The RenderMan Interface

Version 3.2

July, 2000

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This document is version 3.2 of the RenderMan Interface Specification, issued July, 2000. It supercedes version 3.1, originally published in September, 1989 and revised in July, 1995, and version 3.0, originally published in May, 1988.

Version 3.2 makes fundamental changes to version 3.1, including the addition of many new features, corrections of unclear or ambiguous material, and deletion of API calls that have been depricated over the years.

This document is the official technical specification for the RenderMan Interface. It is quite terse and requires substantial prior knowledge of computer graphics in general and photorealistic image synthesis in particular. For a more casual reference to the RenderMan Interface, the reader is directed to *Advanced RenderMan: Creating CGI for Motion Pictures* (Anthony Apodaca and Larry Gritz, 1999). The first and second printings of *Advanced RenderMan* correspond (except for minor errata) to version 3.2 of the RenderMan Interface Specification. Readers are also directed to *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics* (Steve Upstill 1989), which corresponds (except for minor errata) to version 3.1 of the RenderMan Interface Specification.

Appendix I gives an overview of what has changed between version 3.1 and 3.2 of the RenderMan Interface Specification.

Part I

The RenderMan Interface

Section 1

INTRODUCTION

The RenderMan Interface is a standard interface between modeling programs and rendering programs capable of producing photorealistic quality images. A rendering program implementing the RenderMan Interface differs from an implementation of earlier graphics standards in that:

- A photorealistic rendering program must simulate a real camera and its many attributes besides just position and direction of view. High quality implies that the simulation does not introduce artifacts from the computational process. Expressed in the terminology of computer graphics, this means that a photorealistic rendering program must be capable of:
  - hidden surface removal so that only visible objects appear in the computed image,
  - spatial filtering so that aliasing artifacts are not present,
  - dithering so that quantization artifacts are not noticeable,
  - temporal filtering so that the opening and closing of the shutter causes moving objects to be blurred,
  - and depth of field so that only objects at the current focal distance are sharply in focus.
- A photorealistic rendering program must also accept curved geometric primitives so that not only can geometry be accurately displayed, but also so that the basic shapes are rich enough to include the diversity of man-made and natural objects. This requires patches, quadrics, and representations of solids, as well as the ability to deal with complicated scenes containing on the order of 10,000 to 1,000,000 geometric primitives.
- A photorealistic rendering program must be capable of simulating the optical properties of different materials and light sources. This includes surface shading models that describe how light interacts with a surface made of a given material, volume shading models that describe how light is scattered as it traverses a region in space, and light source models that describe the color and intensity of light emitted in different directions. Achieving greater realism often requires that the surface properties of an object vary. These properties are often controlled by texture mapping an image

onto a surface. Texture maps are used in many different ways: direct image mapping to change the surface's color, transparency mapping, bump mapping for changing its normal vector, displacement mapping for modifying position, environment or reflection mapping for efficiently calculating global illumination, and shadow maps for simulating the presence of shadows.

The RenderMan Interface is designed so that the information needed to specify a photorealistic image can be passed to different rendering programs compactly and efficiently. The interface itself is designed to drive different hardware devices, software implementations and rendering algorithms. Many types of rendering systems are accommodated by this interface, including z-buffer-based, scanline-based, ray tracing, terrain rendering, molecule or sphere rendering and the Reyes rendering architecture. In order to achieve this, the interface does not specify how a picture is rendered, but instead specifies what picture is desired. The interface is designed to be used by both batch-oriented and real-time interactive rendering systems. Real-time rendering is accommodated by ensuring that all the information needed to draw a particular geometric primitive is available when the primitive is defined. Both batch and real-time rendering is accommodated by making limited use of inquiry functions and call-backs.

The RenderMan Interface is meant to be complete, but minimal, in its transfer of scene descriptions from modeling programs to rendering programs. The interface usually provides only a single way to communicate a parameter; it is expected that the modeling front end will provide other convenient variations. An example is color coordinate systems – the RenderMan Interface supports multiple-component color models because a rendering program intrinsically computes with an n-component color model. However, the RenderMan Interface does not support all color coordinate systems because there are so many and because they must normally be immediately converted to the color representation used by the rendering program. Another example is geometric primitives – the primitives defined by the RenderMan Interface are considered to be rendering primitives, not modeling primitives. The primitives were chosen either because special graphics algorithms or hardware is available to draw those primitives, or because they allow for a compact representation of a large database. The task of converting higher-level modeling primitives to rendering primitives must be done by the modeling program.

The RenderMan Interface is not designed to be a complete three-dimensional interactive programming environment. Such an environment would include many capabilities not addressed in this interface. These include: 1) screen space or two-dimensional primitives such as annotation text, markers, and 2-D lines and curves, and 2) user-interface issues such as window systems, input devices, events, selecting, highlighting, and incremental redisplay.

The RenderMan Interface is a collection of procedures to transfer the description of a scene to the rendering program. These procedures are described in Part I. A rendering program takes this input and produces an image. This image can be immediately displayed on a given display device or saved in an image file. The output image may contain color as well as coverage and depth information for postprocessing. Image files are also used to input texture maps. This document does not specify a "standard format" for image files.

