reflection direction `R = I - 2*(N.I)*N`. Note that `N` must be normalized (unit length) for this formula to work properly.

`vector` **refract** `(vector I, N; float eta)`

For incident vector `I` and surface orientation `N`, returns the refraction direction using Snell's law. The `eta` parameter is the ratio of the index of refraction of the volume containing `I` divided by the index of refraction of the volume being entered.

```
void fresnel (vector I; normal N; float eta;
              output float Kr, Kt; output vector R, T);
```

According to Snell's law and the Fresnel equations, `fresnel()` computes the reflection and transmission direction vectors `R` and `T`, respectively, as well as the scaling factors for reflected and transmitted light, `Kr` and `Kt`. The `I` parameter is the normalized incident ray, `N` is the normalized surface normal, and `eta` is the ratio of refractive index of the medium containing `I` to that on the opposite side of the surface.

```
point transform (string tospacename; point p_current)
vector vtransform (string tospacename; vector v_current)
normal ntransform (string tospacename; normal n_current)
```

Transform a point, vector, or normal (assumed to be in `"current"` space) into the `tospacename` coordinate system.

```
point transform (string fromspacename, tospacename; point pfrom)
vector vtransform (string fromspacename, tospacename; vector vfrom)
normal ntransform (string fromspacename, tospacename; normal nfrom)
```

Transform a point, vector, or normal (assumed to be represented by its `"fromspace"` coordinates) into the `tospacename` coordinate system.

```
point transform (matrix tospace; point p_current)
vector vtransform (matrix tospace; vector v_current)
normal ntransform (matrix tospace; normal n_current)

point transform (string fromspacename; matrix tospace; point pfrom)
vector vtransform (string fromspacename; matrix tospace; vector vfrom)
normal ntransform (string fromspacename; matrix tospace; normal nfrom)
```

These routines work just like the ones that use the space names but instead use transformation matrices to specify the spaces to transform into.

`point` **rotate** `(point Q; float angle; point P0, P1)`

Returns the point computed by rotating point `Q` by `angle` radians about the axis that passes from point `P0` to `P1`.

`float` **depth** `(point p)`

Return the depth of point `p`, normalized so that points at the near clip plane return a depth of 0, and points at the far clip plane return a depth of 1.

5.6.3 Matrix Functions

`float` **comp** `(matrix M; float r, c)`

Return a component of the matrix: $M_{r,c}$

`void` **setcomp** `(output matrix M; float r, c, x)`

Modify the matrix `m` by setting one of its components: $M_{r,c} = x$.

`float` **determinant** `(matrix M)`

Returns the determinant of matrix `M`.

`matrix` **translate** `(matrix M; point t)`
`matrix` **rotate** `(matrix M; float angle; vector axis)`
`matrix` **scale** `(matrix M; point t)`

Return a matrix that is the result of appending simple transformations onto the matrix `M`. These functions are similar to the RIB `Translate`, `Rotate`, and `Scale` commands, except that the rotation angle in `rotate()` is in radians, not in degrees as with the RIB `Rotate`. There are no perspective or skewing functions.

5.6.4 Pattern Generation Functions

`float` **step** `(float edge, x)`

Returns 0 if $x < edge$ and 1 if $x \geq edge$.

`float` **smoothstep** `(float edge0, edge1, x)`

Returns 0 if $x \leq edge0$, and 1 if $x \geq edge1$, and performs a smooth Hermite interpolation between 0 and 1 when $edge0 < x < edge1$. This is useful in cases where you would want a thresholding function with a smooth transition.

type **noise** `(float u)`
type **noise** `(float u, v)`
type **noise** `(point p)`
type **noise** `(point p; float t)`

The `noise()` function returns a continuous, pseudo-random (but repeatable) scalar field defined on a 1- (`float`), 2- (2 `float`'s), 3- (`point`), or 4-dimensional (`point` and `float`) domain. The function always lies between 0 and 1, with a large-scale average value of 0.5, is fairly isotropic, has no easily detectable periodicity, and has most of its energy at frequencies between 0.5–1. This makes it ideal to use as a basis function for producing complex natural-looking patterns.

The return *type* may be any of `float`, `color`, `point`, `vector`, or `normal`, depending on the type of the variable the result is assigned to (or based on an explicit type cast). For multi-component return types (e.g., `point`), each component is an uncorrelated `float` noise function.

type **pnoise** (float u; uniform float uperiod)
type **pnoise** (float u, v; uniform float uperiod, vperiod)
type **pnoise** (point p; uniform point pperiod)
type **pnoise** (point p; float t; uniform point pperiod; uniform point tperiod)

The pnoise() function is just like noise(), but repeats with the specified periodicity (which must be uniform and will be rounded down to the nearest integer). Other than the user-set periodicity, the appearance and statistical properties of pnoise are identical to those of noise().

type **cellnoise** (float u)
type **cellnoise** (float u, v)
type **cellnoise** (point p)
type **cellnoise** (point p; float t)

The cellnoise() function returns a discrete pseudo-random (but repeatable) scalar field defined on a 1- (float), 2- (2 float's), 3- (point), or 4-dimensional (point and float) domain. The function is uniformly distributed on $(0, 1)$, and has no easily detectable periodicity. The function is continuous between integer values and discontinuous just before integer values — in other words, $cellnoise(x) = cellnoise(\mathrm{floor}(x))$.

The return *type* may be any of float, color, point, vector, or normal, depending on the type of the variable the result is assigned to (or based on an explicit type cast). For multi-component return types (e.g., point), each component is an uncorrelated float cellnoise function.

NEW!

type **random** ( )
type **random** (float x)
type **random** (point p)

Returns a random value uniformly distributed between 0 and 1. The random() function may be either uniform or varying (depending on what it's assigned to) but randomgrid() always returns a uniform value (even if assigned to a varying variable).

The return *type* may be any of float, color, point, vector, or normal, depending on the type of the variable the result is assigned to (or based on an explicit type cast). For multi-component return types (e.g., point), each component is a separate uniformly distributed random number.

If random() is called with no arguments, a truly "random" result is returned, and it is not repeatable. If either a float or point (or, technically, a vector or normal) is passed as an argument, the random() function returns a seemingly random number that is really is a *hash* of its argument (and is therefore repeatable if passed the exact same argument again).

uniform *type* **randomgrid** ( )

Returns a `uniform` random value uniformly distributed between 0 and 1. The `randomgrid()` function is like `random()`, but always returns a `uniform` value (even if assigned to a `varying` variable).

The return *type* may be any of `float`, `color`, `point`, `vector`, or `normal`, depending on the type of the variable the result is assigned to (or based on an explicit type cast). For multi-component return types (e.g., `point`), each component is a separate uniformly distributed random number.

type **spline** (float x; *type* y_0, y_1, ... y_{n-1})
type **spline** (string basis; float x; *type* y_0, y_1, ... y_{n-1})
type **spline** (float x; *type* y[])
type **spline** (string basis; float x; *type* y[])

Fit a spline to uniformly spaced data values $y_0 \ldots y_{n-1}$, returning the interpolated value at x (which will be clamped to lie on $[0, 1]$). Instead of n data values, you may pass an array. The `spline` function will determine the length of the array.

The return *type* may be any of `float`, `color`, `point`, `vector`, or `normal`, depending on the type of the data values `y`. For multi-component splines (e.g., `color` splines), each component is interpolated separately.

Optionally, a spline basis may be specified by name. Valid basis names are: `"catmull-rom"`, `"bezier"`, `"bspline"`, `"hermite"`, or `"linear"`. If no basis name is supplied, `"catmull-rom"` is used. The number of data values must be $4n + 3$ for Bezier splines, and $4n + 2$ Hermite splines. Any number of knots may be used for Catmull-Rom or Linear splines, but the number of knots must be ≥ 4. To maintain consistency with the other spline types, linear splines will ignore the first and last data value, interpolating piecewise-linearly between y_0 and y_{n-2}.

The following basis names are also supported: `"solvecatmull-rom"`, `"solvebezier"`, `"solvebspline"`, `"solvehermite"`, `"solvelinear"`. For any of the `"solve"` spline types, whose data values may only be `float`, the spline's *inverse* is computed. That is, `spline()` returns the lookup value for which the spline's result would be *x*. Results are undefined and likely unstable if the knots do not determine a monotonically increasing spline.

5.6.5 Derivatives, Area Operators, and Antialiased Functions

float **Du** (float x), **Dv** (float x), **Deriv** (float x y)
vector **Du** (point x), **Dv** (point x), **Deriv** (point x; float y)
vector **Du** (vector x), **Dv** (vector x), **Deriv** (vector x; float y)

Compute derivatives of the argument. `Du` and `Dv` compute $\partial x/\partial u$ and $\partial x/\partial v$, respectively, where u and v are the 2D parametric surface coordinates.

`Deriv()` attempts to calculate $\partial x/\partial y$, but it is not particularly robust. We do not recommend relying on `Deriv`, and we consider it deprecated.

`float` **area** `(point p)`

Computes `length(Du(p)*du ^ Dv(p)*dv)`.

`normal` **calculatenormal** `(point p)`

Computes `Du(p) ^ Dv(p)`. In other words, it computes the surface normal of the surface that is defined by point `p` (as `p` is computed at all surface points).

`float` **filterstep** `(float edge, s; ...)`
`float` **filterstep** `(float edge, s0, s1; ...)`

The `filterstep` function provides an analytically antialiased step function. In its two-argument form, it takes parameters identical to `step`, but returns a result that is filtered over the area of the surface element being shaded. In its three-argument form, the step function is filtered in the range between the two values `s0` and `s1` (i.e., `s1-s0 = w`). This low-pass filtering is similar to that done for texture maps.

In both forms, an optional parameter list provides control over the filter function, and may include the following parameters: `"width"` (also known as `"swidth"`), the amount to "overfilter" in `s`; `"filter"`, the name of the filter kernel to apply. The filter may be any of the following: `"catmull-rom"` (the default), `"box"`, `"triangle"`, or `"gaussian"`.

5.6.6 String Functions

`void` **printf** `(string template, ...)`

Much as in C, `printf` takes a template string and an argument list. Where the format string contains the characters `%f`, `%c`, `%p`, `%m`, and `%s`, `printf` will substitute arguments, in order, from the argument list (assuming that the arguments' types are float, color, point-like, matrix, and string, respectively). In addition, `%d` and `%i` will also print `float`'s, truncating and printing them as if they were integers.

`string` **format** `(string template, ...)`

The `format` function, like `printf`, takes a template and an argument list. But `format` returns the assembled, formatted string rather than printing it.

NEW!

`void` **error** `(string template, ...)`

The `error` function is just like `printf`, but is printed as a renderer error message, including information about the name of the shader and the object being shaded.

`string` **concat** `(string s1, ..., sN)`

Concatenates a list of strings, returning the aggregate string.

`float` **match** `(string pattern, subject)`

Does a string pattern match on `subject`. Returns 1 if the `pattern` exists anywhere within `subject` and 0 if the `pattern` does not exist within `subject`. The `pattern` can be a standard Unix expression. Note that the pattern does not need to start in the first character of the `subject` string, unless the pattern begins with the `^` (beginning of string) character.

5.6.7 Texture, Reflection, and Shadow Access Functions

```
float texture (string filename; float s, t; ...params...)
float texture (string filename; float s0, t0, s1, t1, s2, t2, s3, t3;
               ...params...)
color texture (string filename; float s, t; ...params...)
color texture (string filename; float s0, t0, s1, t1, s2, t2, s3, t3;
               ...params...)
```

Perform an antialiased lookup into an image file, indexed by 2D coordinates. The version with a single 2D (s, t) coordinate automatically examines how s and t vary over the surface in order to antialias the lookup. Alternately, four corner coordinates (i.e., a total of 8 floats) may be used to specify a quadrilateral bound of the area to be filtered.

The `texture()` function may perform either a single-channel lookup (returning a `float`) or a three-channel lookup (returning a `color`), depending on what kind of variable the results are assigned to (or whether an explicit type cast is made).

The 2D lookup coordinate(s) may be followed by optional token/value arguments. Meanings of the arguments are described in Table 5.9.

Parameter	Type	Description
`"blur"`	`float`	Specifies additional blur when looking up the texture value. The default value is 0.
`"sblur"`	`float`	Separately specifies `"blur"` in the *s* direction.
`"tblur"`	`float`	Separately specifies `"blur"` in the *t* direction.
`"width"`	`uniform float`	Scales the size of the filter specified by coordinates, or the filter area that the renderer estimates from examining the derivatives of the coordinate. The default value is 1.
`"swidth"`	`uniform float`	Separately specifies `"width"` in the *s* direction.
`"twidth"`	`uniform float`	Separately specifies `"width"` in the *t* direction.
`"fill"`	`uniform float`	Specifies the value to return for any channels that are not present in the texture file.
`"firstchannel"`	`uniform float`	Specifies the first channel to look up from the texture map.
`"alpha"`	`varying float`	The alpha channel (i.e., the next channel after the channels returned by the function call) will be stored in the variable specified. This allows for RGBA lookups in a single call to `texture()`.

Table 5.9: Optional parameters for `texture()`.

```
float environment (string filename; vector R; ...params...  )
float environment (string filename; vector R0, R1, R2, R3; ...params...)
color environment (string filename; vector R; ...params...  )
color environment (string filename; vector R0, R1, R2, R3; ...params...)
```

Perform an antialiased lookup into an environment map, indexed by a direction vector. The version with a single direction automatically examines how the direction varies over the surface in order to antialias the lookup. Alternately, four direction vectors may be used to specify a solid angle to be filtered.

The `environment()` function may perform either a single-channel lookup (returning a `float`) or a three-channel lookup (returning a `color`), depending on what kind of variable the results are assigned to (or whether an explicit type cast is made).

NEW!

If the *filename* parameter is the string `"reflection"`, Entropy will perform ray tracing instead of an environment file lookup. Thus, a shader may perform either environment mapping or ray tracing with a single `environment()` call, and the name of the environment map (which can be passed as an argument to the shader) determines which file to use, or alternately that ray tracing should be performed. The `environment()` function will only "see" objects that are tagged as visible in reflections (with `Attribute "visibility" "reflection" [1]`, see Section 3.2.2). Note that when ray tracing, the optional `"blur"` and `"width"` parameters are only honored if `"samples"` is greater than 1.

The directional coordinate(s) may be followed by optional token/value arguments. Meanings of the arguments are described in Tables 5.10 and 5.11. The latter table describes parameters that only have meaning when ray tracing.

Parameter	Type	Description
`"blur"`	`float`	Specifies additional angular blur when looking up the environment value. The default value is 0.
`"sblur"`	`float`	Separately specifies `"blur"` in the *s* direction.
`"tblur"`	`float`	Separately specifies `"blur"` in the *t* direction.
`"width"`	`uniform float`	Scales the size of the lookup filter. The default value is 1.
`"swidth"`	`uniform float`	Separately specifies `"width"` in the *s* direction.
`"twidth"`	`uniform float`	Separately specifies `"width"` in the *t* direction.
`"fill"`	`uniform float`	Specifies the value to return for any channels that are not present in the map file.
`"firstchannel"`	`uniform float`	Specifies the first channel to look up from the map.
`"alpha"`	`varying float`	The alpha channel will be stored in the variable specified (this allows for RGBA lookups in a single call to `environment()`). For ray tracing, this returns the analogous quantity — an alpha value of 0 indicates that the sample rays hit no objects in the scene.

Table 5.10: Optional parameters for `environment()`.

Parameter	Type	Description
`"samples"`	`uniform float`	How many sample rays to average to give the final result.
`"bias"`	`float`	Ignores ray hits closer than this value in order to prevent incorrect self-reflection of surfaces. If no `"bias"` is supplied, or if its value is less than 0, the global shadow bias will be used.
`"maxhitdist"`	`float`	Ignores ray hits farther than this value (measured in `"current"` space units). If no `"maxhitdist"` is supplied, there is no maximum hit distance.
`"hitdist"`	`varying float`	The average distance to hits for all sample rays will be stored in the variable specified. Only sample rays that hit scene objects will be included in the average. For environment maps, the value will be 1e30.
`"Phit"`	`varying point`	The average position of hits for all sample rays will be stored in the variable specified. Only sample rays that hit scene objects will be included in the average. For environment maps, the variable is unchanged.
`"Nhit"`	`varying vector`	The average surface normal of hits for all sample rays will be stored in the variable specified. Only sample rays that hit scene objects will be included in the average. For environment maps, the variable is unchanged.

Table 5.11: Optional ray tracing parameters for `environment()`. These parameters are ignored when `environment()` is using an environment map instead of ray tracing.

`float` **shadow** `(string shadowname; point P; ...`*params*`... )`
`float` **shadow** `(string shadowname; point P0, P1, P2, P3; ...`*params*`...)`
`color` **shadow** `(string shadowname; point P; ...`*params*`... )`
`color` **shadow** `(string shadowname; point P0, P1, P2, P3; ...`*params*`...)`

Perform an antialiased lookup into a shadow depth map file, returning the amount of occlusion at a point. The version with a single point automatically examines how the position varies over the surface in order to antialias the lookup. Alternately, four points may be used to specify an area over which the shadow information is filtered.

NEW!

If the *shadowname* parameter is the string `"shadow"`, Entropy will perform ray tracing instead of a shadow map file lookup. Thus, a shader may perform either shadow depth mapping or ray tracing with a single `shadow()` call, and the name of the shadow map (which can be passed as an argument to the shader) determines which file to use, or alternately that ray tracing should be performed. The `shadow()` function will only "see" objects that are tagged as visible in shadows (with `Attribute "visibility" "shadow" [1]`, see Section 3.2.2).

The `shadow()` function may return either a `float` or a `color`, depending on what kind of variable the results are assigned to (or whether an explicit type cast is made). When performing a shadow map lookup (which is inherently single-channel), the `color shadow()` function replicates the shadow value in all three channels. When performing a ray traced shadow probe (which is inherently three-channel), the `float shadow()` function uses just the red channel. We therefore recommend always using the `color shadow()` function, which performs as expected for both ray tracing as well as shadow map lookups.

The positional coordinate(s) may be followed by optional token/value arguments. Meanings of the arguments are described in Table 5.12.

Parameter	Type	Description
`"blur"`	`varying float`	Additional blur when looking up the shadow value. The default value is 0 (sharp shadows). For maps, the amount is expressed as a fraction of the shadow map view angle; for ray-traing, it is scaled so that a blur of 1.0 indicates a sampling angle of 90 degrees.
`"sblur"`	`float`	Separately specifies `"blur"` in the *s* direction.
`"tblur"`	`float`	Separately specifies `"blur"` in the *t* direction.
`"width"`	`uniform float`	Scales the size of the lookup filter. The default value is 1.
`"swidth"`	`uniform float`	Separately specifies `"width"` in the *s* direction.
`"twidth"`	`uniform float`	Separately specifies `"width"` in the *t* direction.
`"samples"`	`uniform float`	How many samples to compute and average to give the final result. For shadow map lookups (but not for ray traced shadows), a minimum of 16 samples are taken.
`"bias"`	`float`	The amount to add to lookups in order to prevent incorrect self-shadowing of surfaces. For values less than 0, the global bias (`Option "shadow" "bias"`) is used instead.

Table 5.12: Optional parameters for `shadow()` calls.

5.6.8 Lighting Functions

`color` **ambient** `()`

Returns the contribution of so-called *ambient* light, which comes from no specific location but rather represents the low level of scattered light in a scene after bouncing from object to object.[2]

`color` **diffuse** `(vector N)`

Calculates light widely and uniformly scattered as it bounces from a light source off of the surface. Diffuse reflectivity is approximated by Lambert's law:

$$\sum_{i=1}^{\text{nlights}} Cl_i \max(0, N \cdot L_i)$$

where for each of the i light sources, L_i is the unit vector pointing toward the light, Cl_i is the light color, and N is the unit normal of the surface. The $\max$ function ensures that lights with $N \cdot L_i < 0$ (i.e., those *behind* the surface) do not contribute to the calculation.

The `N` vector is assumed to be of unit length. For lights that have a parameter called `__nondiffuse`, the `diffuse()` function scales the contribution of that light by `(1-__nondiffuse)`.

`color` **specular** `(vector N, V; float roughness)`
`color` **specular** `(string funcname; vector N, V; float roughness)`

NEW!

Computes so-called *specular* lighting, which refers to the way that glossier surfaces have noticeable bright spots or highlights resulting from the narrower (in angle) scattering of light off the surface. The `N` and `V` vectors are assumed to be of unit length. For lights that have a parameter called `__nonspecular`, the `specular()` function scales the contribution of that light by `(1-__nonspecular)`.

If the optional *funcname* is supplied, it selects which of several specular highlight shape functions to use. Currently supported functions include: `"entropy"` (a modified Cook-Torrance-Sparrow) or `"bmrt"` (BMRT's specular function, a modified Blinn-Phong). The function name `"default"`, as well as not specifying a function name at all, will result in using the default specular function set by `Option "render" "specularbrdf"` (see Section 3.1.3).

`color` **specularbrdf** `(vector L, N, V; float roughness)`
`color` **specularbrdf** `(string funcname; vector L, N, V; float roughness)`

NEW!

[2] In most renderers, ambient light is typically approximated by a low-level, constant, non-directional light contribution set by the user in a rather ad hoc manner. When renderers try to accurately calculate this interreflected light in a principled manner, it is known as *global illumination*, or, depending on the exact method used, as *radiosity*, *path tracing*, *Monte Carlo integration*, and others.

Computes the attenuation of light coming from `L` and bouncing off a surface with normal `N` and given `roughness`, as viewed from direction `V`. All of `L`, `N`, and `V` are assumed to be of unit length.

If the optional *funcname* is supplied, it selects which of several specular highlight shape functions to use (see the explanation for `specular()`).

`color` **phong** `(vector N, V; float size)`

Compute specular lighting using the Phong illumination model. The `N` and `V` vectors are assumed to be of unit length. For lights that have a parameter called `__nonspecular`, the `phong()` function scales the contribution of that light by `(1-__nonspecular)`.

`color` **trace** `(vector R; ...)`
`color` **trace** `(point Pos; vector R; ...)`

Use ray tracing to determine the color of light arriving at point `Pos` from the direction `R`. If no `Pos` is supplied, the ray tracing is done from the surface point `P`. The `trace()` function will only "see" objects that are tagged as visible in reflections (with `Attribute "visibility" "reflection" [1]`, see Section 3.2.2).

We strongly encourage the use of the `environment("reflection", ...)` function instead of `trace()`.

`color` **visibility** `(point P0, P1)`

Use ray tracing to compute the visibility (`color 1` if unoccluded, `color 0` if fully occluded) between points `P0` and `P1`. The `visibility()` function will only "see" objects that are tagged as visible in shadows (with `Attribute "visibility" "shadow" [1]`, see Section 3.2.2).

We strongly encourage the use of the `shadow("shadow", ...)` function instead of `visibility()`.

`float` **rayhittest** `(point Pos; vector R; output point Phit; output vector Nhit)`

Use ray tracing to detect the distance to the nearest object as seen from point *Pos* looking in the direction of the vector *R* (which does not need to be normalized). If no object is hit in that direction, the function will return `1e38` and the output parameters *Phit* and *Nhit* will not be altered. If an object is hit, *Phit* will be set to the position of the intersection, *Nhit* will be the surface normal of the object that was hit at the point of intersection, and the return value of `rayhittest()` will be the distance to the object.

The `rayhittest()` function will only "see" objects that are tagged as visible in reflections (with `Attribute "visibility" "reflection" [1]`, see Section 3.2.2).

NEW!

`float` **occlusion** `(point Pos; vector R; float angle;`
`[output vector Nunocc;] ...)`

Use ray tracing to probe the cone with apex *Pos*, in direction *R* and with half-angle *angle*. The return value is the portion of the solid angle that is occluded (1 if all sample rays hit objects, 0 if all sample rays miss all objects). If the optional *Nunocc* is supplied, it will have its value replaced by the average direction of the probe rays that were not occluded. The `occlusion()` function takes optional arguments, much like `trace()`, including `"samples"`, `"bias"`, and `"maxhitdist"`.

The occlusion information is computed sparsely and interpolated, much like indirect illumination, and honors the several indirect illumination attributes described in Section 3.2.2. The `"samples"` parameter, if supplied, overrides the `Attribute "indirect" "nsamples"`.

The `occlusion()` function will only "see" objects that are tagged as visible in reflections (with `Attribute "visibility" "reflection" [1]`, see Section 3.2.2).

Currently, the true value of *angle* is ignored, and a value of `PI/2` is assumed. This means that `occlusion()` samples the hemisphere centered around *R*.