The RenderMan Shading Language is a programming language for extending the predefined functionality of the RenderMan Interface. New materials and light sources can be

created using this language. This language is also used to specify volumetric attenuation, displacements, and simple image processing functions. All required shading functionality is also expressed in this language. A shading language is an essential part of a high-quality rendering program. No single material lighting equation can ever hope to model the complexity of all possible material models. The RenderMan Shading Language is described in Part II of this document.

1.1 Features and Capabilities

The RenderMan Interface was designed in a top-down fashion by asking what information is needed to specify a scene in enough detail so that a photorealistic image can be created. Photorealistic image synthesis is quite challenging and many rendering programs cannot implement all of the features provided by the RenderMan Interface. This section describes which features are required and which are considered advanced, and therefore optional, capabilities. The set of required features is extensive in order that application writers and end-users may reasonably expect basic compatibility between, and a high level of performance from, all implementations of the RenderMan Interface. Advanced capabilities are optional only in situations where it is reasonable to expect that some rendering programs are algorithmically incapable of supporting that capability, or where the capability is so advanced that it is reasonable to expect that most rendering implementations will not be able to provide it.

1.1.1 Required features

All rendering programs which implement the RenderMan Interface must implement the interface as specified in this document. Implementations which are provided as a linkable C library must provide entry points for all of the subroutines and functions, accepting the parameters as described in this specification. All of the predefined types, variables and constants (including the entire set of constant **RtToken** variables for the predefined string arguments to the various RenderMan Interface subroutines) must be provided. The C header file ri.h (see Appendix C, Language Binding Details) describes these data items.

Implementations which are provided as prelinked standalone applications must accept as input the complete RenderMan Interface Bytestream (RIB). Such implementations may also provide a complete RenderMan Interface library as above, which contains subroutine stubs whose only function is to generate RIB.

All rendering programs which implement the RenderMan Interface must:

- provide the complete hierarchical graphics state, including the attribute and transformation stacks and the active light list.
- perform orthographic and perspective viewing transformations.
- perform depth-based hidden-surface elimination.
- perform pixel filtering and antialiasing.

- perform gamma correction and dithering before quantization.
- produce picture files containing any combination of RGB, A, and Z. The resolutions of these files must be as specified by the user.
- provide all of the geometric primitives described in the specification, and provide all of the standard primitive variables applicable to each primitive.
- provide the ability to perform shading calculations using user-supplied RenderMan Shading Language programs. (See Part II, The RenderMan Shading Language.)
- provide the ability to index texture maps, environment maps, and shadow depth maps. (See the section on Basic texture maps.)
- provide the fifteen standard light source, surface, volume, displacement, and imager shaders required by the specification. Any additional shaders, and any deviations from the standard shaders presented in this specification, must be documented by providing the equivalent shader expressed in the RenderMan Shading Language.

Rendering programs that implement the RenderMan Interface receive all of their data through the interface. There must not be additional subroutines required to control or provide data to the rendering program. Data items that are substantially similar to items already described in this specification will be supplied through the normal mechanisms, and not through any of the implementation-specific extension mechanisms (**RiAttribute**, **RiGeometry** or **RiOption**). Rendering programs may not provide nonstandard alternatives to the existing mechanisms, such as any alternate language for programmable shading.

1.1.2 Advanced Capabilities

Rendering programs may also provide one or more of the following advanced capabilities, though it is recognized that algorithmic limitations of a particular implementation may restrict its ability to provide the entire feature set. If a capability is not provided by an implementation, a specific default is required (as described in the individual sections). A subset of the full functionality of a capability may be provided by a rendering program. For example, a rendering program might implement Motion Blur, but only of simple transformations, or only using a limited range of shutter times. Rendering programs should describe their implementation of the following optional capabilities using the terminology in the following list.

Solid Modeling The ability to define solid models as collections of surfaces and combine them using the set operations intersection, union and difference. (See the section on *Solids and Spatial Set Operations*, p. 93.)

Level of Detail The ability to specify several definitions of the same model and have one selected based on the estimated screen size of the model. (See the section on *Detail*, p. 52.)

Motion Blur The ability to process moving primitives and antialias them in time. (See Section 6, *Motion*.)

Depth of Field The ability to simulate focusing at different depths. (See the section on *Camera*, p. 20.)

Special Camera Projections The ability to perform nonstandard camera projections such as spherical or Omnimax projections. (See the section on *Camera*, p. 20.)

Displacements The ability to handle displacements. (See the section on *Transformations*, p. 56.)

Spectral Colors The ability to calculate colors with an arbitrary number of spectral color samples. (See the section on *Additional Options*, p. 35.)

Volume Shading The ability to attach and evaluate volumetric shading procedures. (See the section on *Volume shading*, p. 47.)

Ray Tracing The ability to evaluate global illumination models using ray tracing. (See the section on *Shading and Lighting Functions*, p. 143.)

Global Illumination The ability to evaluate indirect illumination models using radiosity or other global illumination techniques. (See the section on *Illuminance and Illuminate Statements*, p. 129.)

Area Light Sources The ability to illuminate surfaces with area light sources. (See the section on *Light Sources*, p. 42.)