5.6.9 Message Passing

`float` **displacement** `(string name; output` *`type`* `destination)`

Examine the parameters of the displacement shader, looking for a parameter with the given *name*. If the parameter exists and has the same type and storage class as *destination*, the value of the parameter will be stored in *destination* and the `displacement()` function will return `1.0`. If the named parameter is not found in the displacement shader, or if its type and storage class do not match that of *destination*, then *destination* will not be modified and `displacement()` will return `0`.

The `displacement()` function may be called from any other shader, but only when called from a surface shader will it return the correct values of any `output` parameters that were actually set by the displacement shader. Calling `displacement()` from light, atmosphere, or other shaders will only correctly return the ordinary parameter values, not `output` parameters.

`float` **surface** `(string name; output` *`type`* `destination)`

Examine the parameters of the surface shader, looking for a parameter with the given *name*. If the parameter exists and has the same type and storage class as *destination*, the value of the parameter will be stored in *destination* and the `surface()` function will return `1.0`. If the named parameter is not found in the surface shader, or if its type and storage class do not match that of *destination*, then *destination* will not be modified and `surface()` will return `0`.

The `surface()` function may be called from any other shader, and can even be used to query values of `output` parameters that are set by execution of the surface shader,

except when called from the displacement shader (since the surface shader has not yet run).

`float` **atmosphere** `(string name; output` *type* `destination)`

Examine the parameters of the atmosphere shader, looking for a parameter with the given *name*. If the parameter exists and has the same type and storage class as *destination*, the value of the parameter will be stored in *destination* and the `atmosphere()` function will return `1.0`. If the named parameter is not found in the atmosphere shader, or if its type and storage class do not match that of *destination*, then *destination* will not be modified and `atmosphere()` will return `0`.

The `atmosphere()` function may be called from any other shader, but since atmosphere shaders are run last, there's no way that other shaders can find out the values of any `output` parameters that were set by execution of the atmosphere shader.

`float` **lightsource** `(string name; output` *type* `destination)`

When called from inside an `illuminance` loop in a surface or volume shader, this function examines the shader of the current light source being examined, looking for a parameter with the given *name*. If the parameter exists and has the same type and storage class as *destination*, the value of the parameter will be stored in *destination* and the `lightsource()` function will return `1.0`. If the named parameter is not found in the light shader, or if its type and storage class do not match that of *destination*, then *destination* will not be modified and `lightsource()` will return `0`.

The `lightsource()` function may only be called within `illuminance` statements, and may therefore only be called from shaders that can have `illuminance` statements (surface and volume shaders).

`float` **incident** `(string name; output` *type* `destination)`
`float` **opposite** `(string name; output` *type* `destination)`

Examine the parameters of the interior or exterior shader, looking for a parameter with the given *name*. If the parameter exists and has the same type and storage class as *destination*, the value of the parameter will be stored in *destination* and the function will return `1.0`. If the named parameter is not found in the surface shader, or if its type and storage class do not match that of *destination*, then *destination* will not be modified and the function will return `0`.

The `incident()` function queries whichever of the interior or exterior shader is on the same side of the surface as `I`. The `opposite()` function queries whichever of the interior or exterior shader is on the opposite side of the surface as `I`.

The `incident()` and `opposite()` functions may be called from any other shaders, but may only be used to query values of ordinary parameters, not `output` parameters.

5.6.10 Renderer State Queries

`float` **option** `(string name; output` *type* `destination)`

Examine the renderer options, looking for a parameter with the given *name*. If the option exists and has the same type and storage class as *destination*, the value of the option will be stored in *destination* and the `option()` function will return `1.0`. If the name is not recognized or if its type does not match that of *destination*, then *destination* will not be modified and `option()` will return `0`. Table 5.13 lists the option names that Entropy understands. All option values returned are `uniform`.

NEW!

User options (see Section 3.1.3) can be retrieved using the `option()` function using the *name* prefixed with `user:`. For example, if you set a user option with the scene file command

```
Option "user" "float temperature" [212]
```

you may retrieve it from the shader with:

```
option ("user:temperature", tempdata);
```

The data type must, of course, match the declaration at the time of the `Option` call.

Name	Type	Description
`"Format"`	`float` [3]	The x and y resolution and pixel aspect ratio, as passed to the `Format` statement.
`"DeviceResolution"`	`float` [3]	The x and y resolution and the pixel aspect ratio of the actual image, taking into consideration how `RiFrameAspectRatio` or `RiScreenWindow` can effectively override `Format`.
`"FrameAspectRatio"`	`float`	Frame aspect ratio.
`"CropWindow"`	`float` [4]	Boundaries of the crop window.
`"DepthOfField"`	`float` [3]	Fstop, focallength, focaldistance.
`"Shutter"`	`float` [2]	Shutter open and close time.
`"Clipping"`	`float` [2]	Near and far clip plane depths.
`"user:`*name*`"`	user-specified	Retrieves the user-set option *name*.

Table 5.13: Data names known to the `option` function. Option values are always `uniform`.

`float` **attribute** `(string name; output` *type* `destination)`

Examine the attributes of the geometric primitive being shaded, looking for a parameter with the given *name*. If the attribute exists and has the same type and storage class as *destination*, the value of the attribute will be stored in *destination* and the `attribute()` function will return `1.0`. If the name is not recognized or if its type does not match that of *destination*, then *destination* will not be modified and

`attribute()` will return `0`. Table 5.14 lists the attribute names that Entropy understands. All attribute values returned are `uniform`.

NEW!

User attributes (see Section 3.2.2) can be retrieved using the `attribute()` function using the *name* prefixed with `user:`. For example, if you set a user attribute with the scene file command

```
Attribute "user" "point refpoint" [3.14 2 1]
```

you may retrieve it from the shader with:

```
attribute ("user:refpoint", tempdata);
```

The data type must, of course, match the declaration at the time of the `Attribute` call.

Name	Type	Description
`"ShadingRate"`	`float`	Shading rate.
`"Sides"`	`float`	1 or 2, depending on whether the surface is single- or double-sided.
`"Matte"`	`float`	1 if the surface is a `Matte` object, 0 otherwise.
`"dice:motionfactor"`	`float`	The value of the `"motionfactor"` attribute.
`"displacementbound:sphere"`	`float`	The amount of displacement bound.
`"displacementbound:coordinatesystem"`	`string`	The coordinate system used for the displacement bound.
`"identifier:name"`	`string`	The name given to the geometry.
`"user:`*name*`"`	user-specified	Retrieves the user-set attribute *name*.

Table 5.14: Data names known to the `attribute` function. Attribute values are always `uniform`.

`float` **rendererinfo** `(string name; output` *`type`* `destination)`

Return information about the renderer brand and version. If the named field exists and has the same type and storage class as *destination*, its value will be stored in *destination* and the `rendererinfo()` function will return `1.0`. If the name is not recognized or if its type does not match that of *destination*, then *destination* will not be modified and `rendererinfo()` will return `0`. Table 5.15 lists the data names that Entropy understands. All values returned are `uniform`.

`float` **textureinfo** `(string texturename, paramname; output` *`type`* `destination)`

Return information about a particular stored texture file. If the file exists and the parameter is recognized and has the same type and storage class as *destination*, its value will be stored in *destination* and the `textureinfo()` function will return `1.0`. If the file does not exist, or the name is not recognized, or if its type does not match that

Name	Type	Description
"renderer"	string	The brand name of the renderer (e.g., "Entropy").
"version"	float [4]	Major, minor, release, and patch numbers (e.g., { 3, 1, 0, 0 }).
"versionstring"	string	The release numbers expressed as a string (e.g., "3.1.0.0").

Table 5.15: Data names known to the rendererinfo function. Data values are always uniform.

of *destination*, then *destination* will not be modified and textureinfo() will return 0. Table 5.16 lists the data names that Entropy understands. All values returned are uniform.

Name	Type	Description
"resolution"	float [2]	The resolution of the highest resolution version of the image stored in the texture map.
"type"	string	Returns one of: "texture", "shadow", or "environment".
"channels"	float	The number of channels in the texture map.
"viewingmatrix"	matrix	(Shadow maps only) The matrix that transforms points from "current" space to the "camera" space from which the texture was created.
"projectionmatrix"	matrix	(Shadow maps only) The matrix that transforms points from "current" space to a 2D coordinate system where x and y range from -1 to 1.

Table 5.16: Names known to the textureinfo function. All values returned are uniform.

uniform float **raylevel** ( )

Returns 0 if the shader is computing the appearance of a surface directly visible from the camera, 1 if it is calculating the appearance of a one-bounce reflection ray, 2 if a two-bounce reflection ray, etc.

uniform float **isshadowray** ( )

Returns 1 if the shader is being run to evaluate the opacity of an object for the purpose of a ray-traced shadow, otherwise returns 0.

uniform float **isindirectray** ( )

NEW!

Returns nonzero if the shader is being run to evaluate indirect illumination, otherwise returns 0.

`string` **shadername** `( )`
`string` **shadername** `( string shadertype )`

Returns the name of the shader being executed (if no string is passed), or the name of the shader type specified (such as `"surface"`, `"displacement"`, etc.).

Part II

Getting Things Done

Chapter 6

Invoking Entropy from the Command Line

The `entropy` program is the main high-quality renderer of the Entropy package. The format for invoking `entropy` is as follows:

```
entropy [options] scenefile
```

Usually, this will result in one or more image 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.

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

```
myprog | entropy
```

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

6.1 Command line arguments

The following optional command-line arguments are accepted by `entropy`:

`-help`

Print out the possible command line arguments.

`-arch`

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

`-beep`

Rings the terminal bell upon completion of rendering.

`-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 a `CropWindow` option in the scene file takes precedence over any command line arguments).

`-cwd` *path*

Before rendering, change the working directory to *path*.

`-d`

By default, any fully rendered frames are sent to an image file of the type determined by the `Display` command. The `-d` command line option overrides the `Display` command and forces use of the `"framebuffer"` display server.

`-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,

```
entropy -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.

`-res` *xres yres*

Sets the resolution of the output image. Note that if the scene contains a `Format` statement that 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.

`-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.

`-silent`

Suppresses most of the output ordinarily generated (such as percentage of the image completed).

`-progress`

Print an alternate, more brief percentage progress message.

`-Progress`

Print another alternate progress message that is in a format that Alfred understands.

`-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.

`-threads` *n*

NEW!

Uses *n* CPU's (on the same machine) to render the image. The default is $n = 1$.

`-v`

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

`-version`

Print out the version of Entropy.

6.2 Initialization File

Before rendering any scene files specified on the command line or piped to it, `entropy` (or `rgl`) will first read the contents of the file `$ENTROPYHOME/.entropyrc` (if the file exists), followed by the file `$HOME/.entropyrc`. By putting commands in one of these files, you can set various options for Entropy before any other scene input is read. Notice that commands in `$HOME/.entropyrc` can override those set in `$ENTROPYHOME/.entropyrc`.

6.3 Return Codes

NEW!

The `entropy` program itself returns a code of 0 if rendering was completed, 13 if the rendering aborted because no free licenses could be found, and 1 if rendering terminates for any other error condition. These return codes may be checked from scripts that launch `entropy`.

6.4 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
```

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). If the input consists of multiple frames, by default `rgl` will display all of the frames 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 that 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
```

6.4.1 *rgl* Command Line Options

In addition to the `entropy` command-line arguments described in Section 6.1, `rgl` also accepts several additional command-line arguments specific to its functions as a GL-based previewer:

`-1buffer`

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

`-unlit`

> Lights all geometry with a single light at the camera position. This is useful for using *rgl* to preview a 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.

`-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.

`-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).

6.4.2 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: `DepthOfField`, `Shutter`, `PixelSamples`, `PixelFilter`, `Exposure`, `Imager`, `Quantize`, `Hider`, `Atmosphere`, `Opacity`, `TextureCoordinates`, `ShadingRate`, `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.
- Texture map generation functions (e.g., `MakeTexture`) do nothing in *rgl*.

Chapter 7

Image Output

This chapter covers the basic details of how the CG camera is placed in the scene, and the various options that must be set to determine image resolution and framing, camera attributes, image quality, exactly what data is saved, and how you can determine the image file types and other properties.

7.1 The Camera

7.1.1 Camera Projection

Even with the camera as the center of the universe, the scene is still three-dimensional, but the final image is two-dimensional. The reduction from three to two dimensions is accomplished by *projection*. In any projection, all points along a "line of sight" correspond to the same 2D location in the final image. Entropy supports both *orthographic* (parallel lines of sight) and *perspective* (lines of sight converging at a point). Along any line of sight, the closest object to the camera will be the one seen (although if it is partially transparent, you may also see other objects behind it). This is illustrated by Figures 7.1 and 7.2.

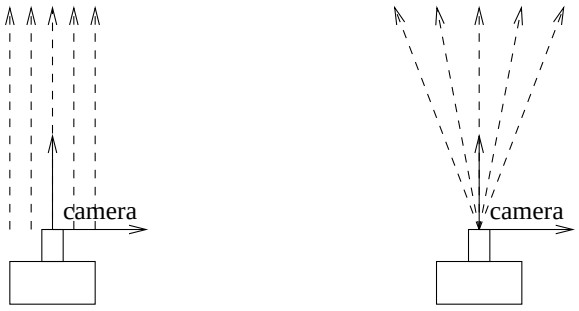


Figure 7.1: Left: Orthographic projections view along parallel rays. Right: Perspective projections view along rays that converge at the camera position.

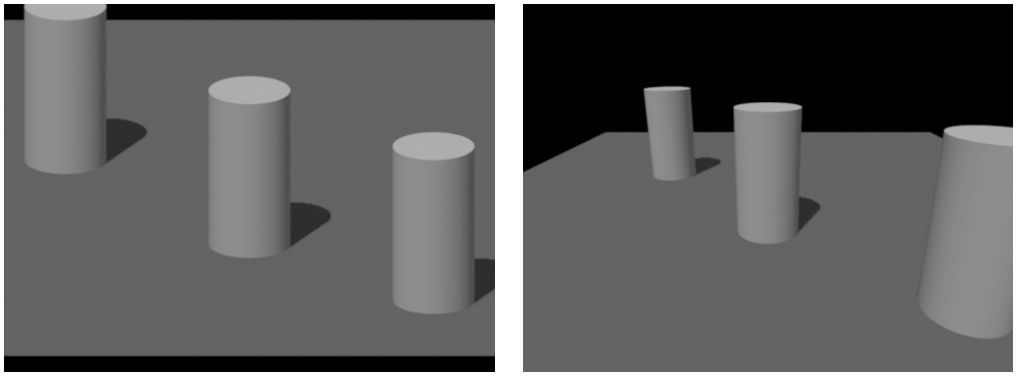


Figure 7.2: Example of an orthographic (left) and perspective (right) projections.

The projection is selected using the `Projection` command (see Section 3.1.1). Setting an orthographic projection is performed by:

```
Projection "orthographic"
```

Orthographic projections are primarily used for reproducing certain architectural or engineering drawing methods, and for creating shadow maps for "distant" light sources (those whose light emanates in parallel rays).

Perspective projections are more commonly used. Perspective projections are much more similar to the projection used in an ordinary camera.[1] You will almost always want to use a perspective projection for "final" images from the main camera. Perspective projections should also be used when generating shadow maps for light sources that project light from a single point (like a spotlight). Setting a perspective projection is performed by:

```
Projection "perspective"
```

The perspective projection also takes an optional `"fov"` parameter which sets the field-of-view angle in degrees. For example, the following command sets a perspective projection with a 30 degree field of view:

```
Projection "perspective" "fov" [30]
```

Each frame should have only one `Projection` command, and it should be prior to any other transformations. If no `Projection` command is encountered, an orthographic projection is used.

7.1.2 Positioning the Camera

Objects in the scene are positioned relative to `"world"` space or some other local coordinate system. This is the result of your having translated or rotated those objects to place them in the scene.

[1] Real cameras are not exactly perfect perspective projections. But those details are beyond the scope of this chapter. Perspective projections are good enough for most usage.

The camera also has a certain position and orientation relative to the world. A special coordinate system called "camera" space is centered about the camera, with the x axis pointing to the camera's right, the y axis pointing up, and the z axis pointing in the direction that the camera is looking.

In reality, the Entropy starts out with "camera" space as the transformation. Any transformation routines (such as ConcatTransform, Translate, or Rotate) that happen prior to WorldBegin are actually positioning the world relative to the camera. These transformations should be correctly specified by your modeling system. However, as an example of how the camera transformation can be computed, please refer to Listing 7.1.

Listing 7.1 Sample code to generate the camera transformation, given a camera position, center of interest, and "up" vector. This relies on a Point class that obeys the usual rules of vector math.

```
void
PlaceCamera (Point pos, Point interest, Point up, bool righthanded)
{
    float M[4][4];
    Point dir = normalize(interest - pos);
    up = normalize(up);
    up -= dot(up, dir) * dir;
    up = normalize(up);
    Point right = normalize(cross(up, dir));
    if (righthanded)
        right = -right;
    M[0][0] = right[0];  M[1][0] = right[1];  M[2][0] = right[2];  M[3][0] = 0;
    M[0][1] = up[0];     M[1][1] = up[1];     M[2][1] = up[2];     M[3][1] = 0;
    M[0][2] = dir[0];    M[1][2] = dir[1];    M[2][2] = dir[2];    M[3][2] = 0;
    M[0][3] = 0;         M[1][3] = 0;         M[2][3] = 0;         M[3][3] = 1;
    RiConcatTransform (M);
    RiTranslate (-pos[0], -pos[1], -pos[2]);
}
```

An early step in rendering is for objects to be transformed into "camera" space. This is illustrated figuratively in Figure 7.3.

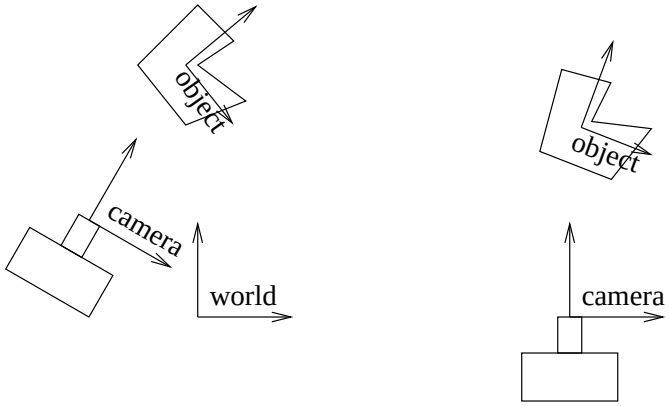


Figure 7.3: Transforming objects into camera space.

7.1.3 Motion Blur

Real cameras have a shutter that stays open for a certain amount of time to expose the film. The longer the shutter stays open, the more that moving objects will appear as blurred streaks on the film. This effect is actually critical to avoiding strobing when rendering frames for an animation. The shutter interval may be set with the `Shutter` command, which takes the opening and closing times. For example,

```
Shutter 0 0.5
```

instructs the camera to open the shutter a time 0 and close it again at time 0.5.

The units (seconds, frames, etc.) do not matter, but they are expected to be calibrated to the same scale as the times specified by `MotionBegin` for any moving or deforming objects.

For real cameras, the longer the shutter stays open, the more light strikes the film, and therefore the brighter the resulting image will be. This is not true for the synthetic camera — the image will be no brighter or dimmer, no matter what the `Shutter` statement specifies. This can be independently adjusted with `Exposure`.

Entropy 3.1 currently supports only two motion times for each motion block. This will be extended in a future release.

7.1.4 Film Exposure

The "film sensitivity" may be adjusted with the `Exposure` command. `Exposure` takes two parameters, *gain* and *gamma*. The gain is a scaling factor that linearly increases the film's sensitivity to light. Gamma refers to the nonlinearity of the film or, alternately, compensates for the nonlinearity of the intended display device.

In general, we do not recommend using a gamma value other than 1. Instead, we strongly recommend always rendering with a strictly linear response and performing gamma correction at display time, and for the particular display device being used.

7.1.5 Depth of Field

NEW!

Depth of field refers to the way objects at a particular distance from the camera appear in sharp focus, while objects that are closer or farther away will appear blurred (see Figure 7.4). It is a physical phenomenon caused by the finite aperture of a camera, and other focusing attributes of the lens system.

By default, Entropy has depth of field turned off, meaning that all objects are in sharp focus, regardless of their depth in the scene. The depth of field effect can be turned on with the `DepthOfField` command, which takes three arguments giving the f-stop, focal length, and focal distance.

The f-stop is the ratio of focal length to lens aperture, much as you would see f-stop settings on a real camera lens — it lets you control the aperture size. The focal length is the distance from the lens opening to the film plane. The focal distance is the depth from the camera at which objects appear in sharp focus. Both the focal length and focal distance are measured in the same units as `"camera"` space.

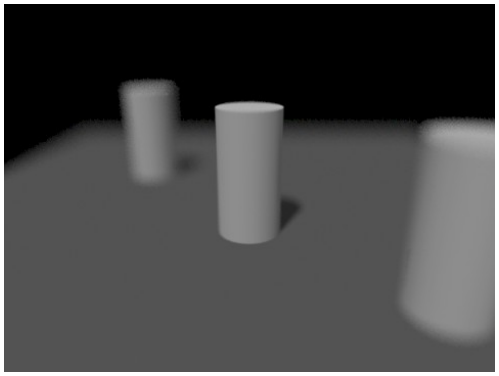


Figure 7.4: Depth of field.

For example, if you had constructed your scene so that `"camera"` space units were meters, then the following command would specify an f/4 aperture on a 50mm lens, set to focus sharply objects that were 3.6 meters from the camera:

```
DepthOfField 4 .05 3.6
```

For real cameras, the wider the aperture (i.e., the smaller the f-stop number), the more light enters the camera, and therefore the brighter the resulting image will be. This is not true for the synthetic camera — the image will be no brighter or dimmer, no matter what the `DepthOfField` statement specifies. This can be independently adjusted with `Exposure`.

7.1.6 Clipping

In addition to objects being not visible to the camera if it is too far to the right or left, top or bottom (that is, off-screen), you can also have the camera ignore objects that are too near to, or too far from the camera. This is something that obviously cannot be done with a real camera, but it can be very useful and often comes in handy with the CG camera. Objects are ignored if their `"camera"` space z values are less than the *hither* plane, or if their `"camera"` space z values are greater than the *yon* plane.

The z clipping planes can be set with the `Clipping` command. For example, to set the hither plane to $z = 0.1$ and the yon plane to $z = 10,000$:

```
Clipping 0.1 10000
```

There is some benefit to attempting to set the `Clipping` carefully. Tightly bounding the depth of interest in your scene can preserve more computational precision in some parts of the rendering process.

7.2 Image Resolution and Framing

7.2.1 Image Format

The image *format* refers to the number and shape of the pixels in the final image. The `Format` command (see Section 3.1.1) can be used to set the x resolution, y resolution, and

pixel aspect ratio. The x and y resolution (whole numbers) are the size of the final image, in pixels. The pixel aspect ratio describes the ratio of the width to height of an individual pixel (and therefore is 1.0 for square pixels). For example, to render an image with 640×480 square pixels:

```
Format 640 480 1.0
```

7.2.2 Frame Aspect Ratio and Screen Window

The aspect ratio of the frame is, by default, determined by the x and y resolution set by the `Format` command. However, you may override the frame aspect ratio using the `FrameAspectRatio` command (see Section 3.1.1). If `FrameAspectRatio` does not match the aspect ratio implied by `Format`, the image will be scaled to fit into the given resolution.

Once the scene is projected to a 2D plane, only a subset of the plane is actually turned into the image. That subset is called the *screen window*. By default, the screen window maps screen window boundaries

$$(-frameaspectratio, frameaspectratio, -1, 1)$$

if $frameaspectratio \geq 1$. If $frameaspectratio < 1$, the default `ScreenWindow` coordinates are:

$$(-1, 1, -1/frameaspectratio, 1/frameaspectratio)$$

Since the default is for the screen window to be centered and with the frame aspect ratio, it is usually not necessary to include the `ScreenWindow` command. However, there are two instances where it is critical: (1) For an orthographic camera, the `ScreenWindow` command is almost certainly required to ensure proper framing of the image. (2) `ScreenWindow` can be used to distort or shear the image by making the screen window's aspect ratio not match the frame aspect ratio, or by using an off-center screen window.

7.2.3 Crop Window

It is often very useful to render a subset of image pixels, particularly if you are debugging or adjusting part of the scene and do not wish to wait for the entire image to rerender every time a tweak is made. This is easily accomplished with the `CropWindow` command, which takes x minimum and maximum, and y minimum and maximum range, expressed as a portion of the total image (i.e., 0–1). For example, to render the upper-right quadrant of the image only:

```
CropWindow 0.5 1 0 0.5
```

Even though the crop window is expressed with floating-point numbers, it will be rounded in such a way as to result in a whole number of pixels. Entropy is careful to round and to filter at the edges so that adjacent crop windows will match up exactly without seams. For example, rendering one image with:

```
CropWindow 0 1 0 0.5
```

and a second image of the same scene with:

```
CropWindow 0 1 0.5 1
```

will *exactly* render all pixels, without repetition and without seams if the two images are assembled together.

7.3 Antialiasing and Filtering

Antialiasing refers to the renderer's efforts to correctly capture details smaller than a pixel (including geometric edges) and to give a properly smooth appearance to the blur that results from motion or depth of field.

There are two options that control the basic time versus quality tradeoffs when performing antialiasing. The first control is the *spatial quality*, which describes the number of subpixel regions (in x and y) comprising each pixel, for example:

```
Option "limits" "spatialquality" [2 2]
```

Dividing pixels into smaller regions, each of which is solved separately, is important to antialiasing because smaller regions are geometrically simpler (contain fewer objects and edges) and therefore easier to approximate with certain simplifying assumptions. More subpixel regions will yield higher quality, but will take somewhat longer to render. Setting `"spatialquality"` to `[1 1]` is good enough for previews, whereas `[2 2]` is high enough quality for most final images.[2] Higher values can be used for scenes that are unusually difficult to antialias (e.g., if there is lots of small geometry like hair, or that contain significant depth-of-field blur).

If your scene contains motion-blurred objects, a second time versus quality control is `"temporalquality"`, for example:

```
Option "limits" "temporalquality" [4]
```

The temporal quality specifies *additional* antialiasing for each spatial subregion. The total effective temporal quality is the `"spatialquality"` multiplied by the `"temporalquality"`. Specifying the spatial and temporal antialiasing controls separately helps the renderer to understand how to allocate its resources more optimally than if a single number were used. In particular, Entropy uses the extra temporal quality only for pixels that actually contain moving geometry. In other words, if only some objects in the scene are moving, less work is done in pixels where all objects are still, therefore rendering is much faster than if the work was done uniformly in all pixels.

As a shortcut, and in order to preserve backward compatibility with older renderers, Entropy still supports the `PixelSamples` directive. Since Entropy has no direct analog to `PixelSamples`, the `PixelSamples xsamp ysamp` command sets both spatial and temporal quality controls as follows:

[2] If you have experience with other renderers, you may think that 2×2 subsamples are not very high quality, but you're probably used to a render that does point sampling. Entropy's use of a semi-analytic antialiasing algorithm results in substantially higher quality (but also somewhat higher expense) for the same number of subpixel regions.

```
Option "limits" "spatialquality" [xsamp/2 ysamp/2]
Option "limits" "temporalquality" [4]
```

For example, `PixelSamples 2 2` is equivalent to setting 1×1 spatial subregions per pixel, with a temporal quality of 4. `PixelSamples 4 4` is equivalent to setting 2×2 spatial subregions per pixel, with a temporal quality of 4. This heuristic was chosen carefully so that a particular `PixelSamples` setting will produce approximately the same overall quality level with Entropy as with a point-sampling renderer such as BMRT or PRMan.

The several spatial subpixel regions must be combined to form the final discrete pixels. To do so with high quality, each pixel gets a weighted average of nearby regions (including regions outside the boundaries of the pixel. This process is known as *filtering*. Filtering has two aspects: the shape of the filter (specified by the name of the filtering function), and the width of the region to which the filter is applied. The default filtering can be set by the `PixelFilter` command. A 2×2 Gaussian filter (the default) may be specified by:

```
PixelFilter "gaussian" 2.0 2.0
```

Some people find the 2×2 Gaussian filter to be overly blurry. If that is the case, you could try a Gaussian filter with thinner width (but we wouldn't recommend using a width of less than 1.5), or you could use a different filter shape. The Catmull-Rom filter has nice edge sharpening properties, and can be specified as:

```
PixelFilter "catmull-rom" 3.0 3.0
```

On the other hand, if it is important that pixels equally weight all regions and not consider any spatial regions outside the pixel boundary, then you would want to specify the (infamously low quality) box filter:

```
PixelFilter "box" 1 1
```

Feel free to experiment with different filter functions and window sizes, to achieve a "look" that is right for your project. The available filters are listed in the formal description of the `PixelFilter` command in Section 3.1.2. Note also that the pixel filter can be specified on the `Display` line itself, without the need for a separate `PixelFilter` command.

7.4 Image Output

Once pixel values are derived by filtering the data from the pixel subregions, an image must be written to disk in some format, or displayed on some device.

Output streams and Channels

Upon rendering, Entropy produces one or more *image output streams*. You can think of each output stream as a separate image of the scene. Each output stream may contain different data — for example, one output stream may consist of color and alpha (RGBA), while another output stream may contain z depth information.

Each image output stream consists of one or more *channels*. A channel is a single "pane" of data, such as red or blue. A greyscale only image is a 1-channel image, an ordinary color RGB image is a 3-channel image, and an RGBA image is a 4-channel image.

Display servers

There are many different types of image file formats or devices on which to display images. Entropy uses programs called *display servers* (sometimes also called *display drivers*) to handle image output and display. There is one display server for each file format or display type. Entropy comes with display servers that understand how to write TIFF files (`"tiff"`), z-depth files (`"zfile"`), shadow maps (`"shadow"`), and how to display the image on a computer screen (`"iv"`). Developers can expand Entropy's format repertoire by writing their own display servers, as explained in Chapter 16.

The basic way to specify what data goes to which file and in what format is with the `Display` command:

`Display` *name format data ...params...*

The *name* is the filename of the file, the *format* is the file format or display device (actually the name of the display server), and *data* is the name of the data to write to the output stream. The optional *params* control various aspects of the image output, including driver-specific options.

As an example, to instruct Entropy to write RGB color data to a TIFF file named `"myfile.tif"`:

```
Display "myfile.tif" "tiff" "rgb"
```

To write an image with RGB and alpha (coverage) as a 4-channel image:

```
Display "myfile.tif" "tiff" "rgba"
```

If the *format* is the special keyword `"file"`, the default file display server will be used (which is `"tiff"`). If *format* is `"framebuffer"`, the default frame buffer display server will be used (which is `"iv"`). Therefore, to display the image "live" to a framebuffer display:

```
Display "myfile.tif" "framebuffer" "rgba"
```

Bit depth, quantization, and dither

Entropy computes pixel values with floating-point precision, but not all output formats support floating-point data. Therefore, the display server may need to convert the raw pixel data to an integer (whole number) representation. This process is known as *quantization*.

The `Display` command takes the optional parameter `"quantize"` that specify the quantization mapping. The `"quantize"` parameter takes an array of four floating-point numbers that specify the *zero* level, *one* level, *min*, and *max* values. When converted to integers, the number of bits per channel is known as the *bit depth*. The bit depth is computed automatically from the *max* quantization value: if $max \leq 255$, an 8-bit file is created; otherwise, if $max \leq 65535$, a 16-bit file is created; otherwise, a floating-point output file is created. Also, if all of *zero*, *one*, *min*, and *max* are 0, floating-point output will be selected.

For example, to write `"myfile.tif"` as 8-bit integers (this is a typical output format, and also the default):

```
Display "myfile.tif" "tiff" "rgba" "quantize" [0 255 0 255]
```

If 8 bits per channel are not enough precision for your application, you could generate a 16 bit per channel image with the following command:

```
Display "myfile.tif" "tiff" "rgba" "quantize" [0 65535 0 65535]
```

To aid in reducing artifacts that result from the float-to-integer conversion, you can add a random *dither* to the image. This is just noise that helps to soften the edges and reduce objectionable banding in the image. The dither amplitude can be set by `Display` using the optional parameter `"dither"`. The default dither level is 0.5. The main reason to override this default is in the case of floating-point images, which do not need dither and therefore should have their dither set to 0.

For example, to output color pixels with full floating-point precision (and no dither):

```
Display "myfile.tif" "tiff" "rgba" "quantize" [0 0 0 0] "dither" [0]
```

Filters

As described earlier in Section 7.3, selection of a pixel filter can be performed by the `PixelFilter` command. The pixel filter can also be set with the `Display` command using the optional parameters `"filter"` (which takes a string giving the filter name) and `"filterwidth"` (which takes two floats that specify the x and y support widths of the filter). For example, to use the Catmull-Rom filter with width 3:

```
Display "myfile.tif" "tiff" "rgba" "filter" ["catmull-rom"]
                "filterwidth" [3 3]
```

As another example, z depth images should not overlap pixel boundaries, must be floating point, and need to use one of the depth filters (`"min"`, `"max"`, or `"average"`). Therefore, the proper `Display` command to write a z depth file is:

```
Display "myfile.z" "zfile" "z" "filter" ["min"]
        "filterwidth" [1 1] "quantize" [0 0 0 0] "dither" [0]
```

Arbitrary Output Variables

In addition to color, alpha, and depth, "global" shader variables (see Table 5.8) and any arbitrary data computed and stored in an `output` variable of the surface shader may be output as an image output stream. Output streams may contain id tags for objects; separate ambient, diffuse, and specular lighting; normal (orientation) information of the surfaces — in short, anything you can compute in the shaders.

To specify multiple output streams, you can use multiple `Display` commands. If the first character of the *name* parameter is the '+' character, the `Display` command is interpreted to indicate an *additional* output stream (rather than replacing the primary output stream). Remember that Entropy allows you to use entirely different display servers, formats, quantization levels, and even pixel filters for each output stream.

For example, to have one output stream containing RGBA quantized to 16 bits per channel, a second output stream with floating-point z depth data, and a third output stream containing floating-point color value for just the specular highlight:

```
Display "myfile.tif" "tiff" "rgba" "quantize" [0 65535 0 65535]
Display "+myfile.z" "zfile" "z" "filter" ["min"]
        "filterwidth" [1 1] "quantize" [0 0 0 0] "dither" [0]
Display "+normals.tif" "tiff" "N"
        "quantize" [0 0 0 0] "dither" [0]
Declare "Cspec" "varying color"
Display "+specpass.tif" "tiff" "Cspec" "filter" ["gaussian"]
        "filterwidth" [2 2] "quantize" [0 0 0 0] "dither" [0]
```

Notice that when output streams contain data declared by the shader, the data name is simply the name of the `output` variable of the shader, and should have its type declared with `Declare`. For shaders that do not contain the `output` variable with the correct name and type, the variable will have value 0.

Setting defaults — older options

We recommend that you set the quantization level, dither, and filtering separately with the `Display` command for each image output stream. However, for backward compatibility, we still support the "older" `Quantize` and `PixelFilter` commands (see Section 3.1.2). Remember that `Quantize` and `PixelFilter` effectively set the defaults if these parameters are not included on the `Display` command; any parameters given to `Display` override these defaults.

7.5 Entropy's Built-in Display Servers

As discussed in Section 7.4, the `Display` command takes the name of a display server, which is a plugin that actually writes the pixels in a particular format. Entropy ships with several display servers already compiled in (and hence, can write to those formats). Users or third parties may expand the formats by writing DSO's/DLL's, as described in Chapter 16. This section describes Entropy's built-in display server types.

7.5.1 `"tiff"` server

The built-in `"tiff"` display server writes output pixels as TIFF files. The driver can write pixels as 8-bit unsigned integer, 16-bit unsigned integer, or floating-point data, depending on the `"quantize"` parameter (or the `Quantize` command). The resulting output file is a totally standard scanline-oriented TIFF. Accepted optional parameters are:

Name	Type	Description
"quantize"	float[4]	Gives the conversion from floating point to integer, and indirectly, the data format of the TIFF file, as described in Section 7.4.
"dither"	float	The amplitude of random dither to add to output pixels, as described in Section 7.4.
"gain"	float	A constant multiplicative factor applied to pixel values prior to quantization.
"gamma"	float	A nonlinear gamma correction factor applied to pixel values prior to quantization.
"compression"	string	The name of the TIFF compression method — one of "none", "lzw", "zip", "packbits", or "deflate".
"rowsperstrip"	string	The number of TIFF rows per strip, expressed as a string.
"ImageDescription"	string	Optional description of the image.
"DocumentName"	string	Optional document name.
"PageName"	string	Optional "page name."
"Artist"	string	Optional artist name.
"Copyright"	string	Optional copyright notice.

If any of the optional "ImageDescription", "DocumentName", "PageName", "Artist", or "Copyright" parameters are given, their contents will be stored in the file as the TIFF tags with the same names.

7.5.2 "shadow" server

The built-in "shadow" display server writes output depth buffers as Entropy shadow map files. Since Entropy stores shadow maps as tile-oriented, floating-point TIFF files, the "ImageDescription", "DocumentName", "PageName", "Artist", and "Copyright" parameters are accepted, and will be stored as TIFF tags in the resulting file.

7.5.3 "zfile" server

The built-in "zfile" display server writes output depth buffers as "zfiles" (adhering to the format used by *PRMan* and several other renderers). Those zfiles may be converted to shadow map files using mkmip. Note that it's usually more expedient to write a shadow map file directly using the "shadow" display server.

7.5.4 "iv" server

The built-in "iv" display server communicates its output images to an interactive session of the iv image viewing program. The operation of iv is described in Chapter 8.

Chapter 8

Viewing Images with `iv`

Once you render images, you need to view them. There are dozens, or possibly hundreds, of programs that can display your ordinary images that Entropy 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.

8.1 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`.

`-justinfo`

When this flag is used, the name and resolution of each file will be printed to `stdout` (as with `-info`), but the images are never displayed and `iv` never enters into its interactive mode.

`-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.

8.2 `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.

`m`

The `m` key actives and deactivates the menu bar.

`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.

8.3 `iv` menu bar functions

Hitting the `m` key toggles a menu bar that appears across the top of the `iv` screen.

From the File menu, you can load additional images into the current `iv` session, or save the currently displayed image as a TIFF file.

Chapter 9

Compiling Shaders

Shaders are the small programs that are run to determine the material appearance of objects, refinements to their shape, attenuation of light through volumes, and the ways that light sources distribute energy in the scene.

Like with many programming language systems, these shader programs must be *compiled*. That is, they must be translated from human-readable form ("source code") into an encoded version that is ready for the renderer to process ("object code"). This extra step also serves another purpose — it allows the shader compiler to check your shader for errors before it is in the middle of rendering a frame.

9.1 Compiling Shaders with `sle`

Entropy provides a compiler for your shaders, a program called `sle`.[1] The remainder of this section describes the basic use of the `sle` program.

9.1.1 Theory of operation

By established convention, shader source code is stored in a file whose name is the same as the name of the shader, with the extension `.sl`. For example, if you had a "plastic" shader, you would store its source code in the file `plastic.sl`.

To compile the shader stored in `plastic.sl`, invoke the shader compiler, `sle`:

```
sle myshader.sl
```

This will result either in a compiled shading language object file called `myshader.sle`, 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. `sle` only can compile one file at a time.

`sle` uses a "preprocessor," `slpp`, which must be present in the `$ENTROPYHOME/bin` directory. If `sle` cannot find `slpp`, it will issue an error message. The `slpp` program is

[1] BMRT users may notice the correspondence between `sle` and BMRT's `slc` program. These are mostly the same program, but because the compiled shader format differs between Entropy and BMRT, we renamed the binaries and changed the filename extension for compiled shaders, in order to reduce confusion for people who are using both packages simultaneously.

really just an ordinary C preprocssor, so you can use all the usual C preprocessor macros (e.g., `#include` and `#define`) in your shaders.

If your shader uses the `#include` preprocessor directive to "include" another file, `sle` will need to know where to find the file. By default, it will only look in the current directory. But it's easy to specify extra directories to search for included files by using the `-I` command-line argument. For example,

```
sle -I/usr/local/shaders/include myshader.sl
```

will look in the directory `/usr/local/shaders/include` for any `#include`'d files. You can specify multiple directories with multiple `-I` arguments.

Since shaders are passed through the preprocessor, you can also define and use macros (with `-D` or with `#define` in the source code) or use "conditional compilation" (`#if`, `#ifdef`, `#ifndef`, `#else` , `#endif`). To aid in writing shaders that can be compiled for a variety of mostly-compatible renderers, we have predefined two preprocessor variables: `EXLUNA` and `ENTROPY`.

The output of `sle` is an ASCII file for a sort of "assembly language" for a virtual machine. When `entropy` renders your frame and needs a particular shader, this assembly code is read, converted to bytecodes, and interpreted to execute your shader. Because the `sle`'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 `.sle` file on any other platform. Don't worry too much that it doesn't compile to true machine code — Entropy's shader interpreter is very efficient and takes advantage of coherence in ways that a true machine compiler probably could not.

9.1.2 Command line arguments

The `sle` 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`*`=`*`value`*

> Just like a C compiler, the `-D` switch, followed immediately by a symbol name (and possibly with an initial value), will define a proprocessor macro symbol. If no *value* is supplied, the macro is defined to have value `1`.
>
> This allows you to have conditional compilation based on defined symbols using the `#if` and `#ifdef` statements in your shader source code files. The `sle` program automatically defines the symbols `EXLUNA` and `ENTROPY`.

`-U`*`symbol`*

Just like a C compiler, *undefines* a symbol. That is, a symbol that ordinarily would have been set is eliminated.

`-o` *`name`*

Specifies an alternate filename for the resulting compiled shader. Without this switch, the output file is the base name of your shader, with the extension `.sle`.

`-q`

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

`-v`

Verbose mode, lots of extra chit-chat.

`-x`

Encrypts the resulting .sle file.

9.2 Using `sletell` to list shader arguments

Sometimes you may need to know the names and default values of the parameters to a shader, but you may not have the shader source code readily available. The `sletell` 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,

```
sletell plastic
```

reports:

```
surface "shaders/plastic.sle"
    "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]
```

`sletell` 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, `sletell` 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 `sletell` will correctly report the default value.

Note that `sletell` has no trouble reporting the parameters from shaders that are encrypted using `sle -x`.

Chapter 10

Using Texture Maps

Although Entropy accepts regular scanline (or strip) oriented TIFF files as texture maps, it is able to perform certain optimizations if the TIFF files you supply happen to be tile-oriented. In particular, Entropy is able to significantly reduce the memory needed for texture mapping with tiled TIFF files.

10.1 `mkmip` Command Reference

The `mkmip` program converts scanline TIFF files into multiresolution, tiled TIFF files, zfiles into shadow maps (tiled float TIFFs), and will combine six views into a cube face environment map.

10.1.1 Texture Maps

`mkmip [options] tifffile texturefile`

This converts ordinary scanline TIFF files into tiled, multilevel MIP-map TIFFs (what Entropy calls "texture" format). Using textures in the tiled multiresolution format offers significant performance improvements over using the original image files directly. Options include:

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

where *wrapmode* is one of: `periodic`, `black`, `clamp`, or `mirror`. 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. For historical reasons, the default is `up`, but we recommend `up-` as the most intuitively-behaved option.

10.1.2 Environment Maps

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

This command takes six TIFF image files (all square and of the same resolution) and combines them into a cubeface environment map in Entropy's preferred environment map format. Options include:

`-fov` *fovangle*

Specifies the field of view of the faces.

`mkmip -envlatl [options] tifffile envfile`

This command takes a single ordinary TIFF file representing a latitude-longitude reflection map, and converts it to a "latlong" environment map in Entropy's preferred format.

`mkmip -lightprobe lpfile envfile`

This command takes a floating point TIFF in spherical "lightprobe" format (see http://www.debevec.org/Probes for the angular formulas) and converts it to an Entropy cube face environment map.

`mkmip -twofish -fov <f> file1 file2 envfile`

This command takes a two TIFF files, each a fisheye image taken 180° apart, and joins them to form an Entropy cube face environment map. The `-fov` option is required, and should contain the field of view in degrees of the images (e.g., 183 for the fisheye lenses for Nikon Coolpix digital cameras).

10.1.3 Converting zfiles to Shadow Maps

`mkmip -shadow [options] zfile shadowfile`

If you've written out a depth map using the `"zfile"` driver, this command converts it to a shadow map. This step is unnecessary if you write depth maps directly as shadow maps using the `"shadow"` driver.

10.2 unmkmip

The `unmkmip` program converts Entropy texture files, lat-long environment maps, and cube face environment maps into ordinary scanline TIFF files. Command line usage is:

NEW!

`unmkmip` *`texfile tifffile`*

Writes the highest resolution level of the texture file *texfile* as an ordinary scanline-oriented TIFF file written as *tifffile*.

`unmkmip` *`latlongfile tifffile`*

Writes the highest resolution level of the latlong environment map file *latlongfile* as an ordinary scanline-oriented TIFF file written as *tifffile*.

`unmkmip` *`cubefacefile tifffilespec`*

Writes the highest resolution level of the cube face environment map file *cubefacefile* as a series of ordinary scanline-oriented TIFF files. One file is written for each face of the cube, and the file names will be *tifffilespec*.0, *tifffilespec*.1, ... *tifffilespec*.5.

Chapter 11

Shadows, Reflections, and Global Illumination

This chapter gives an overview of how to use Entropy to achieve various *global lighting effects*. These effects include shadows, reflections, indirect illumination, and caustics. In some cases, multiple approaches (such as ray tracing and using texture maps) are discussed.

11.1 Shadows

Shadows are crucial to lighting an environment in a believable way. Entropy allows two main techniques to generate shadows: shadow maps and ray tracing.

11.1.1 Shadow Maps

Shadow maps (also sometimes called *shadow depth maps*) are a simple, relatively cheap, and very flexible means to cause a light to cast shadows. The shadow map algorithm works in the following manner. Before rendering the main image, separate images are rendered *from the vantage points of the lights*. Rather than render RGB color images, these light source views record depth only (hence the name, *depth map*). Figure 11.1 shows a simple scene with and without shadows, as well as the depth map that was used to produce the shadows. Most modeling systems geared toward generating input for Entropy will automatically position and render the depth maps for each shadowing light source.

Once these depth maps have been created, the main image is rendered from the point of view of the camera. In this pass, the light shader can determine if a particular surface point is in shadow by comparing its distance to the light against that stored in the shadow map. If it matches the depth in the shadow map, it is the closest surface to the light in that direction, so the object receives light. If the point in question is *farther* than indicated by the shadow map, it indicates that some other object was closer to the light when the shadow map was created. In such a case, the point in question is known to be in shadow.

Shading Language gives us a handy built-in function to access shadow maps:

```
float shadow ( string shadowname; point Ptest; ...  )
```

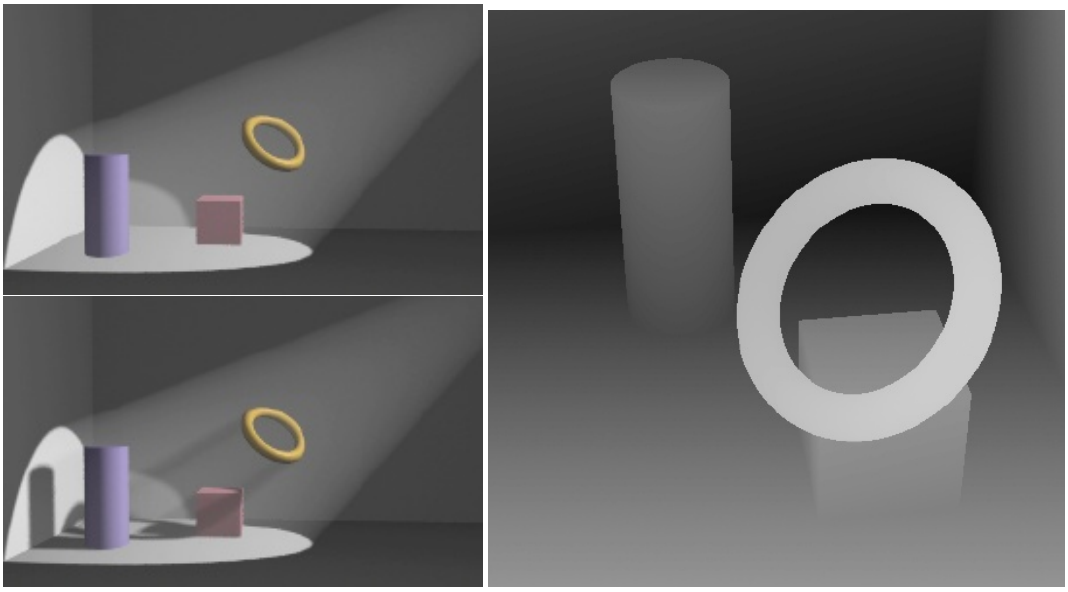


Figure 11.1: Shadow depth maps. A simple scene with and without shadows (left). The shadow map is just a depth image rendered from the point of view of the light source (right). To visualize the map, we assign white to near depths, black to far depths.

The `shadow()` function tests the point `Ptest` (in `"current"` space) against the shadow map file specified by `shadowname`. The return value is 0.0 if `Ptest` is unoccluded, and 1.0 if `Ptest` is occluded (in shadow according to the map). The return value may also be between 0 and 1, indicating that the point is in partial shadow (this is very handy for soft shadows). The `shadow()` call has several optional arguments that can be specified as token/value pairs:

- `"blur"` takes a `float` and controls the amount of blurring at the shadow edges, as if to simulate the penumbra resulting from an area light source (see Figure 11.2). A blur value of 0 makes perfectly sharp shadows; larger values blur the edges. It is strongly advised to add some blur, as perfectly sharp shadows look unnatural and can also reveal the limited resolution of the shadow map.
- `"samples"` is a `float` specifying the number of samples used to test the shadow map. Shadow maps are antialiased by supersampling, so although having larger numbers of samples is more expensive, they can reduce the graininess in the blurry regions. We recommend a minimum of 16 samples, and for blurry shadows it may be quite reasonable to use 64 samples or more.
- `"bias"` is a `float` that *shifts the apparent depth of the objects from the light*. The shadow map is just an approximation, and often not a very good one. Because of numerical imprecisions in the rendering process and the limited resolution of the shadow map, it is possible for the shadow map lookups to incorrectly indicate that a surface is in partial shadow, even if the object is indeed the closest to the light. The

solution we use is to add a "fudge factor" to the lookup to make sure that objects are pushed out of their own shadows. Selecting an appropriate bias value can be tricky. Figure 11.3 shows what can go wrong if you select a value that is either too small or too large.

- `"width"` is a `float` that multiplies the estimates of the rate of change of `Ptest` (used for antialiasing the shadow map lookup). Its use is largely obsolete and we recommend using `"blur"` to make soft shadow edges rather than `"width"`.

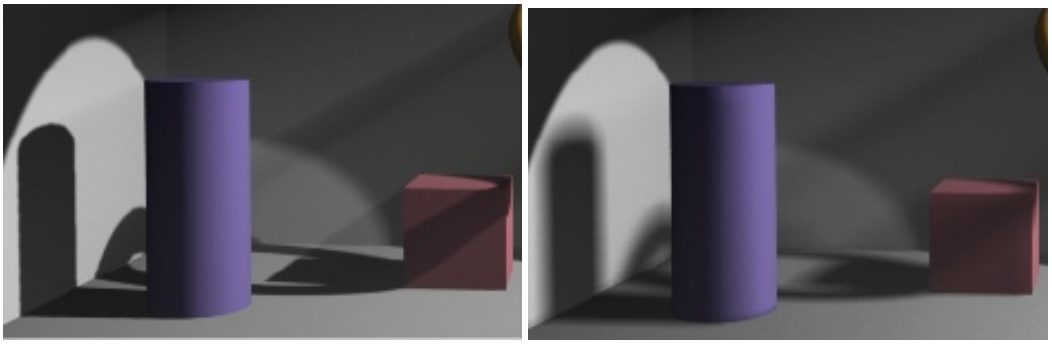


Figure 11.2: Adding blur to shadow map lookups can give a penumbra effect.

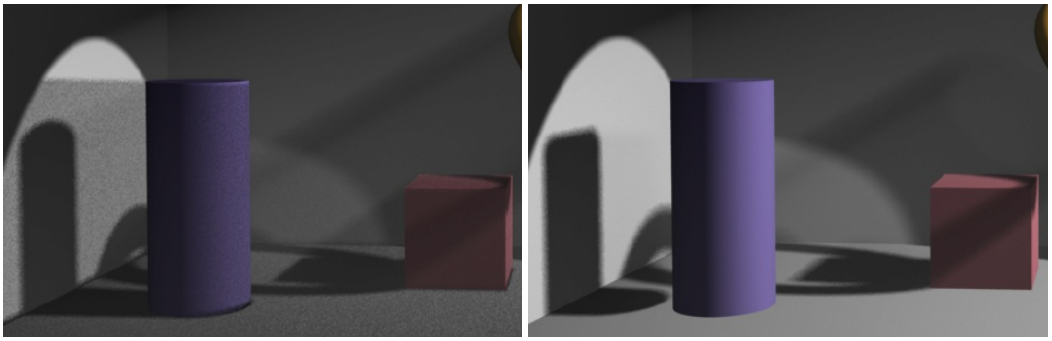


Figure 11.3: Selecting shadow bias. Too small a bias value will result in incorrect self-shadowing (left). Notice the darker, dirtier look compared to Figures 11.2 or 11.1. Too much bias can also introduce artifacts, such as the appearance of "floating objects" or the detached shadow at the bottom of the cylinder (right).

The `Ptest` parameter determines the point at which to determine how much light is shadowed, but how does the renderer know the point of origin of the light? When the renderer creates a shadow map, it also stores in the shadow file the origin of the camera at the time that the shadow map was made — in other words, the emitting point. The `shadow()` function knows to look for this information in the shadow map file. Notice that since the shadow origin comes from the shadow map file rather than the light shader, it's permissible (and often useful) for the shadows to be cast from an entirely different position than the point from which the light shader illuminates. The `shadowspot` shader

Listing 11.1 shows a basic light shader that uses a shadow map. This light shader is still pretty simple, but the entirety of Chapter 14 of *Advanced RenderMan: Creating CGI for Motion Pictures* discusses more exotic features in light shaders.

Listing 11.1 An example of a light shader that uses a shadow depth map.

```
light
spotlight ( float intensity = 1;
            color lightcolor = 1;
            point from = point "shader" (0,0,0);
            point to = point "shader" (0,0,1);
            float coneangle = radians(30);
            float conedeltaangle = radians(5);
            float beamdistribution = 2;
            string shadowname = "";
            float  shadowsamples = 16;
            float  shadowblur = 0;
            float  shadowbias = -1e9;)
{
    uniform vector axis = normalize(to-from);

    illuminate (from, axis, coneangle) {
        float cosangle = (L . axis) / length(L);
        float atten = pow (cosangle, beamdistribution) / (L . L);
        atten *= smoothstep (cos(coneangle), cos(coneangle-conedeltaangle),
                             cosangle);
        Cl = atten * intensity * lightcolor;
        if (shadowname != "")
            Cl *= 1 - shadow (shadowname, Ps, "samples", shadowsamples,
                              "blur", shadowblur, "bias", shadowbias);
    }
}
```

Here are some tips to keep in mind when rendering shadow maps:

- Select an appropriate shadow map resolution. It's not uncommon to use 2k $\times$ 2k or even higher-resolution shadow maps for film work. But choosing a resolution too high is wasteful; experiment to find an appropriate resolution that avoids artifacts.
- View the scene through the "shadow camera" before making the map. Make sure that the field of view is as small as possible, so as to maximize the effective resolution of the objects in the shadow map. Try to avoid your objects being small in the shadow map frame, surrounded by lots of empty unused pixels.
- Remember that depth maps should have a filter width of 1 and should use one of the depth filters: `"min"`, `"max"`, or `"average"`. Entropy will warn you if you choose inappropriate settings for a shadow map pass. For example, the RIB file for the shadow map rendering ought to contain the following options:

```
PixelSamples 1 1
```

```
PixelFilter "min" 1 1
Display "shadow.sm" "shadow" "z"
```

- In shadow maps, only depth is needed, not color. To save time rendering shadow maps, remove all lights, all `Surface` calls (except for surface shaders that modify opacity in a complex way), and increase the number given for `ShadingRate` (unless the object contains a significant amount of displacement).
- When rendering the shadow map, only include objects that will actually cast shadows on themselves or other objects. Objects that only receive, but do not cast, shadows (such as walls or floors) can be eliminated from the shadow map pass entirely. This saves rendering time when creating the shadow map and also eliminates the possibility that poorly chosen bias will cause these objects to incorrectly self-shadow (since they aren't in the maps anyway).
- It is easiest to use the `"shadow"` display type, generating an Entropy shadow map directly. Alternately, you could generate file with the `"zfile"` format, but in that case you must perform an additional step to transform it into a full-fledged shadow map (much as an extra step is often required to turn ordinary image files into texture maps). The command to perform this operation is:

  ```
  mkmip -shadow shadow.zfile shadow.sm
  ```

  This is not necessary if you generate a shadow map directly using the `"shadow"` display server.

11.1.2 Ray Traced Shadows

Entropy also supports ray-traced shadows: instead of using a shadow map, the path joining the light source to the point being shaded is tested for occlusion against other objects in the scene.

Entropy extends the `shadow()` call to support ray tracing. If the `shadowname` parameter is the special name `"shadow"`, ray tracing will be used instead of looking up from a shadow map file. Thus, the preferred method for a light shader to produce shadows using ray tracing is simply to use the `shadow()` call. The advantage to this approach is that there is *no difference* between a light shader that makes shadows using depth maps versus a light shader that uses ray-traced shadows — indeed, a single shader (such as `shadowspot` in Listing 11.1) can accommodate both, switching between techniques depending on the name of the shadow passed as a parameter.

When using ray traced shadows, the `"blur"`, `"bias"`, and `"samples"` parameters work analogously to their use for shadow depth maps. There are some minor differences, however.

The `"blur"` parameter causes a penumbra effect (in fact, a much more realistic appearance than blurred shadow maps), but is measured differently. For shadow maps, the `"blur"` amount is interpreted as a fraction of the resolution of the shadow map camera,

whereas when ray tracing, `"blur"` is interpreted as the apparent angular size of the light source (1.0 being a 90 degree light source). In both cases, 0 means perfectly sharp and larger values quickly become blurrier. But you should not expect the same blur value to have an identical appearance when using ray tracing versus when using shadow maps.

The `"samples"` parameter controls the number of ray samples that are used to antialias the shadow edge and to give the appearance of penumbra (for nonzero `"blur"`). Note that to an even greater degree than with shadow maps, ray traced shadows become much more expensive as `"samples"` increases. Furthermore, you must beware of the potential multiplicative effect that occurs when the light is run multiple times with `Attribute "light" "nsamples"` — the shadow `"samples"` are the number of rays samples *each time* the light is run.

The `"bias"` parameter is also honored (and quite useful) for ray-traced shadows, although the amount of bias required to eliminate self-shadowing artifacts is generally much lower with ray-traced shadows than with shadow maps.

Another thing to keep in mind when using ray-traced shadows is that objects will only cast ray-traced shadows if they are visible to shadow rays. In other words, they must have been tagged with:

```
Attribute "visibility" "shadow" [1]
```

11.1.3 Area Lights

In the real world, shadows are generally not perfectly sharp, but rather have a *penumbra* — an area of partial shadow caused by the fact that the light source itself is not really a single point, but covers an extended angle. The `"blur"` parameter for shadow maps can simulate penumbra rather crudely by simply blurring the shadow shape. When using ray-traced shadows, the `"blur"` parameter can be used to sample the light direction over a solid angle, giving a somewhat more accurate shadow effect but assuming a spherical source of fixed solid angle width. For even more accurate penumbra effects, Entropy supports true area lights.

An *area light* is a light source that is associated with a specific geometric primitive (for example, a `Sphere`, `NuPatch`, `PointsPolygons`, or other primitive). When the light shader is run, the light direction is chosen to be distributed over the primitive. This can give a very pleasing, rich appearance to the lighting, as seen in Figure 11.4. The only way to get the correct shadowing effect with area lights sources is to use ray-traced shadows.

Declaring an area light source is similar to an ordinary `LightSource` declaration, with some minor differences:

- The light is declared with `AreaLightSource` instead of `LightSource`.
- Subsequently declared geometric primitives are added to the light source geometry, until the end of the enclosing `AttributeBegin`/`AttributeEnd` block.
- Because the light is enclosed inside an `AttributeBegin`/`AttributeEnd` block, you must be careful to use `Illuminate` to turn the light "on" for the rest of the scene. (Remember that a light is off by default once you exit the Attribute block in which it was declared.)

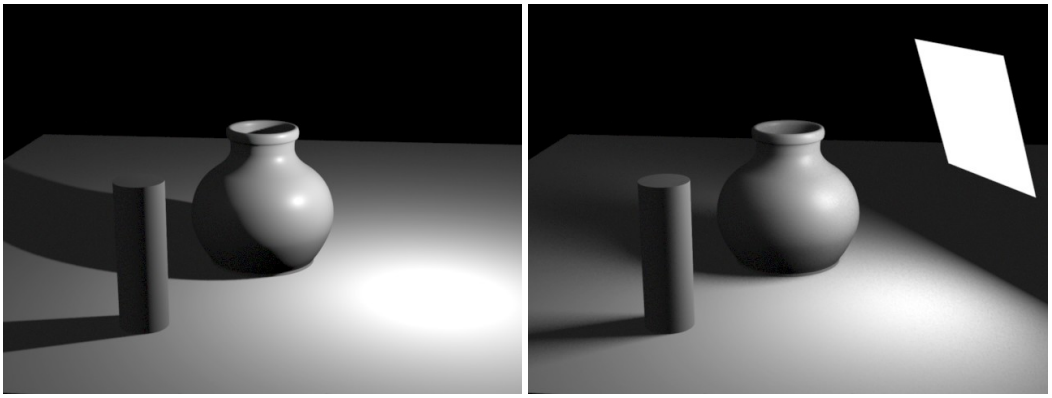


Figure 11.4: True area lights can drastically change the character of light and shadow.

- To reduce noise in the shadow penumbra regions, you will almost certainly want to fine-tune the number of samples for the area light using `Attribute "light" "nsamples"` (see Section 3.2.2).
- To give the appearance of a luminaire, the surface shader assigned to the light source geometry should be `constant.sl` or some other self-illuminating or glowing shadow.

An example declaration of an area light source (from the scene shown in Figure 11.4) is shown below:

```
AttributeBegin
  Attribute "light" "nsamples" [16]
  AreaLightSource "arealight" 1 "intensity" [30]
        "string shadowname" ["shadow"] "float shadowbias" [.15]
  Translate 4 2 4
  Rotate 90 1 0 0
  Rotate 60 0 1 0
  Rotate -20 1 0 0
  Scale 1.5 1.5 1.5
  Surface "constant"    # Give it a glowing appearance
  Patch "bilinear" "P" [-1 1 0  1 1 0  -1 -1 0  1 -1 0]
AttributeEnd
Illuminate 1 1     # Make sure the light is turned on
```

Inside the light shader of an area light (i.e., the shader declared with `AreaLightSource`), the `P` variable is a position on the light geometry (chosen by the renderer), and `N` is the normal of the light geometry at that point. To give the proper effect, the light shader assigned to the area light should be sure to illuminate the scene from `P`. The `arealight.sl` shader shown in Listing 11.2 is a simple light shader suitable for use as an area light source.

IMPORTANT NOTE: In the current implementation, only geometric primitives with 2D parameterizations can be used as area light geometry. This means that `NuPatch`, `Polygon`, `Patch`, and all quadrics (`Sphere`, `Cylinder`, and the like) are fine to use as area light geometry. But primitives that are made from lots of pieces and therefore lack a consistent

Listing 11.2 `arealight.sl`: A simple light shader suitable for use on an area light source.

```
light
arealight (float  intensity = 1;
           color  lightcolor = 1;
           string shadowname = "";
           float  shadowsamples = 1;
           float  shadowblur = 0;
           float  shadowbias = -1e9; )
{
    illuminate (P, N, 1.5707963 /* PI/2 */) {
        Cl = (intensity / (L.L)) * lightcolor;
        if (shadowname != "")
            Cl *= 1 - shadow (shadowname, Ps, "samples", shadowsamples,
                              "blur", shadowblur, "bias", shadowbias);
    }
}
```

parameterization — `PointsPolygons`, `PointsGeneralPolygons`, `SubdivisionMesh`, `Curves`, and `Points`— cannot be used as area light geometry. The next major release of Entropy will certainly lift this restriction.

11.2 Reflections

Many surfaces are polished to a sufficient degree that one can see coherent reflections of the surrounding environment. The presence and appearance of reflections is controlled by the surface shader. Entropy provides for three different methods for making reflections in surfaces: environment mapping, flat reflection mapping, and ray tracing.

Environment mapping work best if the reflective object is curved, and is really the only applicable technique if the environment is painted or is captured from a real scene.

Flat reflection mapping is generally superior to environment mapping if the reflective object is flat (like a floor or a large flat mirror), but is tricky to do properly if there are objects behind the mirror, or if the reflective object is not almost perfectly flat.

Ray tracing is much slower and more memory-intensive than environment or reflection mapping, but is more geometrically accurate and may be the only method that looks right for tricky situations such as mutually-reflective objects.

11.2.1 Environment Maps

Environment mapping takes a pre-rendered, captured, or painted image of the reflective environment and looks up texture indexed by a direction vector, thus simulating reflection. The environment map is either rectangular (called a latitude-longitude environment map) or composed of six axis-aligned directions from a particular point (called a cube-face environment map).

Environment maps may be completely synthetic, from six rendered images, or a painting. Environment maps may also be captured from a real environment, as six 90° photos, two fisheye lenses, or a single spherical "light probe" map.

See Section 10.1 for details on converting various formats into Entropy environment maps. An example of an "unwrapped" environment map, captured from two fisheye lens images of a real scene, is shown in Figure 11.5. Its use on a reflective object is shown in Figure 11.6. Most modeling systems geared toward making input for Entropy will have an automatic facility for generating cube face environment maps of your synthetic scene for any object that is reflective.

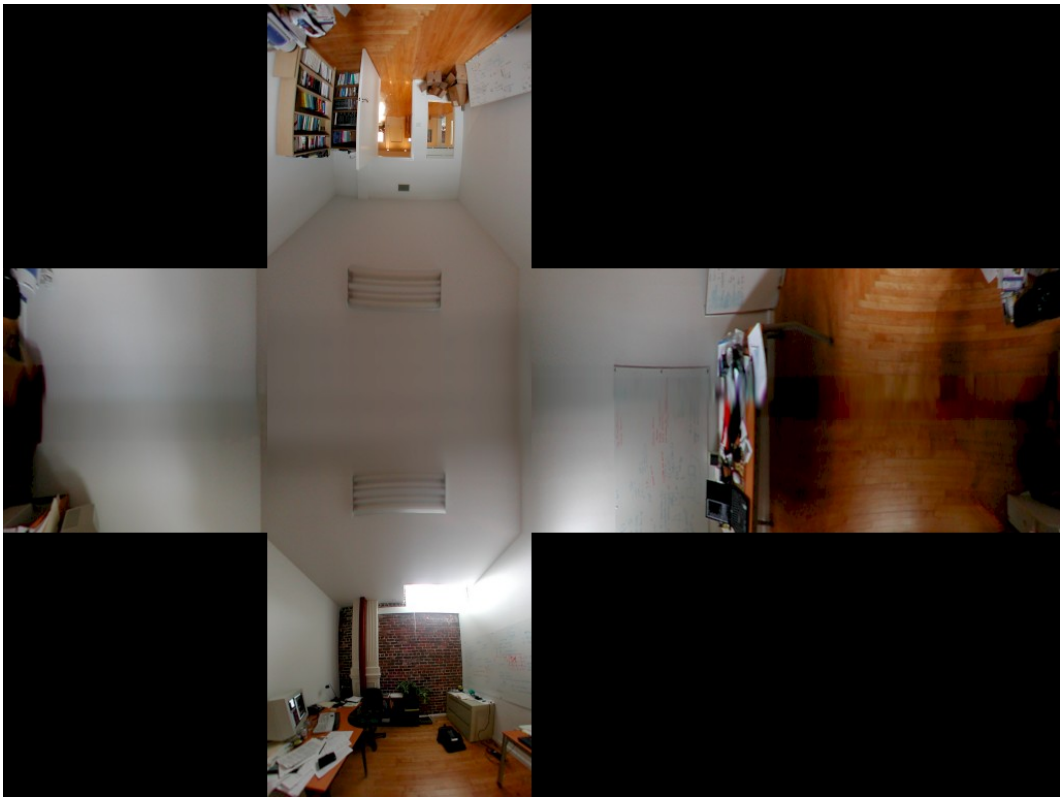


Figure 11.5: An example environment map stitched together from two fisheye lens views of a room.

Accessing an environment map from inside your shader is straightforward with the built-in `environment` function:

```
type environment (string envname, vector R, ...)
```

The `environment()` function is quite analogous to the `texture()` call in several ways:

- The return type can be explicitly cast to either `float` or `color`. If you do not explicitly cast the results, the compiler will try to infer the return type, which could lead to

Figure 11.6: Use of the environment map from Figure 11.5 to make reflections.

ambiguous situations.

- For environment maps, the texture coordinates consist of a direction vector. As with `texture()`, derivatives of this vector will be used for automatic filtering of the environment map lookup. Optionally, four vectors may be given to bound the angle range, and in that case no derivatives will be taken.

- The `environment()` function can take the optional arguments `"blur"`, and `"width"`, which perform the same functions as for `texture()`.

Environment maps typically sample the mirror direction, as computed by the Shading Language built-in function `reflect()`. For example, Listing 11.3 shows a typical use of `environment()` to make a shiny object.

Environment maps must be indexed by a vector in the same coordinate system that they were created in (typically `"world"` space). If you index the environment map in the wrong space (particularly `"current"` space), you could get very strange results with your reflections.

Note also that environment maps are indexed by direction only, not position. Thus, not only is the environment map created from the point of view of a single location, but all lookups are also made from that point. Alternatively, one can think of the environment map as being a reflection of a cube of infinite size. Either way, two points with identical mirror directions will look up the same direction in the environment map. This is most noticeable for flat surfaces, which tend to have all of their points index the same spot on the environment map. This is an obvious and objectionable artifact, especially for surfaces like floors, whose reflections are *very* sensitive to position.

One solution to this problem is to attempt use reflection mapping (if the reflective object is flat), or to use ray tracing. Another solution is to stick to environment mapping, but to try to take the parallax into account. The `reflections.h` file that is distributed with Entropy contains a function called `Environment()` that, in addition to a reflection direction, also takes the name of the environment map's coordinate space and a radius of the projection (essentially the "size of the room"). This information is used to approximate the parallax effect.

Listing 11.3 `shinymetal.sl`: A simple surface shader that makes a mirror-like surface with `environment()`.

```
surface
shinymetal (float Ka = 1, Ks = 1, Kr = 1;
            float roughness = .1;
            string envname = "";
            string envspace = "world";)
{
    vector V = normalize(I);
    normal Nf = faceforward (normalize(N),V);
    vector D = vtransform (envspace, reflect (V, Nf));

    color env;
    if (envname != "")
        env = Kr * color environment (envname, D);
    else env = 0;

    Oi = Os;
    Ci = Os * Cs * (Ka*ambient() + Ks*specular(Nf,-V,roughness) + env);
}
```

11.2.2 Flat Surface Reflection Maps

For the special case of flat objects (such as floors or flat mirrors), there is an even easier and more efficient method for producing reflections, which also solves the problem of environment maps being inaccurate for flat objects.

For the example of a floor, we can observe that the image in the reflection would be identical to the image that you would get if you put the camera underneath the floor. This geometric principle is illustrated in Figure 11.7. Alternately, you would achieve the same effect if you made a mirror-image copy of the scene under the floor.[1]

Most modeling systems geared toward making input for Entropy will have an automatic facility for generating reflection maps for any flat, reflective objects.

Once we create this reflection map, we can turn it into a texture and index it from our shader. Because the pixels in the reflection map correspond exactly to the reflected image in the same pixels of the main image, we access the texture map by the texture's pixel coordinates, not the s, t coordinates of the mirror. We can do this by projecting `P` into `"NDC"` space:

```
    /* Transform to the space of the environment map */
    point Pndc = transform ("NDC", P);
    float x = xcomp(Pndc), y = ycomp(Pndc);
    Ct = color texture (reflname, x, y);
```

[1] Note that the reflection of a scene in a flat mirror is the same thing you'd see if the mirror were a *window* to another room in which everything were reversed.

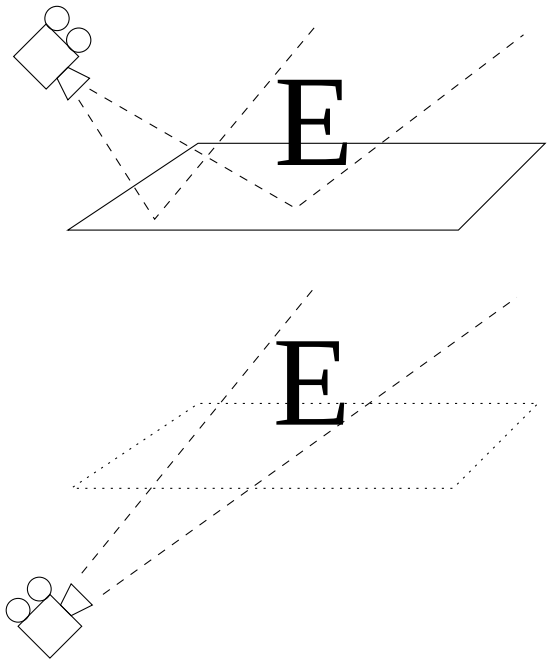


Figure 11.7: Creating a mirrored scene for generating a reflection map. On top, a camera views a scene that includes a reflective floor. The view of the reflection is the same as if the camera was viewing the room from below the floor.

11.2.3 Ray traced reflections and refractions

There are times when environment or reflection mapping is not adequate. Reflection maps are really only applicable if the reflector is perfectly flat. Environment mapping doesn't work well for nearly flat objects, and is incorrect for objects that touch (or nearly touch) the reflective object. There can be severe parallax problems. Neither technique works well for scenes in which there are mutually-reflective objects, or in which an object must reflect another part of itself. Also, it can be difficult (though certainly not impossible) to use environment maps to convincingly model refraction.

Entropy also supports ray-traced reflections and refractions by extending the `environment()` routine.[2]

If the environment name supplied to the `environment()` is the special name `"reflection"`, then the `environment()` function will ray trace in the reflection direction instead of looking up the result from an environment map file. In other words, the `shinymetal.sl` environment mapping shader in Listing 11.3 can be used for ray tracing *without modification*. We only need to use `"reflection"` as the environment map name, and also ensure that the ray direction is expressed in `"current"` space (instead of the usual transformation to `"world"` space for environment map file lookups). In other words, specifying this shader in the scene file as:

[2]There are other functions that can be used for ray tracing, as described in 5.6.8, but for most basic uses of ray tracing, `environment()` is the preferred mechanism.