1.2 Structure of this Document

Part I of this document describes the scene description interface. Section 2 describes the language binding and conventions used in this document. Section 3 provides a brief introduction to the RenderMan Shading Language and its relationship to the RenderMan Interface. Section 4 describes the graphics state maintained by the interface. The state is divided into *options* which control the overall rendering process, and *attributes* which describe the properties of individual geometric primitives. Rendering options include camera and display options as well as the type of hidden surface algorithm being used. Rendering attributes include shading (light sources, surface shading functions, colors, etc.) and geometric attributes including transformations. Section 5 describes the basic geometric surfaces and solid modeling representations used by the RenderMan Interface. Section 6 describes the specification of moving geometry and time-varying shading parameters. Finally, Section 7 describes the process of generating texture maps from standard image files, reporting errors, and manipulating archive files.

Section 2

LANGUAGE BINDING SUMMARY

In this document, the RenderMan Interface is described in the ANSI C language. Other language bindings may be proposed in the future.

2.1 C Binding

All types, procedures, tokens, predefined variables and utility procedures mentioned in this document are required to be present in all C implementations that conform to this specification. The C header file which declares all of these required names, ri.h, is listed in Appendix C, Language Binding Details.

The RenderMan Interface requires the following types:

```
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 [4];
typedef RtFloat     RtMatrix [4][4];
typedef RtFloat     RtBasis [4][4];
typedef RtFloat     RtBound [6];
typedef char        *RtString;
typedef void        *RtPointer;
typedef void        RtVoid;
typedef RtFloat     (RtFilterFunc)(RtFloat, RtFloat, RtFloat, RtFloat);
typedef RtFloat     (RtErrorHandler)(RtInt, RtInt, char *);
typedef RtFloat     (RtProcSubdivFunc)(RtPointer, RtFloat);
typedef RtFloat     (RtProcFreeFunc)(RtPointer);
typedef RtFloat     (RtArchiveCallback)(RtToken, char *, ...);
typedef RtPointer   RtObjectHandle;
```

```
typedef RtPointer    RtLightHandle;
typedef RtPointer    RtContextHandle;
```

All procedures and values defined in the interface are prefixed with **Ri** (for RenderMan Interface). All types are prefixed with **Rt** (for RenderMan type). Boolean values are either RI_FALSE or RI_TRUE. Special floating point values RI_INFINITY and RI_EPSILON are defined. The expression −RI_INFINITY has the obvious meaning. The number of components in a color is initially three, but can be changed (See the section Additional options, p. 33). A bound is a bounding box and is specified by 6 floating point values in the order *xmin, xmax, ymin, ymax, zmin, zmax*. A matrix is an array of 16 numbers describing a 4 by 4 transformation matrix. All multidimensional arrays are specified in row-major order, and points are assumed to be represented as row vectors, not column vectors. For example, a 4 by 4 translation matrix to the location (2,3,4) is specified with

```
{ { 1.0, 0.0, 0.0, 0.0},
  { 0.0, 1.0, 0.0, 0.0},
  { 0.0, 0.0, 1.0, 0.0},
  { 2.0, 3.0, 4.0, 1.0} }
```

Tokens are strings that have a special meaning to procedures implementing the interface. These meanings are described with each procedure. The capabilities of the RenderMan Interface can be extended by defining new tokens and passing them to various procedures. The most important of these are the tokens identifying variables defined by procedures called *shaders*, written in the Shading Language. Variables passed through the RenderMan Interface are bound by name to shader variables. To make the standard predeclared tokens and user-defined tokens similar, RenderMan Interface tokens are represented by strings. Associated with each of the standard predefined tokens, however, is a predefined string constant that the RenderMan Interface procedures can use for efficient parsing. The names of these string constants are derived from the token names used in this document by prepending an RI_ to a capitalized version of the string. For example, the predefined constant token for "rgb" is RI_RGB. The special predefined token RI_NULL is used to specify a *null* token.

In the C binding presented in this document, parameters are passed by value or by reference. C implementations of the RenderMan Interface are expected to make copies of any parameters whose values are to be retained across procedure invocations.

Many procedures in the RenderMan Interface have variable length parameter lists. These are indicated by the syntactical construct *...parameterlist...* in the procedure's argument list. In the C binding described, a *parameterlist* is a sequence of pairs of arguments, the first being an **RtToken** and the second being an **RtPointer**, an untyped pointer to an array of either **RtFloat**, **RtString** or other values. The list is terminated by the special token RI_NULL.

In addition, each such procedure has an alternate vector interface, which passes the *parameterlist* as three arguments: an **RtInt** indicating the length of the parameter list; an array of that length that contains the **RtToken**s; and another array of the same length that contains the **RtPointer**s. This alternate procedure is denoted by appending an uppercase V to the procedure name.

For example the procedure **RiFoo** declared as

RiFoo(*...parameterlist...*)

could be called in the following ways:

```
RtColor colors;
RtPoint points;
RtFloat one_float;
RtToken tokens[3];
RtPointer values[3];
RiFoo ( RI_NULL);
RiFoo ((RtToken)"P", (RtPointer)points, (RtToken)"Cs", (RtPointer)colors,
        (RtToken)"Kd", (RtPointer)&one_float, RI_NULL);
RiFoo (RI_P, (RtPointer)points, RI_CS, (RtPointer)colors,
        RI_KD, (RtPointer)&one_float, RI_NULL);
tokens[0] = RI_P; values[0] = (RtPointer)points;
tokens[1] = RI_CS; values[1] = (RtPointer)colors;
tokens[2] = RI_KD; values[2] = (RtPointer)&one_float;
RiFooV ( 3, tokens, values);
```

It is not the intent of this document to propose that other language bindings use an identical mechanism for passing parameter lists. For example, a Fortran or Pascal binding might pass parameters using four arguments: an integer indicating the length of the parameter list, an array of that length that contains the tokens, an array of the same length containing integer indices into the final array containing the real values. A Common Lisp binding would be particularly simple because it has intrinsic support for variable length argument lists.

There may be more than one *rendering context*. This would allow a program to, for example, output to mutiple RIB files. RenderMan Interface procedure calls apply to the *currently active context*. At any one time, there is at most one globally active rendering context. The RenderMan Interface is not intended to be reentrant. In other words, the active context is truly global to a program process, and there cannot be have multiple simultaneous threads in one process, each with a different active context. Following is an example of writing to multiple contexts, in which a sphere is written to one RIB file and a cylinder is written to a different RIB file (the semantics of the context switching routines are presented in Section 4).

```
RtContextHandle ctx1, ctx2;
RiBegin ("file1.rib");
ctx1 = RiGetContext ( );
RiBegin ("file2.rib");
ctx2 = RiGetContext ( );
...
RiContext (ctx1);
RiSphere (1, -1, 1, 360, RI_NULL);
RiContext (ctx2);
RiCylinder (1, -1, 1, 360, RI_NULL);
RiEnd ( ); /* Ends context 2 */
```

RiContext (ctx1);
...

There is no RIB equivalent for context switching. Additionally, other language bindings may have no need for these routines, or may provide an obvious mechanism in the language for this facility (such as class instances and methods in C++).

2.2 Bytestream Protocol

This document also describes a byte stream representation of the RenderMan Interface, known as the RenderMan Interface Bytestream, or RIB. This byte stream serves as both a network transport protocol for modeling system clients to communicate requests to a remote rendering service, and an archive file format to save requests for later submission to a renderer.

The RIB protocol provides both an ASCII and binary encoding of each request, in order to satisfy needs for both an understandable (potentially) interactive interface to a rendering server and a compact encoded format which minimizes transmission time and file storage costs. Some requests have multiple versions, for efficiency or to denote special cases of the request.

The semantics of each RIB request are identical to the corresponding C entry point, except as specifically noted in the text. In Part I of this document, each RIB request is presented in its ASCII encoding, using the following format:

RIB BINDING

Request parameter1 parameter2... parameterN

Explanation of the special semantics of the RIB protocol for this request.

At the top of the description, *parameter1* through *parameterN* are the parameters that the request requires. The notation '–' in the parameter position indicates that the request expects no parameters. Normally the parameter names suggest their purpose, e.g., *x*, *y*, or *angle*.

In RIB, square brackets ([and]) delimit arrays. Integers will be automatically promoted if supplied for parameters which require floating point values. A parameter list is simply a sequence of string-array pairs. There is no explicit termination symbol as in the C binding. Example parameter lists are:

```
"P" [0 1 2 3 4 5 6 7 8 9 10 11]
"distance" [.5] "roughness" [1.2]
```

The details of the lexical syntax of both the ASCII and binary encodings of the RIB protocol are presented in Appendix C, Language Binding Details.

2.3 Additional Information

Finally, the description of each RenderMan Interface request provides an example and cross-reference in the following format:

EXAMPLE

Request 7 22.9

SEE ALSO

RiOtherRequest

Some examples are presented in C, others in RIB, and a few are presented in both bindings (for comparison). It should be obvious from the syntax which binding is which.

Section 3

RELATIONSHIP TO THE RENDERMAN SHADING LANGUAGE

The capabilities of the RenderMan Interface can be extended by using the Shading Language. The Shading Language is described in Part II of this document. This section describes the interaction between the RenderMan Interface and the Shading Language.

Special procedures, called *shaders*, are declared in this language. The argument list of a shader declares variables that can be passed through the RenderMan Interface to a shader. For example, in the shading language a shader called *weird* might be declared as follows:

```
surface
weird ( float f = 1.0; point p = (0,0,0) )
{
        Cs = Ci * mod ( length(P-p)*f - s + t, 1.0 );
}
```

The shader *weird* is referred to by name and so are its variables.

```
RtFloat foo;
RtPoint bar;

RiSurface ("weird", "f", (RtPointer)&foo, "p", (RtPointer)&bar, RI_NULL);
```

passes the value of *foo* to the Shading Language variable *f* and the value *bar* to the variable *p*. Note that since all parameters are passed as arrays, the single float must be passed by reference.

In order to pass shading language variables, the RenderMan Interface must know the type of each variable defined in a shader. All predefined shaders predeclare the types of the variables that they use. Certain other variables, such as position, are also predeclared. Additional variables are declared with:

RtToken
RiDeclare (char *name, char *declaration)

Declare the name and type of a variable. The declaration indicates the size and semantics of values associated with the variable, or may be RI_NULL if there are no associated values. This information is used by the renderer in processing the variable argument list semantics of the RenderMan Interface. The syntax of *declaration* is:

[*class*] [*type*] ['[' *n* ']']

where class may be *constant*, *uniform*, *varying* (as in the shading language), or *vertex* (position data, such as bicubic control points), and type may be one of: *float*, *integer*, *string*, *color*, *point*, *vector*, *normal*, *matrix*, and *hpoint*. Most of these types are described in Section 11, Types, which describes the data types available in the Shading Language. The Shading Language does not have an *integer* type, but integers may be passed through the interface as arguments to options or attributes. Additionally, the *hpoint* is used to describe 4D homogeneous coordinates (for example, used to describe NURBS control points). Any *hpoint* values are converted to ordinary *points* by dividing by the homogeneous coordinate just before passing the value to the shader.