```
Surface "shinymetal" "envname" ["room.env"] "envspace" ["world"]
```

will get reflections from the environment map `"room.env"`, whereas

```
Surface "shinymetal" "envname" ["reflection"] "envspace" ["current"]
```

would use ray-traced reflections.

Most of the optional environment map parameters are implemented analogously when performing ray tracing. The most useful optional ray tracing parameters are `"blur"`, `"samples"`, and `"bias"`. If you want blurry ray-traced reflections, there's no reason to write a loop with carefully selected samples. Rather, just call `environment()` with an appropriate `"blur"` amount, exactly as you would if you were using an environment map file. The renderer does the rest. Increasing the `"samples"` will antialias your reflections and reduce the noise in the reflection blur (but of course will increase the expense of the shader). The `"bias"` parameter works much like it does with shadow mapping — the bias amount can be used to reduce self-reflection artifacts (sometimes known as "surface acne").

Remember that ray tracing will produce strange results if the ray direction is not expressed in `"current"` space!

11.3 Indirect illumination

Direct illumination refers to light leaving a light source, traveling in a straight line, and arriving (possibly shadowed) at an object seen by the camera. In the real world, much of the light arriving at objects did not come via a straight line from the light. Rather, light will bounce between objects in the scene. This *indirect illumination* can be computed by Entropy, albeit at an additional expense. Figure 11.8 shows the contribution of indirect illumination to an example scene.

Entropy supports indirect illumination using a Monte Carlo technique. Rather than enmeshing the scene and solving the light transport up front (as finite element-based radiosity techniques would do), the Monte Carlo 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.

To use the Monte Carlo irradiance calculations for global illumination, you need to follow the following steps:

1. **Turn on indirect illumination for the scene**

   Add a light source to the scene using the `"indirect"` light shader. The light does not have any parameters.

   ```
   LightSource "indirect" 42
   ```

2. **Receivers**

   All objects that are illuminated by the `"indirect"` light will receive indirect illumination. 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 for those objects.

Figure 11.8: Left: a character lit by a single light source. Much of the character is not illuminated directly by the light. Right: the sky hemisphere is made emissive (by using the `"constant.sl"` shader) and indirect illumination is used. (Model courtesy of Discreet.)

3. **Reflected objects**

   Indirect illumination is, in many ways, just a very blurry ray-traced reflection. As such, only objects that would show up in ray-traced reflections will contribute to the indirect illumination. Thus, all objects that can "be seen" by the indirect illumination rays must be tagged as being visible in reflections:

   ```
   Attribute "visibility" "reflection" [1]
   ```

4. **Adjust Parameters**

   It's expensive to recompute the indirect illumination at every pixel, so it's only done sparsely, with the results interpolated or extrapolated. There are several time/quality controls controlling how often this recomputation is done. These attributes, which can be modified on an object-by-object basis, 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. Smaller numbers are higher quality, but use more memory and take longer to render.

`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.

Speeding up walk-throughs with global illumination

There is a way to greatly speed up global illumination calculations for there special case where you are rendering an animation in which no objects that interact with global illumination move. An example of this is a camera moving through a static environment. Two options make it possible to store and re-use indirect lighting computations from previous renderings:

`Option "indirect" "string savefile" [""]`

If you specify this option with a non-empty string, 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" [""]`

If you specify this option with a non-empty string, the irradiance data cache will 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.

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. You must make the judgment about whether this time savings is worth the additional inaccuracy.

Parameter selection strategy

It can be hard to tune the several parameters that control the speed and quality of the indirect illumination. We recommend choosing one of two strategies.

Strategy 1: Sparse Computation In this strategy, we try to save time by utilizing the caching computations.

```
Attribute "indirect" "float maxpixeldist" [20]
Attribute "indirect" "float maxerror" [0.25]
Attribute "indirect" "integer nsamples" [256]
```

For sparse computation, you will need a larger number of samples (typically at least 256). One or both of the `"maxpixeldist"` and `"maxerror"` parameters can be adjusted as time versus quality tradeoffs (in both cases, lower numbers are higher cost and higher quality).

Strategy 2: Final Gather In this strategy, we force the indirect illumination to be fully recomputed at every shading sample:

```
Attribute "indirect" "float maxpixeldist" [0]
Attribute "indirect" "integer nsamples" [64]
```

By setting the `"maxpixeldist"` to 0, we force recomputation at every shading sample. The `"maxerror"` parameter is unused. We can set `"nsamples"` relatively low (64 is a good starting point), and then adjust it as a straightforward time versus quality knob (larger values are higher cost and higher quality).

The "sparse computation" strategy is much faster in many circumstances. However, the time/quality parameters can be counterintuitive, and this method is sometimes plagued by splotchy artifacts. The "final gather" strategy is more expensive computationally, and will have minimal speed-ups with "seed" files because it does not the cache for the first bounce, but this technique will be free of but blotchy artifacts. It can be noisy, depending on nsamples, but you may find the noise much less objectionable than the splotches. We recommend the "final gather" strategy if you can afford the render times.

11.4 Caustics

Figure 11.9 shows a vase made of plastic, and then changing the material properties to that of glass. We can see that when the vase is made of glass, the shadow seems too "dense." We expect some of the light to make it through the glass anyway, and because of the refractive nature of the glass, it should be concentrated or focused at particular points on the floor.

Caustics are bright spots caused by the focusing of light that is refracted or specularly reflected, particularly by curved objects. This section discusses two ways to generate caustics — one method based on physical simulation with photon mapping, and another very fake method that may nonetheless be an adequate (and less expensive) substitute.

Figure 11.9: A plastic vase (left) and a glass vase (right).

11.4.1 Real caustics

Entropy can compute caustics by simulating the action of light with *photon mapping*. This involves several steps: turning on caustics in the scene, specifying which objects receive caustics, specifying which lights contribute to caustics, and specifying which objects reflect light to form caustics.

1. **Turning on caustics for the scene**

   To turn caustics on in your scene, you must declare a light source with the special `"caustic"` shader (like `"indirect"`, it's really a hook into various renderer internal magic):

   ```
   LightSource "caustic" 87
   ```

2. **Caustic lights**

   For any light sources that should reflect or refract from specular objects, thereby forming caustics, you will need to set the number of photons:

   ```
   Attribute "light" "integer nphotons" [50000]
   ```

   This indicates that a subsequently declared light should shoot 50,000 photons in order to calculate caustics. If you do not set this option, the default is 0, which means that a light will 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). This attribute binds to the light, so it's important to have it declared before (and within the scope of) the light source that should make caustics.

3. **Reflectors and Refractors**

   It's important to give it a few hints about which objects actually specularly reflect or refract lights. For reflective caustics (like mirrors):

```
Attribute "caustic" "color specularcolor" [.9 .9 .9]
```

and for refractive caustics (like glass):

```
Attribute "caustic" "color refractioncolor" [.9 .9 .9]
Attribute "caustic" "float refractionindex" [1.5]
```

Obviously, you can customize the reflection and refraction colors and the index of refraction to anything you like. Be very careful with specular or refraction colors that have components with values greater than 1 — such objects are increasing the energy of light as it bounces off them, which doesn't happen in the real world and can lead to strange results in your scene.

Any object that does not have either the `"specularcolor"` or `"refractioncolor"` set (or has both set to `[0 0 0]`) will not reflect or refract caustics. (They may still reflect or refract images of the scene, depending on their shaders.)

Also, all objects that reflect or refract caustics must be visible to reflection rays:

```
Attribute "visibility" "reflection" [1]
```

4. **Blockers**

   All objects that can *block* caustics (i.e., shadow their transport from light to reflector, or from reflector to receiver) must be marked as visible to shadow rays:

   ```
   Attribute "visibility" "shadow" [1]
   ```

5. **Receivers**

   You should use `RiIlluminate` to turn the master `"caustic"` light on for objects that receive caustics (that is, those objects that have the bright caustics focused on them). Turn it off for objects that should not not receive caustics. Illuminating just the objects that are known to receive caustics can save lots of rendering time.

   Also, all objects that receive caustics must be visible to reflection rays:

   ```
   Attribute "visibility" "reflection" [1]
   ```

   There are also two attributes that can affect the time/quality trade-offs for caustic receivers:

   `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.

The final image using this technique is shown in Figure 11.10.

Figure 11.10: “Real” caustics.

11.4.2 Fake caustics

Perhaps even more so than refractions or shadows, it is very hard for the viewer to reason about the correct appearance of caustics, and therefore potentially easy to fake. A rather cartoonish understanding of caustics might be as follows: in the interior where a cast shadow would ordinarily be, we see a bright spot. Can we mimic the appearance without simulating the phenomenon? The main visual phenomenon appears to be that within the normal shadow region, we expect some of the light to make it through the glass anyway, but distorted and perhaps concentrated in the center.

Observe that in the `spotlight` shader (Listing 11.1) we blocked light by using a `shadow()` call:

```
Cl *= 1 - shadow (shadowname, Ps, "samples", samples,
                  "blur", blur, "bias", bias);
```

If `1 - shadow()` is the amount of light that gets through, then a light that multiplied by `shadow()` instead would illuminate only the occluded region. To restrict ourselves to the interior of the occluded region, we can blur the boundaries of the shadow map lookup and threshold the results:

```
float caustic = shadow (shadowname, Ps, "samples", samples,
                        "blur", blur, "bias", bias);
caustic = smoothstep (threshold, 1, caustic);
Cl *= caustic;
```

We can also add a little noise to simulate distortion in the glass:

```
point PL = transform ("shader", Ps);
caustic *= noiseamp * pow (noise(PL*noisefreq), noisepow);
```

We apply the caustic light only to the floor, using the very same shadow map that our ordinary spotlight was using to create the shadows. The results can be seen in Figure 11.11. Compare to the image computed from the "real" caustic computations. In many situations, the "fake" technique can be yield satisfactory results more quickly than the photon map simulation; in other cases, only the simulation will do. The shader for the fake caustic light can be found in Entropy's `shaders` directory as `fakecausticspotlight.sl`.

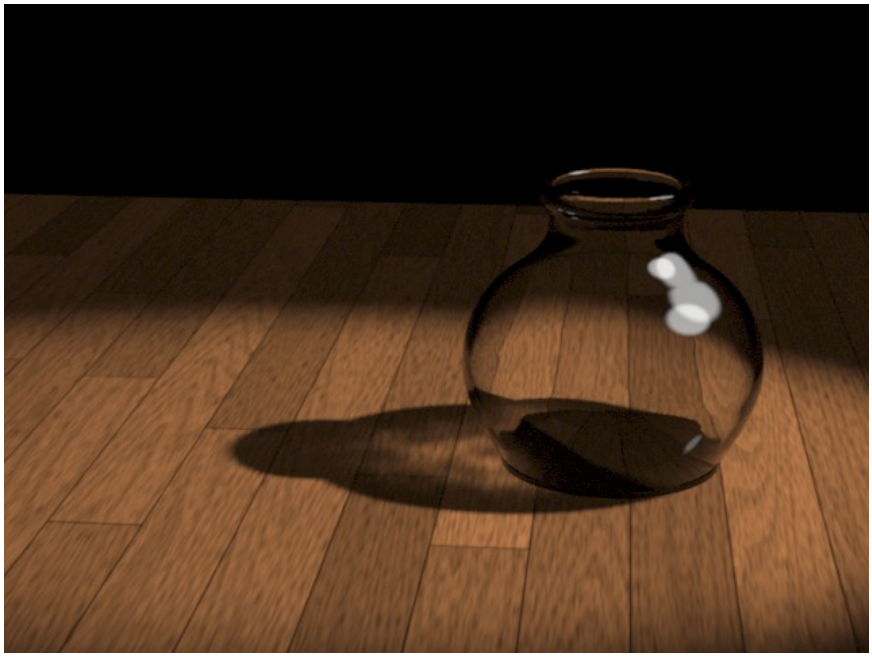


Figure 11.11: Fake caustics, including noise.

11.5 HDRI / Environment lighting

It is often useful to light objects using an environment map as the light source. This kind of technique is particularly handy for inserting synthetic objects into real scenes or lighting environments. To have a remotely accurate representation of a scene's lighting, 8-bit integer values of 0-255 will not do. Instead, to achieve a reasonable quality level, an *HDR* (high

dynamic range) representation is needed, which really means that you need some kind of floating point environment map.

Once you have an HDR environment map representing the lighting in your scene, there are several techniques that Entropy supports to light scenes using that map. The easiest two techniques are described below.

11.5.1 Technique 1: Sky sphere

One simple way to light a scene using an environment map is to simply place a sphere around the whole scene, which has a surface shader that is emissive (glows, without needing external lights), and let the indirect illumination do its job. This involves the following steps:

1. Place a large sphere around the whole scene.
2. The surface shader of the sphere should color itself using a blurred lookup from the environment map, without needing any other lights. The `"envsurf"` shader that comes with Entropy is ideal. For example:

```
Surface "envsurf" "string envname" ["hdroom.env"] "float blur" [0.1]
```

3. Be sure to make the sphere visible to reflection rays (so it shows up in the indirect illumination). If you don't want the sphere (and the image on it) to be visible to the camera, make sure to make it invisible to the camera. See Section 3.2.2 for the commands to make objects visible (or not) to reflections or the camera.
4. Make sure you are using indirect illumination by adding the `"indirect"` light to the scene, making objects visible to reflections, and tuning the indirect parameters appropriately (see Section 11.3).

The main drawback to this technique is that you may require a large number of samples for your indirect illumination. The indirect `"nsamples"`, `"maxpixeldist"`, and `"maxerror"` should be tuned as described in Section 11.3.

11.5.2 Technique 2: As a light source

A second technique is to make the environment lighting look like an actual light source in the scene. The `"envlight"` light source shader that comes with Entropy is an example shader that does exactly this, by looking up the lighting from the environment map. The lighting is scaled by the "local occlusion" (calculated with the `occlusion()` function) and the direction of the environment map lookup is perturbed toward the average unoccluded direction. The steps for using this technique are straightforward:

1. Add a light using the `"envlight"` shader, specifying the name of the environment map as the `"envname"` parameter.
2. Make objects visible to the occlusion calculations by making them visible to reflection rays.

3. Adjust the indirect attributes described in Section 11.3. The various `"indirect"` attributes affect the occlusion caching analogously to the way they affect indirect illumination. Just as with indirect illumination, we recommend using a "final gather" step, but you might be able to save time by using sparse sampling and the occlusion cache.

With this technique, it is not necessary to add an `"indirect"` light (but is certainly allowed if you want to also account for light interreflected between objects).

Chapter 12

Optimizing Your Renderings

Rendering can take a very long time, even considering that Entropy can render complex scenes very efficiently, perhaps more so than almost any other renderer. Choices in scene design and Entropy invocation options can make a huge difference in the amount of time and memory required to render. Careful choices can make the difference between fast and slow rendering, and sometimes between slow rendering and not being able to render a scene at all.

The remainder of this chapter lists a number of things to keep in mind for efficient rendering. If your renders are taking too long, the first thing you should do is run through this checklist to see if any of these tips are applicable to your scene.

12.1 Don't ray trace if you don't have to

Ray tracing is expensive. It takes lots of time to trace the rays, and it gets more expensive the more geometry is in your scene. Ray tracing also greatly increases the amount of memory required to render your scene — since a ray-traced reflection or shadow could hit an object at any time, it's almost never safe to throw the object away, even if Entropy is no longer working on the section of the screen that the object occupies. When few objects are ray traced, Entropy can be extremely aggressive about minimizing the number of geometric primitives that are in memory at any one time.

There are several ways that ray tracing can be avoided:

- Use environment maps (for curved objects) or reflection maps (for flat objects) instead of ray-traced reflections. It is a rare circumstance when the viewer can really tell the difference between a ray-traced reflection and an environment map.

  When it doesn't work: Environment mapping breaks down when you have two objects that are mutually reflective, or an object that must reflect a part of itself, or if a reflective object is touching (or nearly touching) another object. But when these circumstances are not present

- Use shadow depth maps instead of ray-traced shadows.

  When it doesn't work: If you are using area lights or HDRI environment lighting, and want the shadows and penumbras to look good, you probably need ray tracing.

- Use bounce lights instead of global illumination.

  Sure, there are times when you really want to use global illumination. There are other times when you can carefully place ordinary lights to achieve the same appearance.

- Use fake lights instead of caustics.

  Section 11.4.2 gives an example of a way that caustics can be simulated without actually performing photon mapping.

12.2 If you must ray trace, don't do it wastefully

- Only cast shadows from key lights. Dim lights used for fill or bounce effects often don't need to be shadowed at all, let alone shadowed with ray tracing.

- Be careful with `"samples"` in ray-traced reflections and shadows. Expense of ray tracing is directly proportional to the number of rays. If reflections and shadows are not blurry, you may only need a small number of samples,

- Use `Attribute "visibility"` to tag as "ray-traceable" only those objects that need to be seen in reflections, or that need to actually cast shadows. For example, in an enclosed room, the walls, floor, and ceiling generally do not cast shadows (what would they cast shadows on?), and thus do not need to be marked as visible in ray-traced shadows. If the only reflective object in the scene is a flat mirror, objects behind the mirror cannot show up in the reflection, and thus do not need to be ray traceable.

  In general, the fewer objects are that are ray traceable, the faster your scenes will render, and the less memory Entropy will require to render them.

- For objects that must be ray-traceable and have displacement shaders on them, consider using `Attribute "render" "tracedisplacements" [0]`. This makes ray tracing much cheaper, as long as you're willing to accept bump mapping rather than true displacements, as seen in the reflections. Note that this attribute does not affect the way objects appear to the camera.

- Use stand-ins. `Attribute "visibility"` can be used to make a complex scene that's visible only to the camera, but not in any shadows or reflections, and a much simpler version of the scene that is only ray traced, and not visible to the camera at all.

- Limit the ray depth with `Option "limits" "raydepth"`. Do you really need to see reflections in reflections? Similarly, when using global illumination, `Option "maxbounce"` should be kept low, unless you can really see the difference made by extra bounces.

- If you are using global illumination and not much in the scene is moving (especially if just the camera is moving), make sure to use `Option "savefile"` and `Option "seedfile"`.

- If using global illumination or caustics, carefully tune the controls described in Sections 3.1.3 and 3.2.2 so that they are as cheap as possible without compromising noticeable quality.

- Some users have reported being able to speed up indirect illumination by increasing the `"minshadowbias"`.

- For ray tracing, it is much more efficient to use an actual `Sphere` than a `NuPatch` that is shaped like a sphere. And both of those choices are much more efficient than using lots of polygons to approximate the shape of a sphere.

12.3 Use short-cuts when computing maps

- Choose appropriate resolutions.

  Don't compute higher resolution shadow, reflection, and environment maps than you have to. Use your best judgment, but try to have the shadow or environment map resolution be not much higher than the number of pixels that the cast shadow or reflective object will be in the final image. For example, if you are rendering your final images at video res (720×486) and you have a reflective object covering at most 1/4 of the screen, rendering the environment map at 512×512 on each side should be more than good enough.

- Not all lights need to cast shadows.

  Only key lights need shadows — you can often get away without shadows on dimmer fill lights.

- Not all objects cast shadows or appear in reflections.

  As mentioned earlier with ray tracing, you should eliminate objects from shadow maps if they aren't expected to cast their shadows onto anything else. Common examples include floors, ceilings, etc. Similarly, objects that don't need to be seen in reflections should simply not be included when creating environment and reflection maps.

- Share maps among objects.

  Suppose you have two reflective objects in the scene. You could compute two separate cube-face environment maps, one for each object. But would the viewer really know if you just computed one environment map, and used it for several of the environment maps in your scene?

- Share maps among frames.

  If the objects in your shadow map aren't moving around, then you can compute the map once for the entire shot, and reuse it for all the frames. If only some objects are moving, consider putting the stationary object in one shadow map (computed only once) and the moving objects in a second map (computed every frame), using a light shader that understands how to combine both maps to compute shadows.

Similarly, if your reflective object is moving around but not much else is, you can probably compute the environment map once and use it for all the frames.

- Use generic maps if you can.

  Do you really need to see the scene clearly in your reflective object, or are you just putting a little bit of dim, blurry reflection to give the feeling that the object is shiny? If the latter, you may not notice if the reflection isn't really an image of the scene at all. Consider keeping a few "generic" environment maps around — one for outdoors, one for indoors, etc. — to throw on objects that need a little bit of reflection. Maybe you can avoid making any scene-specific environment maps entirely.

- Use cheap rendering options for shadow maps

  Shadow maps just record depth of objects in each pixel. Shadow maps don't record color, but they do need to know position and opacity. When doing a shadow map pass, be sure to turn off all lights, use a simple stand-in surface shader (like `constant.sl`) except for those shaders that modify `Oi` for opacity effects, and remove the displacement shader unless it is moving the surface by an appreciable amount.

  Also, for surfaces that are not displaced, consider using a larger number (coarser tessellation) for `ShadingRate`, for example `ShadingRate 16`. This will greatly speed up the rendering of the shadow map pass.

12.4 Tune the options

Make sure you are using the correct resolution for your image. Video resolution is `Format 640 480 1`, or if you're a real format geek, maybe `Format 720 486 0.9`. Film resolution is much less than many people suspect — theatrical films generally have CG elements with a resolution of at most 2048 horizontal pixels (and quite frequently much less). If you are rendering 3k or 4k images, you should be very careful that you really require such high resolution.

`PixelSamples` should be kept as low as possible without sacrificing quality. Due to Entropy's new antialiasing algorithm, `PixelSamples 4 4` is probably sufficient for most images, including those with a moderate amount of motion blur. In special circumstances, you may wish to increase `PixelSamples`, or even to set the spatial and temporal antialiasing levels separately (see Section 3.1.3).

Entropy uses `ShadingRate 1` by default. Some older articles and books claimed that `ShadingRate 0.25` should be used for "final images," but we believe that if your shaders properly antialias themselves, `ShadingRate 1` is the right size, even for high-quality work. Smaller values may occasionally be needed for special purposes, but generally speaking, if you are tempted to use a smaller `ShadingRate`, you may be compensating for a bad shader or for another renderer option being set incorrectly.

The bucket size (how much screen space is processed at once) is determined automatically by Entropy. But advanced users may wish to tune this parameter for difficult shots with `Option "limits" "bucketsize"`. The optimal value is somewhat scene-dependent. If you're rendering hundreds of frames of a difficult scene, it may be worth first

rendering a single frame with several different bucket size parameters — say, 8x8, 16x16, and 32x32. Then pick the one with the smallest rendering times and/or memory sizes, and use that value for the rest of the frames.

12.5 Use high-level geometric primitives

Entropy is excellent at taking high-level descriptions of curved geometric primitives and subdividing them just enough to draw them without artifacts. It's not very good at taking objects that are already diced into too many pieces.

Unlike many renderers, high-level primitives like NURBS are not inherently expensive. In fact, with Entropy, a NURBS patch has about the same expense as a *single* polygon that is the same size on screen. Thus, specifying curved primitives with `NuPatch` or `SubdivisionMesh` is considerably less expensive than specifying the same shape with thousands of tiny polygons, and additionally the curved surface will never show any polygonal artifacts.

If you are doing any ray tracing, there is additional savings to be had by using quadrics (like `Sphere` or `Cylinder`) rather than a `NuPatch` of the same shape.

Try to construct your area lights from simple geometry — `Sphere`, `Cylinder`, and bilinear patches are the most efficient. The `Sphere` and `Cylinder` primitives also are special-cased for more efficient area light sampling. A true `Sphere` or `Cylinder` as an area light will render much faster, and with much less noisy shadows, than an equivalently-shaped `NuPatch`.

12.6 Use procedural geometry

Entropy will internally store geometric primitives in one of three forms: (1) the original description of the primitive from the scene file, basically just the control vertices themselves; (2) higher-resolution, split, diced, and shaded version of the primitive; or (3) not stored at all.

Entropy will aggressively cull (throw away without storing) any primitives that are off-screen or that are occluded behind other opaque objects (assuming that the object cannot be ray traced). Entropy will try to keep the geometric description of the primitive as just the control vertices for as long as possible, converting to the higher-res version only when it's working on the part of the image that the object occupies, and will try to reclaim the memory as soon as it is done with the part of the screen where the object is seen.

This process can be further enhanced by using procedural geometry — that is, using a `Procedural` primitive as a stand-in for a large amount of geometry that is close together. Because an "unexpanded" procedural takes little more room than its bounding box, use of `Procedural` primitives can reduce the memory used for geometry to the bare minimum — even the control vertices don't need to be stored until Entropy is working on that part of the screen, at which time the `Procedural` is run and the real geometry is read in. Furthermore, if there are 1000 primitives inside a procedural, that procedural can be culled (if it's off screen or occluded) much more quickly than if all the primitives inside it would have to be examined and culled individually.

Aggressive use of `Procedural`'s, for example by storing all complex objects or spatially-coherent collections of objects in separate files referred to by `DelayedReadArchive`'s, can easily reduce the memory footprint of a complex scene by a factor of 4 or more. This could make the difference between being able to render a scene at all, or having it take more memory than would be available on your machine.

12.7 Don't be wasteful in your shaders

Entropy allows you to customize the appearance of objects in an extremely flexible manner, by providing a shading language in which the user can perform arbitrary computations. Of course, that freedom also allows shader to be arbitrarily expensive. It's not much of an exaggeration to say that shaders are executed in the *inner loop* of the renderer. Thus, a little attention to efficiency in shader programming can greatly affect overall rendering time.

Even a very expensive surface or displacement shader will only be run once per surface point. But an expensive light shader may be run many times — once for each light illuminating the object. Thus, it pays to be especially careful that your light shaders don't do any more work than is necessary. It also pays to use `Illuminate` liberally, to make sure that lights are "turned off" for any geometry that is outside the range of the light (too far away to be significantly illuminated, or in the wrong position to be illuminated).

Try to identify computations that do not take on different values for each point on your surface; those computations should be done using `uniform` variables. Declaring variables as `uniform` can save both time and storage during shader execution. If your shaders have `if`, `for`, or `while` statements, try to ensure that the conditional tests controlling the loops only involve `uniform` quantities and variables. Conditionals and loops that are controlled by `varying` conditions execute much more slowly.

Favor vector-oriented operations where possible. For example, suppose you have a color `C` and you want to multiply its green component by 0.5. Consider the following statement:

```
C = color (comp(C,0), comp(C,1)*0.5, comp(C,2));
```

This actually involves five instructions — three `comp()` calls, a multiply, and assigning a color from floats. But if you wrote it as follows:

```
C *= color (1, 0.5, 1);
```

That executes as a single multiply instruction (in-place multiplication of a color with a uniform color constant).

Be careful with the most expensive shader functions: `transform()` (really a matrix/vector multiply, or about 30 flops), `noise()` (the 3D version is about 80 flops), `texture()` (expensive filtering, plus it could cause a file read). We're not saying not to do these operations — you have to make your shaders look good. But be aware of the expensive operations and try not to use them if you cannot see the effect.

12.8 Use multi-threading

If you are rendering a single frame on a two-CPU machine, you should definitely be invoking `entropy` with the `-threads 2` option. It probably won't run exactly twice as fast, but it's often close, depending on the scene.

Part III

Developer’s Resources

Chapter 13

C API for Generating Scene Files

Chapter 2 described the formatting and commands for Entropy's input scene files. Typically, scene files will be generated by a modeling system or an appropriate plugin or converter. Most users will only have the need to read existing scene files, and occasionally modify them in minor ways. For developers implementing applications that create scene files, Entropy provides a C language API for writing scene files. The remainder of this chapter describes this API.

Since this section is intended for experienced developers, the API will be described in terms of ANSI C prototypes for the API functions, and some additional explanation as necessary. There is a nearly one-to-one correspondence between C API routines and scene file commands, so detailed explanations about the functionality of each routine is not necessary here.

13.1 Data Types

The following type definitions are used to compactly express what data are needed by API functions:

```
typedef short   RtBoolean;
typedef int     RtInt;
typedef float   RtFloat;
typedef char   *RtToken;
typedef RtFloat RtColor[3];
typedef RtFloat RtPoint[3];
typedef RtFloat RtVector[3];
typedef RtFloat RtNormal[3];
typedef RtFloat RtHpoint[3];
typedef RtFloat RtMatrix[4][4];
typedef RtFloat RtBasis[4][4];
typedef RtFloat RtBound[6];
typedef char   *RtString;
typedef char   *RtPointer;
#define RtVoid void
```

13.2 Parameter Lists and Declarations

Just as with scene files, many API routines, including geometric primitives and shader declarations, take a variable-length list of optional arguments in the form of *token-value pairs*. The C API has two means of passing token-value lists: as variable-length argument lists ("`varargs`"), and as separate arrays. Each routine that accepts token-value lists will have two versions, one for each method.

For the `varargs` version, a routine with a prototype such as:

```
void RiFoo (float a, ...);
```

will be passed alternating tokens (`const char *`, or `RtToken`) and pointers to array data, with the token/value list terminated by a `NULL`. For example, to call `RiFoo` with the parameter `a` = 1.5, and passing a point `"P"` and a string `"texturename"`:

```
RtPoint Pval;  /* Assume it gets a value somehow */
const char *name = "bar.tx";
RiFoo (1.5, "P", &Pval, "texturename", &name, NULL);
```

Alternately, there will always be a corresponding routine `RiFooV` with prototype:

```
RtVoid RiFooV (float a, int nargs, RtToken params[], RtPointer vals[]);
```

The fixed arguments will be identical, but instead of a varargs `...`, there will be three trailing arguments: an integer supplying the number of token-value pairs, and arrays of the parameter names and pointers to their values. As an example, an equivalent call of `RiFoo` corresponding to the example above would be:

```
RtPoint Pval;  /* Assume it gets a value somehow */
const char *name = "bar.tx";
RtToken toks[10];
RtPointer vals[10];
int nargs = 0;
toks[nargs] = "P";
vals[nargs] = &Pval;
++nargs;
toks[nargs] = "texturename";
vals[nargs] = &name;
++nargs;
RiFooV (1.5, nargs, toks, vals);
```

The ordinary varargs routines and their array equivalents are interchangeable. You may choose whichever one is more convenient for your application.

As with variable arguments in ASCII scene files, because these parameters represent user data, their types must be declared to the renderer prior to being used. This can be accomplished either by using "in-line declarations" by fully qualifying the data type in the parameter name (just as it was done in Section 2.3), or by using the `RiDeclare` function that exactly corresponds to the scene file `Declare` command:

`RtToken` **RiDeclare** (`RtToken` *name*, `RtToken` *declaration*

Adds a new parameter name and its type to the global name-type dictionary. The syntax for type declaration is identical to the scene file `Declare` command. The `RiDeclare` routine returns a new `RtToken` that is a *unique identifier* for the dictionary entry.

We recommend using "in-line declarations" rather than using `RiDeclare`, due mainly to the inconvenience and potential confusion arising from the single global dictionary. The `RiDeclare` syntax is mainly supported for backward-compatibility with older modeling programs.

13.3 Rendering Contexts and Block Structure

There may be multiple *rendering contexts* active, with at most being designated the *current rendering context*. All API routines (except those that start and stop contexts) apply only to the current context.

```
RtVoid RiBegin (RtToken name);
```

Begins a new rendering context and makes that new context the current context. The *name*, in the case where the context is outputting a scene file archive, is the name of the file. If the first character of *name* is the pipe symbol (`|`), the scene file output will be *piped* to a command or program given by the remainder of the *name*.

```
RtVoid RiEnd ();
```

Ends the current rendering context. After `RiEnd`, there is no current context.

```
RtContextHandle RiGetContext ();
void RiContext (RtContextHandle handle);
```

`RiGetContext` returns an opaque pointer specifying a unique handle for the current rendering context. `RiContext` switches the current context to the one specified by the *handle*.

```
RtVoid RiFrameBegin (RtInt number);
RtVoid RiFrameEnd (void);
RtVoid RiWorldBegin (void);
RtVoid RiWorldEnd (void);
RtVoid RiAttributeBegin (void);
RtVoid RiAttributeEnd (void);
RtVoid RiTransformBegin (void);
RtVoid RiTransformEnd (void);
```

These functions begin and end frames, the world block, and push/pop attributes or transformations. They all work analogously to the similarly named scene file commands described in Chapter 3.

```
RtVoid RiMotionBegin (int ntimes, ...);
RtVoid RiMotionBeginV (int ntimes, float *times);
RtVoid RiMotionEnd ();
```

Delimits a motion block. Between the `MotionBegin` and `MotionEnd`, there should be *ntimes* commands giving time-varying data at time values given by *times*. In the varargs version, the time values are successively pulled off the stack.

13.4 Options and Attributes

13.4.1 Options

```
RtVoid RiClipping (float hither, float yon);
RtVoid RiClippingPlane (float x, float y, float z, float nx, float ny, float nz);
RtVoid RiCropWindow (float xmin, float xmax, float ymin, float ymax);
RtVoid RiDepthOfField (float fstop, float focallength, float focaldistance);
RtVoid RiDisplay (char *name, RtToken type, RtToken mode, ...);
RtVoid RiDisplayV (char *name, RtToken type, RtToken mode,
                           int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiExposure (float gain, float gamma);
RtVoid RiFormat (int xres, int yres, float aspect);
RtVoid RiFrameAspectRatio (float aspect);
RtVoid RiHider (RtToken type, ...);
RtVoid RiHiderV (RtToken type, int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiImager (char *name, ...);
RtVoid RiImagerV (char *name, int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiOption (char *name, ...);
RtVoid RiOptionV (char *name, int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiPixelSamples (float xsamples, float ysamples);
RtVoid RiProjection (char *name, ...);
RtVoid RiProjectionV (char *name, int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiQuantize (RtToken type, int one, int qmin, int qmax, float ditheramp);
RtVoid RiRelativeDetail (float relativedetail);
RtVoid RiScreenWindow (float left, float right, float bot, float top);
RtVoid RiShutter (float opentime, float closetime);
```

These routines set renderer options. Their semantics are identical to the analogous scene file commands described in Section 3.1.

```
RtVoid RiPixelFilter (RtFilterFunc function, float xwidth, float ywidth);
```

Sets the pixel filter function. The following predefined pixel filters may be used as the *function*:

```
float RiBoxFilter (float x, float y, float xwidth, float ywidth);
float RiCatmullRomFilter (float x, float y, float xwidth, float ywidth);
float RiGaussianFilter (float x, float y, float xwidth, float ywidth);
float RiSincFilter (float x, float y, float xwidth, float ywidth);
float RiTriangleFilter (float x, float y, float xwidth, float ywidth);
```

13.4.2 Attributes

```
RtVoid RiAttribute (char *name, ...);
RtVoid RiAttributeV (char *name, int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiColor (RtColor Cs);
RtVoid RiDetail (RtBound bound);
RtVoid RiDetailRange (float minvis, float lowtran, float uptran, float maxvis);
RtVoid RiGeometricApproximation (RtToken type, float value);
RtVoid RiMatte (RtBoolean onoff);
RtVoid RiOpacity (RtColor Cs);
RtVoid RiOrientation (RtToken orientation);
RtVoid RiReverseOrientation (void);
RtVoid RiShadingInterpolation (RtToken type);
RtVoid RiShadingRate (float size);
RtVoid RiSides (int nsides);
RtVoid RiTextureCoordinates (float s1, float t1, float s2, float t2,
                             float s3, float t3, float s4, float t4);
```

These routines set renderer attributes. Their semantics are identical to the analogous scene file commands described in Section 3.2.

13.4.3 Transformations

```
RtVoid RiConcatTransform (RtMatrix transform);
RtVoid RiCoordinateSystem (RtToken space);
RtVoid RiCoordSysTransform (RtToken space);
RtVoid RiIdentity (void);
RtVoid RiPerspective (float fov);
RtVoid RiRotate (float angle, float dx, float dy, float dz);
RtVoid RiScale (float dx, float dy, float dz);
RtVoid RiSkew (float angle, float dx1, float dy1, float dz1,
                  float dx2, float dy2, float dz2);
RtVoid RiTransform (RtMatrix transform);
RtVoid RiTranslate (float dx, float dy, float dz);
```

These routines alter the current transformation analogously to those described in Section 3.2.3.

13.4.4 Shaders and Lights

```
RtVoid RiAtmosphere (char *name, ...);
RtVoid RiAtmosphereV (char *name, int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiDisplacement (char *name, ...);
RtVoid RiDisplacementV (char *name, int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiExterior (char *name, ...);
```

```
RtVoid RiExteriorV (char *name, int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiInterior (char *name, ...);
RtVoid RiInteriorV (char *name, int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiSurface (char *name, ...);
RtVoid RiSurfaceV (char *name, int nparams, RtToken *tokens, RtPointer *values);
```

Sets the shaders analogously to the scene file commands described in Section 3.2.4.

```
RtLightHandle RiAreaLightSource (char *name, ...);
RtLightHandle RiAreaLightSourceV (char *name,
                            int nparams, RtToken *tokens, RtPointer *values);
RtLightHandle RiLightSource (char *name, ...);
RtLightHandle RiLightSourceV (char *name,
                            int nparams, RtToken *tokens, RtPointer *values);
```

Creates a light or area light source. These routines return an opaque pointer called an `RtLightHandle` that can be passed to `RiIlluminate`. Note that this is different than the scene file command, which, due to the constraints of one-way communication, must pass a handle number to the light source declaration.

```
RtVoid RiIlluminate (RtLightHandle light, RtBoolean onoff);
```

Turn a light source on or off. The light is specified by an `RtLightHandle` that was returned when the light or area light was declared.

13.5 Geometric Primitives

13.5.1 Polygons and Polygon Meshes

```
RtVoid RiPolygon (int nverts, ...);
RtVoid RiPolygonV (int nverts,
                            int nparams, RtToken *tokens, RtPointer *values);
```

Create a single convex polygon with *nvertices* vertices. The token-value list must contain position data (`"P"`) and may optionally contain other primitive variables.

```
RtVoid RiGeneralPolygon (int nloops, int *nverts, ...);
RtVoid RiGeneralPolygonV (int nloops, int *nverts,
                            int nparams, RtToken *tokens, RtPointer *values);
```

Create a single, possibly concave, polygon with *nloops* loops. The *nverts* array, whose length is *nloops*, contains the number of vertices in each loop. The list of token-value pairs must contain position data (`"P"`) and may optionally contain other primitive variables. The length of the vertex point list should be the sum of the numbers of vertices in all loops.

```
RtVoid RiPointsPolygons (int npolys, int *nverts, int *verts, ...);
RtVoid RiPointsPolygonsV (int npolys, int *nverts, int *verts,
                          int nparams, RtToken *tokens, RtPointer *values);
```

Create *npolys* convex polygons with shared vertex data. The data in *nverts* is the number of vertices in each face. The array *verts*, whose length should be the sum of all entries in *nverts*, contains the vertex indices for each face, in order. The token-value list must contain position data (`"P"`) and may optionally contain other primitive variables. The length of the vertex point list must be at least as long as the highest vertex index in *verts* plus one.

```
RtVoid RiPointsGeneralPolygons (int npolys, int *nloops,
                          int *nverts, int *verts, ...);
RtVoid RiPointsGeneralPolygonsV (int npolys, int *nloops,
                          int *nverts, int *verts,
                          int nparams, RtToken *tokens, RtPointer *values);
```

Create *npolys* general polygons with shared vertex data. The array *nloops* contains the number of loops in each polygon. The array *nverts*, whose length must be the total number of loops in all polygons (i.e., the sum of all entries in *nloops*), contains the number of vertices in each loop (in polygon order). The array *verts*, whose length should be the sum of all entries in *nverts*, contains the vertex indices for each loop, for each face, in order. The token-value list must contain position data (`"P"`) and may optionally contain other primitive variables. The length of the vertex point list must be at least as long as the highest vertex index in *verts* plus one.

13.5.2 Control-Point Primitives

```
RtVoid RiBasis (RtBasis ubasis, int ustep, RtBasis vbasis, int vstep);
```

This routine sets the u and v basis matrices for cubic `Patch`, `PatchMesh`, and `Curves` primitives, and the u and v step sizes for cubic `PatchMesh` and `Curves` primitives. The basis matrices and steps are ordinary attributes and may be saved and restored with `AttributeBegin` and `AttributeEnd`.

The basis matrices are either specified by name (one of `RiBezierBasis`, `RiBSplineBasis`, `RiCatmullRomBasis`, or `RiHermiteBasis`) as a 4×4 matrix of floats. The step sizes are integers and should be 3 for Bezier, 1 for B-spline or Catmull-Rom, or 2 for Hermite.

```
RtVoid RiPatch (RtToken type, ...);
RtVoid RiPatchV (RtToken type, int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiPatchMesh (RtToken type, int nu, RtToken uwrap,
                          int nv, RtToken vwrap, ...);
RtVoid RiPatchMeshV (RtToken type, int nu, RtToken uwrap, int nv, RtToken vwrap,
                          int nparams, RtToken *tokens, RtPointer *values);
```

Creates a single patch or a rectangular mesh of patches. The *type* should be `"bilinear"` or `"bicubic"`. In the case of a bicubic patch or patch mesh, the bicubic basis and the mesh step size will be the ones specified in the attribute state by the `Basis` function. The *uwrap* and *vwrap* may be either `"nonperiodic"` or `"periodic"`, indicating whether or not the u and v directions wrap all the way around to form a continuous ring.

```
RtVoid RiNuPatch (int nu, int uorder, float *uknot, float umin, float umax,
                  int nv, int vorder, float *vknot, float vmin, float vmax,
                  ...);
RtVoid RiNuPatchV (int nu, int uorder, float *uknot, float umin, float umax,
                  int nv, int vorder, float *vknot, float vmin, float vmax,
                  int nparams, RtToken *tokens, RtPointer *values);
```

Create a NURBS (non-uniform rational B-spline) mesh. The meanings of the parameters are identical to the scene file `NuPatch` command.

```
RtVoid RiTrimCurve (int nloops, int *ncurves, int *order,
                  float *knot, float *amin, float *amax,
                  int *n, float *u, float *v, float *w);
```

Sets the trim curve, analogously to the scene file `NuPatch` command. Note that the trim curve is an attribute, and may be saved and restored along with the rest of the attribute state.

```
RtVoid RiSubdivisionMesh (RtToken scheme, int nfaces, int *nvertices,
                          int *vertices, int ntags, RtToken *tags,
                          int *nargs, int *intargs, float *floatargs, ...);
RtVoid RiSubdivisionMeshV (RtToken scheme, int nfaces, int *nvertices,
                          int *vertices, int ntags, RtToken *tags,
                          int *nargs, int *intargs, float *floatargs,
                          int nparams, RtToken *tokens, RtPointer *values);
```

Create a subdivision surface mesh. The *scheme* specifies the name of the subdivision method (currently only `"catmull-clark"` is recognized). Much like `RiPointsPolygons`, the array *nverts* (of length *nfaces*) contains the number of vertices in each face. The array *vertices*, whose length should be the sum of all entries in *nverts*, contains the vertex indices for each face, in order. The token-value list must contain position data (`"P"`) and may optionally contain other primitive variables. The length of the vertex point list (`"P"`) must be at least as long as the highest vertex index in *verts* plus one.

Faces, edges, and vertices may be tagged with additional properties. The *tags* array contains the tag names. The array *nargs* has length $ntags \times 2$, and for each tag contains the number of integer arguments, followed by the number of floating-point arguments, for that tag. The arrays *intargs* and *floatargs* contain all of the integer and float arguments, respectively (the length of the *intargs* should be the sum of all the even elements of *nargs*, and the length of the *floatargs* should be the sum of all

the odd elements of *nargs*). The tags and their expected arguments are explained in Section 4.4.

```
RtVoid RiCurves (RtToken degree, int ncurves, int *nverts, RtToken wrap, ...);
RtVoid RiCurvesV (RtToken degree, int ncurves, int *nverts, RtToken wrap,
                          int nparams, RtToken *tokens, RtPointer *values);
```

Draws *ncurves* curve primitives. The *type* may be either `"linear"` or `"cubic"`. Piecewise cubic curves use the v basis matrix set by `Basis`. The array *nvertices* has length equal to the *ncurves*, and its data are the number of vertices in each curve. The string *wrap* is either `"periodic"` or `"nonperiodic"`, describing whether or not the individual curves wrap end-to-end. The token-value list must contain position data (`"P"`) and may also contain other primitive variables. The total number of control points `"P"` must be the total vertices in all the curves (i.e., the sum of all entries in the *nvertices* array).

```
RtVoid RiPoints (int npts, ...);
RtVoid RiPointsV (int npts, int nparams, RtToken *tokens, RtPointer *values);
```

Draws *npts* point-like particles. The token-value list must contain position data (`"P"`) and may also contain other primitive variables.

13.5.3 Quadrics

```
RtVoid RiCone (float height, float radius, float thetamax, ...);
RtVoid RiConeV (float height, float radius, float thetamax,
                          int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiCylinder (float radius, float zmin, float zmax, float thetamax, ...);
RtVoid RiCylinderV (float radius, float zmin, float zmax, float thetamax,
                          int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiDisk (float height, float radius, float thetamax, ...);
RtVoid RiDiskV (float height, float radius, float thetamax,
                          int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiHyperboloid (RtPoint point1, RtPoint point2, float thetamax, ...);
RtVoid RiHyperboloidV (RtPoint point1, RtPoint point2, float thetamax,
                          int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiParaboloid (float rmax, float zmin, float zmax, float thetamax, ...);
RtVoid RiParaboloidV (float rmax, float zmin, float zmax, float thetamax,
                          int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiSphere (float radius, float zmin, float zmax, float thetamax, ...);
RtVoid RiSphereV (float radius, float zmin, float zmax, float thetamax,
                          int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiTorus (float majorrad, float minorrad,
                  float phimin, float phimax, float thetamax, ...);
```

```
RtVoid RiTorusV (float majorrad, float minorrad,
                 float phimin, float phimax, float thetamax,
                          int nparams, RtToken *tokens, RtPointer *values);
```

Create the desired quadric primitive. Quadric parameters are explained in Section 4.6.

13.5.4 Implicit Surfaces

```
RtVoid RiBlobby (int nleaf, int ncode, int code[],
                 int nflt, float flt[], int nstr, RtString str[], ...);
RtVoid RiBlobbyV (int nleaf, int ncode, int code[],
                 int nflt, float flt[], int nstr, RtString str[],
                 int n, RtToken tokens[], RtPointer params[]);
```

13.5.5 Procedural Geometry

```
RtVoid RiProcedural (RtPointer data, RtBound bound,
                 RtProcSubdivFunc subdivfunc, RtProcFreeFunc freefunc);
```

Adds a procedural primitive to the scene. The procedural primitive is enclosed in the *bound*, and carries around information pointed to by the opaque pointer *data*. When the contents of the *bound* are needed, the renderer will call the function *subdivfunc*. When the renderer is done with the procedural primitive, it will call the *freefunc*. These functions have types and arguments defined by:

```
typedef RtVoid (*RtProcSubdivFunc)(RtPointer data, float detail);
typedef RtVoid (*RtProcFreeFunc)(RtPointer data);
```

The *data* parameter passed to the subdivide and free functions is the same blind *data* pointer that was specified to `RiProcedural`. The *detail* is a level-of-detail parameter that contains the size of the bounding box on screen (expressed in square pixels).

Three useful built-in subdivide functions exist which may be specified by name (as opposed to the user supplying her own function):

```
RtVoid RiProcDelayedReadArchive (RtPointer data, float detail);
RtVoid RiProcRunProgram (RtPointer data, float detail);
RtVoid RiProcDynamicLoad (RtPointer data, float detail);
```

These built-in functions correspond to the similarly-named functions described in Section 4.8. There is also one built-in free function:

```
RtVoid RiProcFree (RtPointer data);
```

The `RiProcFree` function simply calls `free(data)`.

13.6 External Resources

```
void RiArchiveRecord (RtToken type, char *format, ...);
```

This function writes raw data into the output stream with a *format* and optional arguments, just like the C `printf` function. If *type* is `"comment"`, the output is prepended by the comment symbol (`#`) and terminated by a linefeed. If *type* is `"structure"`, the output is prepended by two comment symbols (`##`) and terminated by a linefeed. If *type* is `"verbatim"`, the output is emitted into the scene file without any modification and is not terminated by a newline.