The optional bracket notation indicates an array of *n type* items, where *n* is a positive integer. If no array is specified, one item is assumed. If a *class* is not specified, the identifier is assumed to be uniform.

RiDeclare also installs *name* into the set of known tokens and returns a constant token which can be used to indicate that variable. This constant token will generally have the same efficient parsing properties as the 'RI_' versions of the predefined tokens.

RIB BINDING

Declare *name declaration*

EXAMPLE

```
RiDeclare ( "Np", "uniform point" );
RiDeclare ( "Cs", "varying color" );

Declare "st" "varying float[2]"
```

In addition to using **RiDeclare** to globally declare the type of a variable, the type and storage class of a variable may be declared "in-line." For example:

```
RiSurface ( "mysurf", "uniform point center", &center, RI_NULL );
RiPolygon ( 4, RI_P, &points, "varying float temperature", &temps, RI_NULL );

Patch "bilinear" "P" [...] "vertex point Pref" [...] "varying float[2] st" [...]
```

When using these in-line declarations, the storage class and data type prepend the token name. Thus, any RenderMan Interface routines or RIB directives that take user-specified data will examine the tokens, treating multi-word tokens that start with class and type names as an in-line declaration. The scope of an in-line declaration is just one data item — in other words, it does not alter the global dictionary or affect any other data transmitted

through the interface. Any place where user data is used and would normally require a preceeding **RiDeclare**, it is also legal to use an in-line declaration.

The storage modifiers vertex, varying, uniform, and constant are discussed in the section on Uniform and Varying Variables in Part II and in Section 5, Geometric Primitives. All procedure parameter tokens and shader variable name tokens named in this document are standard and are predefined by all implementations of the RenderMan Interface. In addition, a particular implementation may predeclare other variables for use with implementation-specific options, geometry, etc.

Whenever a point, vector, normal, matrix, or hpoint variable is passed through the RenderMan Interface to shaders, the values are assumed to be relative to the current coordinate system. This is sometimes referred to as *object* coordinates. Different coordinate systems are discussed in the Camera section.

Whenever colors are passed through the RenderMan Interface, they are expected to have a number of floats equal to the number of color samples being used by the interface. This defaults to 3, but can be changed by the user (see the section on Additional options).

Section 4

GRAPHICS STATE

The RenderMan Interface is similar to other graphics packages in that it maintains a *graphics state*. The graphics state contains all the information needed to render a geometric primitive. RenderMan Interface commands either change the graphics state or render a geometric primitive. The graphics state is divided into two parts: a global state that remains constant while rendering a single image or frame of a sequence, and a current state that changes from geometric primitive to geometric primitive. Parameters in the global state are referred to as *options*, whereas parameters in the current state are referred to as *attributes*. Options include the camera and display parameters, and other parameters that affect the quality or type of rendering in general (e.g., global level of detail, number of color samples, etc.). Attributes include the parameters controlling appearance or shading (e.g., color, opacity, surface shading model, light sources, etc.), how geometry is interpreted (e.g., orientation, subdivision level, bounding box, etc.), and the current modeling matrix. To aid in specifying hierarchical models, the attributes in the graphics state may be pushed and popped on a graphics state stack.

The graphics state also maintains the interface *mode*. The different modes of the interface are entered and exited by matching Begin-End command sequences.

RiBegin (**RtToken** *name*)
RiEnd (void)

> **RiBegin** creates and initializes a new rendering context, setting all graphics state variables to their default values, and makes the new context the active one to which subsequent **Ri** routines will apply. Any previously active rendering context still exists, but is no longer the active one. The *name* may be the name of a renderer, to select among various implementations that may be available, or the name of the file to write (in the case of a RIB generator). RI_NULL indicates that the default implementation and/or output file should be used.
>
> **RiEnd** terminates the active rendering context, including performing any cleanup operations that need to be done. After **RiEnd** is called, there is no active rendering context until another **RiBegin** or **RiContext** is called.
>
> All other RenderMan Interface procedures must be called within an active context (the only exceptions are **RiErrorHandler**, **RiOption**, and **RiContext**).

RtContextHandle RiGetContext (void)
RiContext (**RtContextHandle** *handle*)

RiGetContext returns a handle for the current active rendering context. If there is no active rendering context, RI_NULL will be returned. **RiContext** sets the current active rendering context to be the one pointed to by *handle*. Any previously active context is not destroyed.

There is no RIB equivalent for these routines. Additionally, other language bindings may have no need for these routines, or may provide an obvious mechanism in the language for this facility (such as class instances and methods in C++).

RiFrameBegin (**RtInt** *frame*)
RiFrameEnd (void)

The bracketed set of commands **RiFrameBegin**-**RiFrameEnd** mark the beginning and end of a single frame of an animated sequence. *frame* is the number of this frame. The values of all of the rendering options are saved when **RiFrameBegin** is called, and these values are restored when **RiFrameEnd** is called.

All lights and retained objects defined inside the **RiFrameBegin**-**RiFrameEnd** frame block are removed and their storage reclaimed when **RiFrameEnd** is called (thus invalidating their handles).

All of the information that changes from frame to frame should be inside a frame block. In this way, all of the information that is necessary to produce a single frame of an animated sequence may be extracted from a command stream by retaining only those commands within the appropriate frame block and any commands outside all of the frame blocks. This command need not be used if the application is producing a single image.

RIB BINDING

FrameBegin *int*
FrameEnd -

EXAMPLE

```
RiFrameBegin (14);
```

SEE ALSO

RiWorldBegin

RiWorldBegin()
RiWorldEnd()

When **RiWorldBegin** is invoked, all rendering options are frozen and cannot be changed until the picture is finished. The *world-to-camera transformation* is set to the *current transformation* and the *current transformation* is reinitialized to the identity. Inside an **RiWorldBegin**-**RiWorldEnd** block, the *current transformation* is interpreted to be the *object-to-world transformation*. After an **RiWorldBegin**, the interface can accept geometric primitives that define the scene. (The only other mode in which geometric primitives may be defined is inside a **RiObjectBegin**-**RiObjectEnd** block.) Some rendering programs may immediately begin rendering geometric primitives as they are defined, whereas other rendering programs may wait until the entire scene has been defined.

RiWorldEnd does not normally return until the rendering program has completed drawing the image. If the image is to be saved in a file, this is done automatically by **RiWorldEnd**.

All lights and retained objects defined inside the **RiWorldBegin**-**RiWorldEnd** world block are removed and their storage reclaimed when **RiWorldEnd** is called (thus invalidating their handles).

RIB BINDING

WorldBegin -
WorldEnd -

EXAMPLE

```
RiWorldEnd ();
```

SEE ALSO

RiFrameBegin

The following is an example of the use of these procedures, showing how an application constructing an animation might be structured. In the example, an object is defined once and instanced in subsequent frames at different positions.

```
RtObjectHandle BigUglyObject;
RiBegin ();
    BigUglyObject = RiObjectBegin ();
    ...
    RiObjectEnd ();
    /* Display commands */
    RiDisplay (...):
    RiFormat (...);
    RiFrameAspectRatio (1.0);
    RiScreenWindow (...);
    RiFrameBegin (0);
        /* Camera commands */
        RiProjection (RI_PERSPECTIVE,...);
        RiRotate (...);
```

```
        RiWorldBegin ();
            ...
            RiColor (...);
            RiTranslate (...);
            RiObjectInstance ( BigUglyObject );
            ...
        RiWorldEnd ();
    RiFrameEnd ();
    RiFrameBegin (1);
        /* Camera commands */
        RiProjection (RI_PERSPECTIVE,...);
        RiRotate (...);
        RiWorldBegin ();
            ...
            RiColor (...);
            RiTranslate (...);
            RiObjectInstance ( BigUglyObject );
            ...
        RiWorldEnd ();
    RiFrameEnd ();
    ...
RiEnd ();
```

The following begin-end pairs also place the interface into special modes.

```
RiSolidBegin ()
RiSolidEnd ()

RiMotionBegin ()
RiMotionEnd ()

RiObjectBegin ()
RiObjectEnd ()
```

The properties of these modes are described in the appropriate sections (see the sections on *Solids and Spatial Set Operations*, p. 93; *Motion*, p. 97; and *Retained Geometry*, p. 95).

Two other begin-end pairs:

```
RiAttributeBegin ()
RiAttributeEnd ()

RiTransformBegin ()
RiTransformEnd ()
```

save and restore the attributes in the graphics state, and save and restore the current transformation, respectively.

All begin-end pairs (except **RiTransformBegin**-**RiTransformEnd** and **RiMotionBegin**- **RiMotionEnd**), implicitly save and restore attributes. Begin-end blocks of the various types

may be nested to any depth, subject to their individual restrictions, but it is never legal for the blocks to overlap.

4.1 Options

The graphics state has various *options* that must be set before rendering a frame. The complete set of options includes: a description of the camera, which controls all aspects of the imaging process (including the camera position and the type of projection); a description of the display, which controls the output of pixels (including the types of images desired, how they are quantized and which device they are displayed on); as well as renderer run-time controls (such as the hidden surface algorithm to use).

4.1.1 Camera

The graphics state contains a set of parameters that define the properties of the camera. The complete set of camera options is described in Table 4.1, Camera Options.

The viewing transformation specifies the coordinate transformations involved with imaging the scene onto an image plane and sampling that image at integer locations to form a raster of pixel values. A few of these procedures set display parameters such as *resolution* and *pixel aspect ratio*. If the rendering program is designed to output to a particular display device these parameters are initialized in advance. Explicitly setting these makes the specification of an image more device dependent and should only be used if necessary. The defaults given in the *Camera Options* table characterize a hypothetical framebuffer and are the defaults for picture files.

The camera model supports near and far clipping planes that are perpendicular to the viewing direction, as well as any number of arbitrary user-specified clipping planes. Depth of field is specified by setting an f-stop, focal length, and focal distance just as in a real camera. Objects located at the focal distance will be sharp and in focus while other objects will be out of focus. The shutter is specified by giving opening and closing times. Moving objects will blur while the camera shutter is open.