```
void RiReadArchive (RtString filename, RtArchiveCallback callback, ...);
void RiReadArchiveV (RtString filename, RtArchiveCallback callback,
                     int nparams, RtToken *tokens, RtPointer *values);
```

Parse and execute commands from an archive file in *filename*. If archive structure records are encountered during parsing, the *callback* routine will be executed. The function prototype for the callback routine is:

```
typedef RtVoid (*RtArchiveCallback)(RtToken, char *, ...);
```

If you do not need to intercept comments or structures, you can just supply `NULL` for the callback routine and for the optional arguments.

```
RtVoid RiMakeCubeFaceEnvironment (char *px, char *nx, char *py, char *ny,
                     char *pz, char *nz, char *tex, float fov,
                     RtFilterFunc filterfunc,
                     float swidth, float twidth, ...);
RtVoid RiMakeCubeFaceEnvironmentV (char *px, char *nx, char *py, char *ny,
                     char *pz, char *nz, char *tex, float fov,
                     RtFilterFunc filterfunc, float swidth, float twidth,
                     int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiMakeLatLongEnvironment (char *pic, char *tex, RtFilterFunc filterfunc,
                          float swidth, float twidth, ...);
RtVoid RiMakeLatLongEnvironmentV (char *pic, char *tex, RtFilterFunc filterfunc,
                     float swidth, float twidth,
                     int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiMakeShadow (char *pic, char *tex, ...);
RtVoid RiMakeShadowV (char *pic, char *tex,
                     int nparams, RtToken *tokens, RtPointer *values);
RtVoid RiMakeTexture (char *pic, char *tex, RtToken swrap, RtToken twrap,
                     RtFilterFunc filterfunc, float swidth, float twidth,
                     ...);
RtVoid RiMakeTextureV (char *pic, char *tex, RtToken swrap, RtToken twrap,
                     RtFilterFunc filterfunc, float swidth, float twidth,
                     int nparams, RtToken *tokens, RtPointer *values);
```

These routines convert ordinary image files into texture, environment, and shadow maps that can be accessed efficiently. They are equivalent to the routines explained in Section 2.5.2.

13.7 Using `ri.h` and `libribout`

The ANSI C type definitions and prototypes for the API described in this chapter can be found in the file `ri.h`, which comes in the `include` directory of the Entropy distribution. Therefore, any program that wishes to use the functions must:

```
#include "ri.h"
```

The library `libribout` implements the C API in such a way that executing the API calls will cause an ASCII scene file to be output to the file designated as the argument to `RiBegin`.

Chapter 14

Calling C functions from Shaders

Although the shading language has both an extensive collection of built-in library functions (see Section 5.6) and syntax for defining and calling user-written functions, there are some things that are impossible to do in ordinary shaders. Examples include (non-texture) file access, building of complex data structures, and access to certain OS functionality. For this reason, there is a mechanism by which a functions written in C and compiled into DSO's/DLL's[1] can be called from shaders. These routines are sometimes called "DSO Shadeops."

14.1 Theory of Operation

To support polymorphism[2] in the shading language, a *dispatch table* must be in the DSO. Each polymorphic version of the function in the dispatch table actually has three associated routines:

- an *initializer*, which is called before the first time the DSO shadeop is executed;
- the *implementation*, which is the actual C function called to do the operation that the DSO Shadeop is designed to perform;
- a *cleanup* routine, which is called after the DSO Shadeop is no longer needed.

The developer compiles these routines into a DSO. When compiling the shader, the function may be called in the same manner as any built-in or user-defined shader function. When the shader compiler, `sle`, encounters a call for a function that is not defined, it will search all directories specified by the `-I` switch, looking for DSO's. Any DSO's that are found will be searched for the appropriately-named dispatch table (see below), and if found will understand that the call is to a DSO Shadeop.

For rendering, the DSO must be compiled and placed in one of the directories that contain compiled shaders (see the search path options described in Section 3.1.3). The DSO will be loaded only as needed at render time.

[1] For brevity, we will refer to these as "DSO's." Windows users, please understand that this is perfectly synonymous with "DLL."

[2] *Polymorphism* is the ability for one named function to have several versions, each with different sets of parameters.

14.2 Calling Conventions

The dispatch table is an array of DS_DispatchTableEntry records, which is defined in "dsoshadeop.h" as:

```
typedef struct {
    const char *implementation_prototype;
    const char *initializer;
    const char *cleanup_routine;
} DS_DispatchTableEntry;
```

The fields of the dispatch entry are character strings containing the names of the routines to use. All routines must have implementation prototypes, but the initializer and cleanup routines are optional; having no routine is signified with an empty string (""). The end of the table is signified by a dispatch entry whose implementation routine is given by an empty string ("").

The table must be named *op*_shadeops, where *op* is the name of the function we are implementing. For example, to implement a function called myabs, the dispatch table might look like this:

```
EXPORT DS_DispatchTableEntry myabs_shadeops[] = {
     { "float myabs_f(float)", "myabs_init", "myabs_cleanup" },
     { "point myabs_p(point)", "", "" },
     { "vector myabs_v(vector)", "", "" },
     { "", "", "" }
};
```

The initializer and cleanup fields just supply the names of their functions. However, the implementation contains a complete function prototype showing the argument types and return value type, but using the name of the *C implementation*, rather than the shading language routine name.

Note that the EXPORT symbol is defined in dsoshadeop.h and correctly handles OS-specific declarations of which routines must be visible to the renderer.

Initialization routines, which are optional, have the following prototype:

```
EXPORT void * init (int, void *);
```

The initialization function returns a void *. If the initializer allocates memory or creates data structures, it should return a blind pointer to this memory. The prototype takes an int and a void *, but these parameters are not currently used.

Implementation routines have the following prototype:

```
EXPORT int implement (void *data, int nargs, void **args);
```

The data is the same blind pointer returned by the initializer routine, or NULL if there was no initializer. The nargs and args are the number, and array of pointers to, the actual arguments to the shadeop. The args[0] is the pointer to where the function result should be stored. For void functions, args[0] is unused. In either case, the actual arguments passed to the function are args[1] through args[nargs].

Finally, the optional cleanup routines have the prototype:

```
EXPORT void cleanup (void *data);
```

The cleanup routine is passed the same `data` pointer that was created by the initializer and was passed to the implementation. If this points to any data structures or allocated memory, the cleanup routine is responsible for freeing up the memory.

14.3 Example

As an illustrative example, below is the full source code for the implementation of an absolute value function, `myabs`. The `myabs` function is just like the built-in `abs()` for `float` values, but also works on a per-component for vectors and points. The source code below would ordinarily be stored in `myabs.c` and compiled into the file `myabs.so` (on Unix or Linux) or `myabs.dll` (on Windows).

```
#include <math.h>
#include <stdio.h>
#include "dsoshadeop.h"

#ifdef __cplusplus
extern "C" {
#endif

EXPORT DS_DispatchTableEntry myabs_shadeops[] = {
     { "float myabs_f(float)", "myabs_init", "myabs_cleanup" },
     { "point myabs_p(point)", "", "" },
     { "vector myabs_p(vector)", "", "" },
     { "", "", "" }
};


EXPORT int myabs_f (void *data, int nargs, void **args)
{
    float *result = (float *) args[0];
    float *x = (float *) args[1];
    *result = fabs (*x);
    return 0;
}


EXPORT int myabs_p (void *data, int nargs, void **args)
{
    float *result = (float *) args[0];
    float *x = (float *) args[1];
    for (int i = 0;  i < 3;  ++i, ++result, ++x)
        *result = fabs (*x);
    return 0;
}
```

```
EXPORT void *myabs_init (int, void *)
{
    /* myabs() doesn't really need an initializer, but if it did, here
     * is where it would go.
     */
    return NULL;
}


EXPORT void myabs_cleanup (void *data)
{
    /* myabs() doesn't really need an cleanup, but if it did, here
     * is where it would go.
     */
}

#ifdef __cplusplus
}  /* extern "C" */
#endif
```

Below is an example shader that calls the `myabs()` function. The `myabs()` function is called just like any built-in or user-defined function; no special declaration is necessary. Also note that polymorphism is correctly resolved based upon the arguments to the function.

```
surface testmyabs ()
{
    float a = myabs(s-t);   /* Take absolute value of the float (s-t) */
    point pa = myabs(P);    /* Make the components of P positive */
    printf ("s-t = %f, myabs(s-t) = %f\n", s-t, a);
    printf ("P = %p, myabs(P) = %p\n", P, pa);
}
```

The source code for `myabs.c` and `testmyabs.sl` can be found in the Entropy distribution under `examples/src/dsoshadeop`.

14.4 Tips

There are a few things to remember to make your DSO Shadeop experience more pleasant:

- You may write your DSO Shadeops in C++, but the exported routines and the dispatch table must have "C linkage," which is why the `extern "C" {...}` is used in the example above.
- On Windows, the `EXPORT` macro (which actually expands to `__declspec(dllexport)`) is required, or the renderer will not be able to correctly reference the DSO routines.
- It is okay to put multiple DSO shadeops in the same DSO. Just be careful that each shadeop has its own, properly named, dispatch table. It is also okay for multiple (presumably cooperating) shadeops to share initialization and cleanup routines. Entropy

is careful to only call the initialize and cleanup routines once, even if it's used by multiple shadeops.

- Even though ordinary shaders are interpreted, they are usually very fast. Even though DSO shadeops are compiled, there is significant overhead involved in making calls to them. Therefore, a function that could be performed in the shading language will almost never be sped up significantly by implementing it as a DSO shadeop. The DSO shadeop mechanism does not exist for speed.

  Rather, DSO shadeops are best used to add functionality that could not be performed at all in ordinary shaders. Examples include: file I/O (other than reading texture files), building complex data structures such as large tables or spatial search trees, or accessing OS functionality such as pipes or resource management.

Chapter 15

Interrogating Shader Arguments with `libsleargs`

Some Entropy users who are software developers may need to be able to understand the parameters of compiled shaders. Entropy comes with a library, `libsleargs`, that allows C++ applications to parse compiled shaders, reporting the name, type, storage class, and default values of each shader parameter. If you are not developing such applications, this chapter will not contain any useful information.

15.1 The `sleArgs` class

The header file `sleargs.h` provides a definition for the C++ class `sleArgs`. The `sleArgs` class defines the following public member functions:

`sleArgs::`**sleArgs** `(const char *`*shadername*`, const char *`*shaderpath*`=NULL)`

> The constructor of `sleArgs` takes two arguments: the *shadername* is the name of the shader to read. The optional *shaderpath* is a colon-separated (or semicolon-separated) list of directories in which to search for the named shader.

`Type sleArgs::`**shadertype** `( )`

> Returns the type of the shader as one of the enumerated type `sleArgs::Type` (one of `TYPE_SURFACE`, `TYPE_DISPLACEMENT`, `TYPE_LIGHT`, `TYPE_VOLUME`, `TYPE_IMAGER`).

`const char *sleArgs::`**shadername** `( )`

> Returns the name of the shader.

`int sleArgs::`**nargs** `( )`

> Returns the number of shader parameters that the shader accepts.

`const Symbol * sleArgs::`**getarg** `(const char *`*name*`)`
`const Symbol * sleArgs::`**getarg** `(int` *i*`)`

Return a pointer to a `sleArgs::Symbol` record for one particular shader parameter. The first form looks up the symbol by *name*, returning `NULL` if there is no parameter by that name. The second form looks up the parameter by number (indexed beginning with 0).

`const Symbol *sleArgs::`**getarrayelement** `(const Symbol *`*array*`, int` *index*`)`

Given the pointer to a parameter symbol for an array parameter, *array*, and a particular *index*, return a pointer to a `sleArgs::Symbol *` for the particular element index.

`static const char * sleArgs::`**typestr** `(Type` *t*`)`

This helper function takes an `sleArgs::Type` and returns the string for that data type. For example, `typestr(sleArgs::TYPE_FLOAT)` returns `"float"`.

Several of these member functions use two types that are defined locally to the `sleArgs` class, `sleArgs::Type` and `sleArgs::Symbol`. There is probably no clearer explanation of these types than simply listing their definitions:

```
enum Type {
    TYPE_ERROR = -1, TYPE_UNKNOWN = 0,
    TYPE_FLOAT = 1, TYPE_COLOR, TYPE_POINT, TYPE_VECTOR, TYPE_NORMAL,
    TYPE_MATRIX, TYPE_STRING,
    TYPE_SURFACE = 16, TYPE_DISPLACEMENT, TYPE_LIGHT,
    TYPE_VOLUME, TYPE_IMAGER
};

struct Symbol {
    const char *name;         // argument name
    Type type;                // data type
    int arraylen;             // array length, or 0 if not an array
    bool output, varying;     // is it an output param?  is it varying?
    const char *spacename;    // name of space, or NULL if not applicable

    Symbol ();
    bool valisvalid ();
    float floatval (int i) const;
    const char *stringval (int i) const;
};
```

15.2 Using `sleargs.h` and `libsleargs`

Following are the basic steps to following in order to parse the arguments of a shader.

1. Your program should be sure to include the file `sleargs.h`:

```
#include "sleargs.h"
```

2. Begin parsing shader arguments by creating an object of type `sleArgs`. The `sleArgs` constructor requires the name of the file, and optionally the search path.

```
char *shadername;   /* = "plastic" or whatever */
sleArgs argparser (shadername);
```

3. You can check the name of the shader and its type as follows:

```
const char *sname = argparser.shadername();
sleArgs::Type stype = argparser.shadertype();
```

   The `shadertype()` method returns type `sleArgs::Type`, which can be one of `TYPE_SURFACE`, `TYPE_DISPLACEMENT`, `TYPE_LIGHT`, `TYPE_VOLUME`, `TYPE_IMAGER` (all defined locally to class `sleArgs`).

   If the shader was not found or could not be correctly parsed, `shadername()` will return `NULL` and `shadertype()` will return `sleArgs::TYPE_ERROR`.

4. You can determine the total number of shader parameters using the `nargs()` method:

```
int n = argparser.nargs();
```

   For each argument, you can retrieve information about it using the `getarg()` methods, which returns a pointer to a `sleArgs::Symbol`. You can access by name,

```
const sleArgs::Symbol *sym;
sym = argparser.getarg ("Kd");
```

   or by argument number (between 0 and `nargs()-1`),

```
sym = argparser.getarg (3);
```

   The `sleArgs::Symbol` structure contains information about that parameter: its name, its type (also one of `sleArgs::Type`, such as `TYPE_FLOAT` or `TYPE_VECTOR`), whether or not it's `varying`, whether or not it's an `output` parameter, its array length (0 if it's not an array), a pointer to its default value, and (where applicable) the name of the space of the default value (e.g., `"shader"`).

5. The default values of the symbol can be accessed through `Symbol::floatval()` or `Symbol::stringval()`:

```
if (sym->type == sleArgs::TYPE_COLOR)
    printf ("color = %g %g %g\n", sym->floatval(0),
            sym->floatval(1), sym->floatval(2));
else if (sym->type == sleArgs::TYPE_STRING)
    printf ("string = '%s'\n", sym->stringval(0));
```

6. If the symbol is an array (`sym->arraylen > 0`), you can access its elements individually as symbols using the `getarrayelement()` method:

```
const sleArgs::Symbol *elemsym;
elemsym = argparser.getarrayelement (sym, index);
```

7. When the `sleArgs` object (`argparser` in our example above) is destroyed or exists the scope, it will free any resources that it had allocated. There is no cleanup that the user is required to do.

15.3 Example: `sletell` source code

As an illustrative example of the use of `libsleargs`, we present the full source code of the `sletell` application.

Listing 15.1: `sletell.c` source code

```
////////////////////////////////////////////////////////////////////////
// sletell.c -- read and report the parameters to a compiled shader (.sle)
//
// Entropy / BMRT are:
// (c) Copyright 2001 Exluna, Inc.  All rights reserved.
//
// $Revision: 1.3 $   $Date: 2001/08/20 17:01:58 $
//
////////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

#include "release.h"
#include "sleargs.h"


static void
printoptions (void)
{
    cerr << SHADER_TELL " - Give info on compiled shader file (."
        SHADER_EXTENSION ")\n"
        "    (c) Copyright 2001 Exluna, Inc.  All Rights Reserved.\n"
        "Usage: " SHADER_TELL " <shadername>\n"
        "\n";
    cerr.flush();
}



static void
print_default (const sleArgs::Symbol *v)
{
    switch (v->type) {
    case sleArgs::TYPE_FLOAT :
        cout << v->floatval(0) << '\n';
        break;
    case sleArgs::TYPE_COLOR :
        cout << "\"" << (v->spacename ? v->spacename : "rgb")
             << "\" [" << v->floatval(0) << ' ' << v->floatval(1)
             << ' ' << v->floatval(2) << "]\n";
```

```
        break;
    case sleArgs::TYPE_POINT :
    case sleArgs::TYPE_VECTOR :
    case sleArgs::TYPE_NORMAL :
        cout << "\"" << (v->spacename ? v->spacename : "current")
             << "\" [" << v->floatval(0) << ' ' << v->floatval(1)
             << ' ' << v->floatval(2) << "]\n";
        break;
    case sleArgs::TYPE_STRING :
        cout << "\"" << v->stringval(0) << "\"\n";
        break;
    case sleArgs::TYPE_MATRIX :
        cout << "[ ";
        for (int m = 0;  m < 16;  ++m)
            cout << v->floatval(m) << ' ';
        cout << "]\n";
        break;
    }
}



static void
sletell (const char *sname)
{
    char shadername[256];
    strcpy (shadername, sname);
    // If the bozo typed in "foo.sle" instead of "foo", chop off the ".sle"
    int len = shadername ? strlen(shadername) : 0;
    if (len > 4 &&  !strcmp(shadername+len-4, "." SHADER_EXTENSION))
        shadername[len-4] = 0;

    sleArgs args(shadername);
    sleArgs::Type shadertype = args.shadertype();
    if (shadertype == sleArgs::TYPE_ERROR) {
        cerr << SHADER_TELL ": Could not find shader \"" << shadername
             << "\"\n";
        return;
    }
    cout << sleArgs::typestr(shadertype) << " \"" << shadername << "\"\n";

    for (int i = 0;  i < args.nargs();  ++i) {
        const sleArgs::Symbol *v = args.getarg(i);
        cout << "    \"" << v->name << "\" \""
             << (v->output ? "output " : "")
             << (v->varying ? "varying " : "uniform ")
             << sleArgs::typestr(v->type);
        if (v->arraylen > 0) {
            cout << "[" << v->arraylen << "]";
            cout << "\"\n\t\tDefault values:\n";
            for (int a = 0;  a < v->arraylen;  ++a) {
                cout << "\t\t\t[" << a << "] ";
                print_default (args.getarrayelement(v, a));
            }
            continue;
        }
        if (! v->valisvalid()) {
            cout << "\"\n";
            continue;
        }
        cout << "\"\n\t\tDefault value: ";
        print_default (v);
    }
```

```
    cout << '\n';
    cout.flush();
}



int
main (int argc, char **argv)
{
    if (argc <= 1)
        printoptions();
    for (int i = 1;  i < argc;  ++i) {
        if (! strcmp (argv[i], "-") ||  ! strcmp (argv[i], "-help") ||
            ! strcmp (argv[i], "-h"))
        {
            printoptions ( );
            break;
        }
        sletell (argv[i]);
    }
    return 0;
}
```

Chapter 16

Writing Custom Display Servers

Entropy ships with the ability to write images in a few standard formats: TIFF files for ordinary images, a simple format for depth buffer images, etc. But Entropy is by no means restricted to those formats that its developers prefer. Programmers can write C++ plug-ins called "display servers" (which exist as DSO's or DLL's) that capture Entropy's raw pixel output, and do whatever they want with that data (including, but certainly not limited to, saving them to a custom file format).

Users and developers are encouraged to write these display servers, and even to distribute them for free or as commercial "third party extensions."

16.1 Theory of Operation

The renderer's primary output consists of floating-point pixel data. Pixels are sent to one or more "displays streams" each of which will receive one or more "channels" of data. For example, four channels representing red, green, blue, and alpha may be sent to one display stream. A second display stream might have a single channel of depth (z) data.

The display streams are sent to "display servers," which can be quite flexible. A display server might write the data to a file in a particular format, or might display the data on a CRT. A display server might even further process the data before handing it to another process. We call them "servers" because they are receiving requests sent by a "client" (the renderer).

Although the raw pixel data is all in floating-point format, the display server may need to quantize the data to integer values in order to write it to a particular file or device. Some file formats or devices may require particular formats (for example JPEG files must be 8 bits per channel). Other formats or devices may be able to handle a variety of data formats, but the user may have requested a particular format for the data. This information, among other things, is communicated to the display server through a set of parameters, some provided automatically by the renderer and others optionally given to the `Display` command.

All Entropy display servers are subclass of the `ExDisplay` class. To create a custom display server, a developer should subclass `ExDisplay`, overriding the pure virtual functions. The resulting class methods can be put into a DSO (DLL on Windows systems), and this will be loaded at runtime by the renderer. The DSO itself has two public

entry point symbols: `DisplayVersion`, which returns the display interface version (to confirm that renderer and display server are using the same version of the protocol) and `CreateDisplay`, which returns a pointer to the custom subclass of `ExDisplay`.

16.2 The `ExDisplay` class

The header file `exdisplay.h` provides a definition for a C++ class called `ExDisplay`. This class is actually an *abstract base class* — it provides almost no functionality itself, but defines a standard interface for the renderer to communicate with a display server. A programmer can create a custom display server by *subclassing* the `ExDisplay` class, and supplying the implementations of the missing member functions.

The `ExDisplay` class defines the interface for the following public member functions, which your custom display driver must implement:

`ExDisplay::`**ExDisplay** `()`

> The constructor for the display server, which should require no arguments. You can set up data structures in the constructor if you have to, but it's probably more appropriate to do so in the `open` member function.

`ExDisplay::~`**ExDisplay** `()`

> The virtual destructor of your display server should close any files you opened, free any allocated memory, and so on.

`bool ExDisplay::`**needsScanline** `()`

> The renderer sends rectangles of pixel data to the display server. If your display server can only accept full scanlines for the output image, or can only accept the scanlines one at a time and in screen order, you should provide a `needsScanline` method that returns `true`. On the other hand, if your display server can receive arbitrary rectangles of the image pixels in arbitrary order, and you provide a `needsScanline` method that returns `false`, Entropy will take advantage of this ability.

`bool ExDisplay::`**canRedraw** `()`

> Some display servers may wish to be treated as "write once." If that is the case with your display server, it should provide a `canRedraw` method that returns `false`. On the other hand, if your display server can accept "rewrites" of the same pixels (for example, for re-renderings of an image that is being displayed on the screen), then you should provide a `canRedraw` method that returns `true`.