The imaging transformation proceeds in several stages. Geometric primitives are specified in the object coordinate system. This canonical coordinate system is the one in which the object is most naturally described. The object coordinates are converted to the world coordinate system by a sequence of *modeling transformations*. The world coordinate system is converted to the camera coordinate system by the camera transformation. Once in camera coordinates, points are projected onto the image plane or screen coordinate system by the projection and its following screen transformation. Points on the screen are finally mapped to a device dependent, integer coordinate system in which the image is sampled. This is referred to as the raster coordinate system and this transformation is referred to as the raster transformation. These various coordinate systems are summarized in Table 4.2 Point Coordinate Systems.

These various coordinate systems are established by camera and transformation commands. The order in which camera parameters are set is the opposite of the order in which the

Camera Option	Type	Default	Description
Horizontal Resolution	*integer*	640*	The horizontal resolution in the output image.
Vertical Resolution	*integer*	480*	The vertical resolution in the output image.
Pixel Aspect Ratio	*float*	1.0*	The ratio of the width to the height of a single pixel.
Crop Window	4 *floats*	(0,1,0,1)	The region of the raster that is actually rendered.
Frame Aspect Ratio	*float*	4/3*	The aspect ratio of the desired image.
Screen Window	4 *floats*	(-4/3,4/3,-1,1)*	The screen coordinates (coordinates after the projection) of the area to be rendered.
Camera Projection	*token*	"orthographic"	The camera to screen projection.
World to Camera	*transform*	identity	The world to camera transformation.
Near and Far Clipping	2 *floats*	(epsilon,infinity)	The positions of the near and far clipping planes.
Other Clipping Planes	list of planes	–	Additional planes that clip geometry from the scene.
f-Stop	*float*	infinity	Parameters controlling depth of field.
Focal Length	*float*	–	
Focal Distance	*float*	–	
Shutter Open	*float*	0	The times when the shutter opens and closes.
Shutter Close	*float*	0	

* Interrelated defaults

Table 4.1: Camera Options

Coordiante System	Description
"object"	The coordinate system in which the current geometric primitive is defined. The modeling transformation converts from object coordinates to world coordinates.
"world"	The standard reference coordinate system. The camera transformation converts from world coordinates to camera coordinates.
"camera"	A coordinate system with the vantage point at the origin and the direction of view along the positive z-axis. The projection and screen transformation convert from camera coordinates to screen coordinates.
"screen"	The 2-D normalized coordinate system corresponding to the image plane. The raster transformation converts to raster coordinates.
"raster"	The raster or pixel coordinate system. An area of 1 in this coordinate system corresponds to the area of a single pixel. This coordinate system is either inherited from the display or set by selecting the resolution of the image desired.
"NDC"	Normalized device coordinates — like "raster" space, but normalized so that x and y both run from 0 to 1 across the whole (uncropped) 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).

Table 4.2: Point Coordinate Systems

imaging process was described above. When **RiBegin** is executed it establishes a complete set of defaults. If the rendering program is designed to produce pictures for a particular piece of hardware, display parameters associated with that piece of hardware are used. If the rendering program is designed to produce picture files, the parameters are set to generate a video-size image. If these are not sufficient, the resolution and pixel aspect ratio can be set to generate a picture for any display device. **RiBegin** also establishes default screen and camera coordinate systems as well. The default projection is orthographic and the screen coordinates assigned to the display are roughly between ± 1.0. The initial camera coordinate system is mapped onto the display such that the +x axis points right, the $+y$ axis points up, and the $+z$ axis points inward, perpendicular to the display surface. Note that this is left-handed.

Before any transformation commands are made, the *current transformation matrix* contains the identity matrix as the screen transformation. Usually the first transformation command is an **RiProjection**, which appends the projection matrix onto the screen transformation, saves it, and reinitializes the *current transformation matrix* as the identity camera transformation. This marks the current coordinate system as the camera coordinate system. After the camera coordinate system is established, future transformations move the world coordinate system relative to the camera coordinate system. When an **RiWorldBegin** is executed, the *current transformation matrix* is saved as the camera transformation, and thus the world coordinate system is established. Subsequent transformations inside of an **RiWorldBegin**-**RiWorldEnd** establish different object coordinate systems.