```
bool ExDisplay::open (const char *name,
                      int numChannels, const char **channelNames,
                      int xRes, int yRes,
                      int rxorigin, int ryorigin, int rxpixels, int rypixels,
                      int nparams, const Parameter *params)
```

The open method communicates all of the information from the graphics state (given by Display, Quantize, and so on). The name is the name of the output file. The numChannels argument gives the number of output channels for this file (for example, 3 for an RGB image), and channelNames is an array of pointers to the names of the channels. The total resolution of the image is xRes $\times$ Yres, but the crop region that is actually being rendered has origin at (rxorigin, ryorigin) and has a size of rxpixels $\times$ rypixels.

The params argument points to an array of nparams parameters, stored as structures of:

```
enum Type {
    TYPE_UNKNOWN = 0, TYPE_FLOAT, TYPE_INT, TYPE_STRING
};

struct Parameter {
    const char *token;
    Type type;
    int numElements; /* How many things of 'type' are there ? */
    const void *value;
};
```

The parameter list will always include the following parameters:

Name	Type	Elements	Description
dither	float	1	dither amplitude
quantize	float	4	zero, one, min, max
gain	float	1	Gain value for pixels
gamma	float	1	Gamma correction requested for pixels
NP	float	16	world-to-NDC matrix
Nl	float	16	world-to-camera matrix
HostComputer	string	1	Name of the computer
Software	string	1	Name of the renderer
OriginalSize	int	2	full resolution of the image
origin	int	2	upper-left (x,y) of the crop window
near	float	1	Near clipping plane z value
far	float	1	Far clipping plane z value

In addition to those required parameters, the list will also contain any optional parameters that were specified with the Display command (see Section 3.1.2).

This method should return true if it completed correctly. If an error occurs, it should call the error() method to set the error message, and return false.

`bool ExDisplay::`**start** `()`

This method is called by the client to mark the beginning of transmission of image pixels.

This method should return `true` if it completed correctly. If an error occurs, it should call the `error()` method to set the error message, and return `false`.

`bool ExDisplay::`**data** `(const PixelPacket &pixels)`

The `data()` method actually communicates a collection of pixels to the display server. The pixels are arranged in a rectangle. Which pixels are covered, and the data for those pixels, are all stored in the `PixelPacket` structure that is given as a reference. The `PixelPacket` structure, which is defined locally to the `ExDisplay` class, is:

```
struct PixelPacket {
    int x0, y0;
    int x1, y1;
    int pixelStride;
    int rowStride;
    const float *data;
};
```

The pixels cover the rectangle whose upper-left pixel coordinates are (`x0`, `y0`) and whose lower-right pixel coordinates are (`x1`, `y1`). The pixel data themselves are pointed to by `data`. The `pixelStride` gives the step size (in number of floats) between adjacent horizontal pixels, and `rowStride` gives the step size between adjacent vertical pixels.

This method should return `true` if it completed correctly. If an error occurs, it should call the `error()` method to set the error message, and return `false`.

`bool ExDisplay::`**finish** `()`

The `finish` method is called when all of the image pixels have been sent to the display server. It's possible, if your `canRedraw` function returned `true`, that additional `start()`/`finish()` cycles could occur.

This method should return `true` if it completed correctly. If an error occurs, it should call the `error()` method to set the error message, and return `false`.

The `ExDisplay` defines several "helper functions." You may find these useful when implementing your custom display drivers. These routines are provided for you by Entropy; you do not need to implement them on your own.

`float ExDisplay::`**exposure** `(float value, float gain, float invgamma)`

Applies gain and gamma correction as follows:

```
if (value > 0.f)
     return pow (value * gain, invgamma);
else return 0.0f;
```

`int ExDisplay::`**quantize** `(float value, float zero, float one, float Min, float Max, float ditheramp)`

Rescales the floating-point `value` and truncates to an integer as follows:

```
float dither = ditheramp * (2.0f*random01()-1.0f);
return (int)(.5+clamp(zero+value*(one-zero), Min, Max)+dither);
```

`const void * ExDisplay::`**findValue** `(const char *tok, int &index, int nparams, const Parameter *params)`

Given a list of `nparams` pointed to by `params`, search for one whose name is given by `tok`. Set `index` to the element number of the `params` array that matched, and return a pointer to the data itself. If there is no parameter in the list that matches the name `tok`, return `NULL`. This helper function is primarily intended to be called from the server's `open()` method.

`const void * ExDisplay::`**findValue** `(const char *tok, Type typeExpected, int numElementsExpected, int &index, int nparams, const Parameter *params)`

Given a list of `nparams` pointed to by `params`, search for one whose name is given by `tok` and whose type and number of elements also match `typeExpected` and `numElementsExpected`. Set `index` to the element number of the `params` array that matched, and return a pointer to the data itself. If there is no parameter in the list that matches, return `NULL`. This helper function is primarily intended to be called from the server's `open()` method.

`void` **error** `(const char *message)`

Sets the error message, which can be retrieved later by the client. The contents of `message` will be copied into a new area, so it's okay to change `message` after the `error()` call.

16.3 Making the DSO/DLL

Entropy may encounter a display request like this:

```
Display "out.tif" "mytiff" "rgba"
```

This indicates that RGBA data (4 channels) should be written to the file `out.tif` by the display server `"mytiff"`. If the display server is not the name of a format already known to Entropy (such as `"tiff"` or `"zfile"`), Entropy will search for a DSO named `"mytiff.so"` (on Unix-like systems) or a DLL named `"mytiff.dll"` (on Windows). Entropy will search through the several directories specified by `Option "searchpath" "display"` (see Section 3.1.3).

Assuming that the DSO/DLL matching the name is found, it will be scanned for two "public entry points" which must be present and have ordinary "C linkage." The declarations are as follows:

```
extern "C" {
  int DisplayVersion (void);
  ExDisplay *CreateDisplay (int nparams, const ExDisplay::Parameter *params);
}
```

The `DisplayVersion()` function must return the version of the display protocol, given in `exdisplay.h` as `EX_DISPLAY_VERSION`. This ensures that the DSO/DLL was compiled expecting the same protocol version that is being used by the renderer. If the protocols do not match, the display server will not be used by Entropy and an error message will be printed.

The second entry point, `CreateDisplay()`, returns a pointer to an `ExDisplay` — specifically, to a new `ExDisplay` object created to implement your custom display server. The `CreateDisplay()` function takes a parameter list just like the `ExDisplay::open()` method. It's up to you whether or not to do anything with the parameters.

Finally, the DSO/DLL must be compiled. Please refer to Appendix **??** for details on how to compile DSO/DLL libraries for your platform.

16.4 Example Display Server, Step by Step

As an example, let's describe writing a very simple TIFF file display server. For simplicity, let's assume it can only output 8-bit data. This is much simpler than the full-featured TIFF display server that comes built into Entropy, but it's still a useful example. We will create the `"mytiff"` driver as follows:[1]

1. We start by making sure our file, which we'll name `mytiff.cpp`, includes the `"exdisplay.h"` file and declares a display server path that appropriately subclasses the public interface of `ExDisplay`. Notice that we declare some private data that we'll need to store as we write the file.

```
#include "tiffio.h"
#include "exdisplay.h"

class MyTIFFDriver : public ExDisplay {
public:
    MyTIFFDriver () : tif(NULL), filename(NULL) { };
    virtual ~MyTIFFDriver ();

    virtual bool needsScanline ();
    virtual bool canRedraw ();
    virtual bool open (const char *name,
                       int numChannels, const char **channelNames,
                       int xRes, int yRes,
                       int rxorigin, int ryorigin,
                       int rxpixels, int rypixels,
                       int nparams, const Parameter *params);
    virtual bool start ();
    virtual bool data (const PixelPacket &pixels);
    virtual bool finish ();
private:
    TIFF *tif;
    char *filename;
    int xres, yres, fullxres, fullyres, xorigin, yorigin;
```

[1] We will assume use of the excellent TIFF I/O library that can be found at `www.libtiff.org`.

```
    float zero, one, minval, maxval, ditheramp;
    float gain, gamma;
    int samples, bitspersample, sampleformat;
    bool debug;
    int alphachannel;
};
```

Notice that we've declared some private data that we use to keep track of the open TIFF file handle, quantization levels, and so on.

2. To keep things simple, we won't buffer anything up. Rather, we'll just write scanlines as we get them. So we create methods to inform the client that we must receive whole single scanlines in order, and cannot accept repeats of already-transmitted scanlines:

```
bool MyTIFFDriver::needsScanline() { return true; }

bool MyTIFFDriver::canRedraw() { return false; }
```

3. The `open()` method is the tricky. We must look for particular items in the parameter list that give quantization, dither, gain, and gamma. For any of these that are not present, we must use reasonable defaults. Also, we must use the quantization levels to determine whether we are outputting 8-bit, 16-bit, 32-bit, or floating point images.

```
bool
MyTIFFDriver::open (const char *name,
                    int numChannels, const char **channelNames,
                    int xRes, int yRes,
                    int rxorigin, int ryorigin,
                    int rxpixels, int rypixels,
                    int nparams, const Parameter *params)
{
    filename = strdup(name);
    fullxres = xRes;
    fullyres = yRes;
    xorigin = rxorigin;
    yorigin = ryorigin;
    xres = rxpixels;
    yres = rypixels;
    samples = numChannels;

    // Set reasonable defaults
    zero = 0.0f;  one = 255.0f;
    minval = 0.0f;  maxval = 255.0f;
    ditheramp = 0.5f;
    gain = gamma = 1.0f;
    debug = false;

    // Try to identify the alpha channel
    alphachannel = -1;
    for (int c = 0;  c < numChannels;  ++c) {
        if (! strcmp(channelNames[c], "a"))
            alphachannel = c;
    }

    int index;
    float *fval;
    int *ival;
    if (fval = (float *)findValue("quantize", TYPE_FLOAT, 4, index,
```

```
                                    nparams, params)) {
        zero = fval[0];
        one = fval[1];
        minval = fval[2];
        maxval = fval[3];
    }
    if (fval = (float *)findValue("dither", TYPE_FLOAT, 1,
                                    index, nparams, params))
        ditheramp = fval[0];
    if (fval = (float *)findValue("gain", TYPE_FLOAT, 1, index,
                                    nparams, params))
        gain = fval[0];
    if (fval = (float *)findValue("gamma", TYPE_FLOAT, 1, index,
                                    nparams, params))
        gamma = fval[0];
    if (ival = (int *)findValue("debug", TYPE_INT, 1,
                                   index, nparams, params))
        debug = (ival[0] != 0);
    if (maxval == 0.0f) {
        sampleformat = SAMPLEFORMAT_IEEEFP;
        bitspersample = 32;
    } else {
        sampleformat = SAMPLEFORMAT_UINT;
        if (maxval > 255.0)
            bitspersample = 16;
        else bitspersample = 8;
    }
    if (debug)
        cerr << "Opening " << name << endl;
    return true;
}
```

4. Our `start()` method actually opens the TIFF file, setting all the tags based on the input parameters:

```
bool
MyTIFFDriver::start ()
{    if (debug)
        cerr << "start\n";
    tif = TIFFOpen (filename, "w");
    if (!tif) {
        error ("Could not open TIFF file");
        return false;
    }
    TIFFSetField (tif, TIFFTAG_IMAGEWIDTH, xres);
    TIFFSetField (tif, TIFFTAG_IMAGELENGTH, yres);
    TIFFSetField (tif, TIFFTAG_XPOSITION, (double)xorigin);
    TIFFSetField (tif, TIFFTAG_YPOSITION, (double)yorigin);
    TIFFSetField (tif, TIFFTAG_PIXAR_IMAGEFULLWIDTH, fullxres);
    TIFFSetField (tif, TIFFTAG_PIXAR_IMAGEFULLLENGTH, fullyres);
    TIFFSetField (tif, TIFFTAG_ORIENTATION, ORIENTATION_TOPLEFT);
    TIFFSetField (tif, TIFFTAG_PLANARCONFIG, PLANARCONFIG_CONTIG);
    TIFFSetField (tif, TIFFTAG_SAMPLESPERPIXEL, samples);
    TIFFSetField (tif, TIFFTAG_BITSPERSAMPLE, bitspersample);
    TIFFSetField (tif, TIFFTAG_SAMPLEFORMAT, sampleformat);
    TIFFSetField (tif, TIFFTAG_PHOTOMETRIC,
                  (samples > 1) ? PHOTOMETRIC_RGB : PHOTOMETRIC_MINISBLACK);
    if (samples == 4) {
        unsigned short sampleinfo = EXTRASAMPLE_ASSOCALPHA;
        TIFFSetField (tif, TIFFTAG_EXTRASAMPLES, 1, &sampleinfo);
    }
    return true;
}
```

5. The `data()` method must write a scanline to the TIFF file. The trickiest part is handling all the quantization and gamma correction, attempting to avoid unnecessary computation when the default values are used, and correctly converting to the appropriate integer types.

```
bool
MyTIFFDriver::data (const PixelPacket &pixels)
{
    if (debug)
        cerr << "data of range (" << pixels.x0 << ',' << pixels.y0 << ") - ("
             << pixels.x1 << ',' << pixels.y1 << ")\n";
    unsigned char *buf;
    int xsamps = xres*samples;
    buf = (unsigned char *)alloca (xsamps*bitspersample/8);
    float *fbuf = (float *)alloca (xsamps*sizeof(float));
    for (int y = pixels.y0;  y <= pixels.y1;  ++y) {
        int x, c, i;
        // Gamma correct, dither, and quantize
        int offset = (y-pixels.y0)*pixels.rowStride;
        for (x = pixels.x0, i = 0;  x <= pixels.x1;  ++x) {
            for (c = 0;  c < samples;  ++c)
                fbuf[i++] = pixels.data[offset+c];
            offset += pixels.pixelStride;
        }
        if (gain != 1.0f || gamma != 1.0f) {
            float invgamma = 1.0f/gamma;
            for (x = 0;  x < xsamps;  x += samples)
                for (c = 0;  c < samples;  ++c)
                    if (c != alphachannel)  // Don't correct alpha!
                        fbuf[x+c] = exposure (fbuf[x+c], gain, invgamma);
        }
        if (bitspersample == 8) {
            for (x = 0;  x < xsamps;  ++x)
                buf[x] = (unsigned char) quantize (fbuf[x], zero, one,
                                                   minval, maxval, ditheramp);
        } else if (bitspersample == 16) {
            for (x = 0;  x < xsamps;  ++x)
                ((unsigned short *)buf)[x] =
                    (unsigned short) quantize (fbuf[x], zero, one,
                                               minval, maxval, ditheramp);
        } else {
            for (x = 0;  x < xsamps;  ++x)
                ((float *)buf)[x] = fbuf[x];
        }
        TIFFWriteScanline (tif, buf, y-yorigin);
    }
    return true;
}
```

6. The `finish()` method merely closes the open TIFF file:

```
bool
MyTIFFDriver::finish()
{
    if (debug)
        cerr << "finish\n";
    if (tif)
        TIFFClose (tif);
    return true;
}
```

7. The `MyTIFFDriver` destructor must "clean up." There isn't much to do except to free the `filename` data that we had previously allocated with `strdup()`.

```
MyTIFFDriver::~MyTIFFDriver()
{
    if (debug)
        cerr << "destroying MyTIFFDriver\n";
    if (filename)
        free (filename);
}
```

8. Finally, we need to make the two "C" linkable entry points, `CreateDisplay()` and `DisplayVersion()`. We use the `EXPORT` macro defined in `exdisplay.h`, which contains the magic commands to make the symbols exportable (this is only needed on some operating systems).

```
extern "C" {

EXPORT ExDisplay *CreateDisplay (int, const ExDisplay::Parameter *)
{
    return new MyTIFFDriver();
}


EXPORT int DisplayVersion (void)
{
    return EX_DISPLAY_VERSION;
}

}
```

The full source code for this sample display server can be found in the `examples/src/display/` directory in the Entropy distribution. In addition, that directory contains the source code for a sample display server that can be used to communicate between Entropy display servers and *PRMan* display servers.

Part IV

Appendices

Appendix A

Compatibility Guide

A.1 Differences between Entropy, BMRT, and PRMan

In the following subsections, when we refer to "BMRT," we are specifically comparing to BMRT 2.6, and when we refer to "PRMan," we are comparing to Pixar's PhotoRealistic RenderMan 3.9.

A.1.1 Geometric Primitives

Entropy fully supports the `Curves` and `Blobby` primitives, whereas BMRT does not.

If `Points` primitives are very large on screen in *PRMan*, they will appear as hexagons. With Entropy, `Points` primitives are perfectly round no matter what size they are on screen.

Entropy does not support CSG, whereas both PRMan and BMRT do. Entropy effectively ignores `SolidBegin` and `SolidEnd`, drawing all parts of CSG objects.

Entropy supports the primitive variable storage class `"facevarying"`, which is not supported by BMRT or *PRMan* 3.9.

A.1.2 Rendering Pipeline Issues

BMRT is strictly a conventional ray tracer — for each pixel, one or more rays are intersected with scene geometry to reveal the closest object behind the pixel. Entropy uses a much more efficient "scanline" algorithm to determine which objects are visible, though shaders can still trace rays for reflections and refractions. The main result of this is that if you aren't using a substantial amount of ray tracing or global illumination, a given scene will render much faster and using much less memory in Entropy than it would in BMRT.

PRMan has an `"eyesplits"` option that is used to help deal with objects that cross the camera ($z = 0$) plane. Entropy has no such equivalent option, and has no trouble dealing with objects that cross the camera plane, or even that intersect the camera position itself.

Entropy can have any number of display output channels, and they may be individually assigned any filter and width. *PRMan* has a limit of 16 display output channels and they are all filtered with `"box" 1 1`. BMRT does not support multiple output channels at all.

Entropy supports arbitrary clipping planes with `ClippingPlane`.

A.1.3 Antialiasing

Entropy's new antialiasing and hidden-surface algorithm is not based on point sampling (as are BMRT and PRMan), and so there isn't a direct analog to `PixelSamples`. There are now two new options: `Option "limits" "spatialquality"` and `Option "limits" "temporalquality"` that directly control the antialiasing quality levels (see Chapter 3 for details). For backwards compatibility, `PixelSamples` now indirectly sets these two attributes, such that the overall quality closely matches what you would get with the same `PixelSamples` setting for BMRT or PRMan (even though it's doing something *entirely* different under the covers).

Entropy supports motion blur, and depth of field, whereas BMRT 2.6 does not.

A.1.4 Shading

Although Entropy, BMRT, and PRMan have almost completely compatible shading language source code syntax, their formats for compiled shaders are not interchangeable. To avoid confusion, Entropy's shader compiler is named `sle` and names its compiled shaders with the `.sle` extension, versus BMRT's `slc` compiler and `.slc` extension, and PRMan's `shader` compiler and `.slo` extension. Similarly, Entropy shaders are interrogated with a library now called `libsleargs`, described in a header file `sleargs.h`.

In BMRT, as each screen ray determines the closest object, shaders are run at the single location that the ray hit. (Strictly speaking, BMRT runs shaders at the intersection and also at two nearby points, in order to estimate derivatives.) In Entropy, points are not shaded only one (or three) at a time, but rather large sections of geometric primitives are shaded. This leads to *much* faster execution of shaders, better texture access coherence, more accurate calculations of derivatives, and many other benefits.

In Entropy and PRMan, `"current"` space (the space that all parameters and global variables are in when the shader executes) is the same as `"camera"` space, whereas in BMRT `"current"` space is `"world"` space.

In shader source code, Entropy allows `//` to signify that the remainder of the text line is a comment (just like C++). BMRT's and PRMan's shader compilers do not treat `//` as a comment marker.

Entropy defines the `ENTROPY` and `EXLUNA` symbols (which you can use for `#ifdef` and so on) during shader compilation. BMRT defines the `BMRT` and `EXLUNA` symbols. PRMan does not define either of these symbols.

Entropy looks for the preprocessor, `slpp`, specifically in the `$ENTROPYHOME/bin` directory. You *must* set `$ENTROPYHOME` for `slc` to compile shaders.

Entropy's `specular()` function is quite different than BMRT (we think the new one looks nicer), and is similar (but not identical) to PRMan's `specular()` function.

PRMan's `noise()` functions differ in appearance from Entropy's. Objects whose appearances are largely due to `noise()` patterns will appear different. However, PRMan's and Entropy's `noise()` functions are statistically similar.

Entropy extends the `texture()` and `environment()` functions with optional `"firstchannel"`, `"maxhitdist"`, `"alpha"`, and `"hitdist"` parameters. (See Section 5.6.7.) These parameters are not understood (but should be safely ignored) by PRMan and BMRT.

Entropy extends the `Option` and `Attribute` scene file commands, and the `option()` and `attribute()` shading language functions, to allow users to create and set their own options and attributes, and to retrieve their values from shaders. BMRT and PRMan do not allow this.

A.1.5 Ray Tracing

Entropy supports a variety of ray tracing and global illumination functionality, described in Sections 5.6.8, 3.1.3, and 3.2.2.

Compared to BMRT, Entropy has extended functionality of the `trace()` function: the position parameter is optional (defaulting to `P` if not supplied), and it can take optional token/value arguments analogous to environment mapping routines (see Section 5.6.8). However, we strongly recommend that `trace` be avoided, and instead the new extensions to `environment()` be used for ray tracing (see Section 5.6.7). Similarly, though BMRT's `visibility` routine is supported by Entropy, we recommend that ray traced shadows be computed by using the extensions to `shadow()` (Section 5.6.7).

BMRT had all objects appear in ray traced shadows and reflections by default. Entropy, by default, does not include any objects in either ray traced shadows or reflections. Objects may be individually be put in the ray traced shadow or reflection lists by using `Attribute "visibility"` (see Section 3.2.2). Note that this replaces BMRT's `Attribute "render" "visibility"`, which had some notational sloppiness and is now considered deprecated.

PRMan does not perform any ray tracing or global illumination, will not make use of any of the aptions or attributes related to ray tracing, indirect illumination, or caustics, and has no equivalent to the `indirect` or `caustic` shaders. PRMan's `trace()` function always returns zero and does not take the optional texture-like arguments (such as `"blur"`), and the `raylevel()`, `isshadow()` functions do not exist at all in PRMan. *PRMan* does not support the ray-tracing extensions to `environment()` and `shadow()`.

A.1.6 Miscellaneous

BMRT 2.6 and earlier would always check the current working directory for shaders, textures, and RIB archive files. If the files were not found in the current directory, the directories in the appropriate search path would be checked in turn. Entropy always strictly respects the search path set in the scene file (or in `.entropyrc`). If you want the current directory searched, you need to be sure that `"."` is explicitly in the search path.

Many of the sample shaders that come with Entropy have been changed from the similarly-named shaders that come with BMRT and PRMan, in order to be improved or to take advantage of new Entropy features not present in other renderers. In some cases parameters have been added, removed, or renamed. CAUTION!

Older versions of `rgl` did not support object instancing (`ObjectBegin`, `ObjectEnd`, `ObjectInstance`). Entropy's `rgl` (as well as `entropy`) support object instancing from RIB files. However, we consider this a deprecated feature and strongly discourage its use, favoring `ReadArchive` or `Procedural` instead.

BMRT's `Attribute "render" "truedisplacement"` is unnecessary with Entropy and is considered deprecated. By default, all objects with displacement shaders undergo

true displacement. However, there is a new hint, `Attribute "render" "tracedisplacements"`, that can instruct the renderer to use a cheaper bump-mapping approximation to displacement for ray tracing of reflections or shadows (but in either case, true displacement is used for the camera view of the object).

Entropy has a new, C++-based, interface for programming your own display servers. This is described in Chapter 16.

Entropy's `mkmip` program allows texture wrap mode `"mirror"`.

Entropy accepts either integer or string light identifiers for the `LightSource`, `AreaLightSource`, and `Illuminate` RIB commands. This is compatible with *PRMan* 10. Earlier versions of Entropy, *PRMan*, and *BMRT* only accepted integers as light identifiers.

A.2 Deprecated Functionality

Deprecated

There are a number of features available in BMRT or other compatible renderers which we now consider *deprecated*. Though we still support these commands or features, we consider them obsolete and they exist in Entropy strictly for backward-compatibility with existing scene files designed for other renderers. It's possible that these commands will someday be disabled entirely. In the mean time, while they are currently harmless, we discourage use of these features and urge you to use the preferred alternatives.

A.2.1 Scene File Commands

The following scene file commands are supported for backwards compatibility only, and should be avoided when constructing new scenes:

Bound `[`*xmin xmax ymin ymax zmin zmax*`]`

This is not used by Entropy.

ObjectBegin *id*
ObjectEnd
ObjectInstance *id*

`ObjectBegin`, which takes an integer *id* argument, and `ObjectEnd` bracket the definition of an object. `ObjectInstance` adds the object (identified by the same integer *id* used at its creation) to the scene. An object may be instanced multiple times.

Entropy supports instanced objects in scene files for backward compatibility only; its use is considered deprecated. `ReadArchive` (or better, `Procedural`) should be used if you want to make multiple identical copies of an object.

ShadingInterpolation *type*

Previously used to set interpolation to either `"smooth"` or `"constant"`. This is not used by Entropy, and even renderers that support it should always use `"smooth"`.

SolidBegin *solidtype*
SolidEnd

Entropy does not support CSG. The `SolidBegin` and `SolidEnd` statements are effectively ignored, and all parts of CSG objects are drawn.

Index