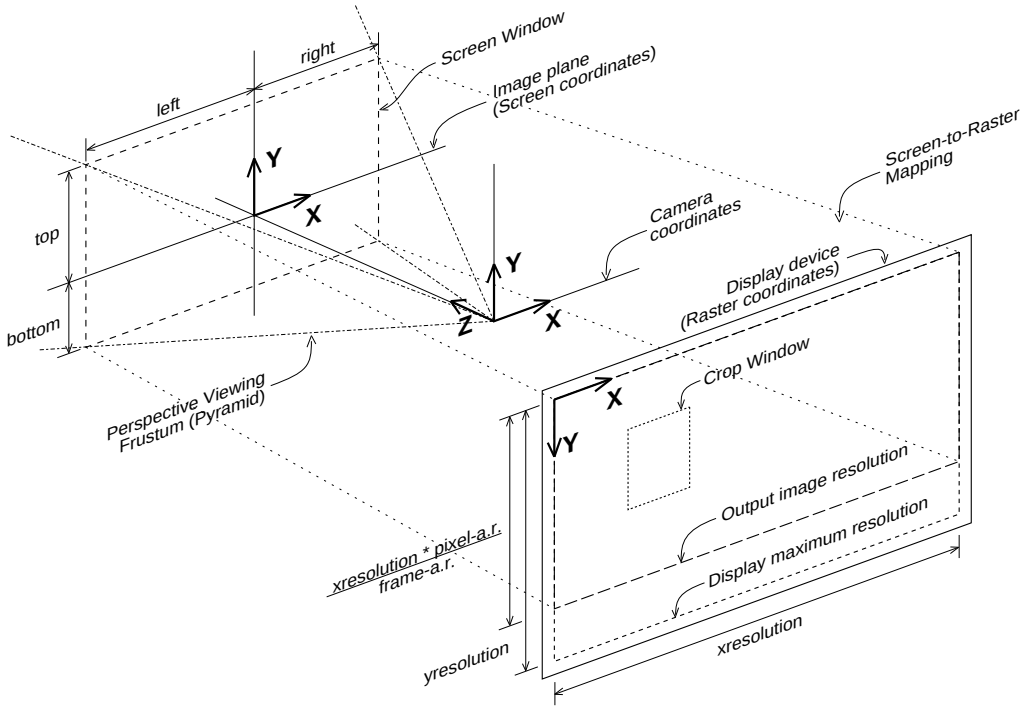


Figure 4.1: Camera-to-Raster Projection Geometry

The following example shows how to position a camera:

```
RiBegin ();
    RiFormat ( xres, yres, 1.0 );          /* Raster coordinate system */
    RiFrameAspectRatio ( 4.0/3.0 );        /* Screen coordinate system */
    RiFrameBegin (0);
        RiProjection ("perspective",...);  /* Camera coordinate system */
        RiRotate (... );
        RiWorldBegin ();                   /* World coordinate system */
            ...
            RiTransform (...);             /* Object coordinate system */
        RiWorldEnd ();
RiFrameEnd ();
RiEnd ();
```

The various camera procedures are described below, with some of the concepts illustrated in Figure 4.1, Camera-to-Raster Projection Geometry.

RiFormat (**RtInt** xresolution, **RtInt** yresolution, **RtFloat** pixelaspectratio)

Set the horizontal (*xresolution*) and vertical (*yresolution*) resolution (in pixels) of the image to be rendered. The upper left hand corner of the image has coordinates (0,0) and the lower right hand corner of the image has coordinates (*xresolution*, *yresolution*). If the resolution is greater than the maximum resolution of the device, the desired image is clipped to the device boundaries (rather than being shrunk to fit inside the device). This command also sets the pixel aspect ratio. The pixel aspect ratio is the ratio of the physical width to the height of a single pixel. The pixel aspect ratio should normally be set to 1 unless a picture is being computed specifically for a display device with non-square pixels.

Implicit in this command is the creation of a display viewport with a

$$\text{viewportaspectratio} = \frac{\text{xresolution} \cdot \text{pixelaspectratio}}{\text{yresolution}}$$

The viewport aspect ratio is the ratio of the physical width to the height of the entire image.

An image of the desired aspect ratio can be specified in a device independent way using the procedure **RiFrameAspectRatio** described below. The **RiFormat** command should only be used when an image of a specified resolution is needed or an image file is being created.

If this command is not given, the resolution defaults to that of the display device being used (see the Displays section, p. 27). Also, if xresolution, yresolution or pixelaspectratio is specified as a nonpositive value, the resolution defaults to that of the display device for that particular parameter.

RIB BINDING

Format *xresolution yresolution pixelaspectratio*

EXAMPLE

Format 512 512 1

SEE ALSO

RiDisplay, **RiFrameAspectRatio**

RiFrameAspectRatio (**RtFloat** frameaspectratio)

frameaspectratio is the ratio of the width to the height of the desired image. The picture produced is adjusted in size so that it fits into the display area specified with **RiDisplay** or **RiFormat** with the specified frame aspect ratio and is such that the upper left corner is aligned with the upper left corner of the display.

If this procedure is not called, the frame aspect ratio defaults to that determined from the resolution and pixel aspect ratio.

RIB BINDING

FrameAspectRatio *frameaspectratio*

EXAMPLE

RiFrameAspectRatio (4.0/3.0);

SEE ALSO

RiDisplay, **RiFormat**

RiScreenWindow (**RtFloat** left, **RtFloat** right, **RtFloat** bottom, **RtFloat** top)

This procedure defines a rectangle in the image plane that gets mapped to the *raster coordinate system* and that corresponds to the display area selected. The rectangle specified is in the *screen coordinate system*. The values *left*, *right*, *bottom*, and *top* are mapped to the respective edges of the display.

The default values for the screen window coordinates are:

(–*frameaspectratio*, *frameaspectratio*, –1, 1).

if *frameaspectratio* is greater than or equal to one, or

(–1, 1, –1/*frameaspectratio*, 1/*frameaspectratio*).

if *frameaspectratio* is less than or equal to one. For perspective projections, this default gives a centered image with the smaller of the horizontal and vertical fields of view equal to the field of view specified with **RiProjection**. Note that if the camera transformation preserves relative x and y distances, and if the ratio

$$\frac{\text{abs}(right - left)}{\text{abs}(top - bottom)}$$

is not the same as the frame aspect ratio of the display area, the displayed image will be distorted.

RIB BINDING

ScreenWindow *left right bottom top*
ScreenWindow [*left right bottom top*]

EXAMPLE

ScreenWindow -1 1 -1 1

SEE ALSO

RiCropWindow, **RiFormat**, **RiFrameAspectRatio**, **RiProjection**

RiCropWindow (**RtFloat** xmin, **RtFloat** xmax, **RtFloat** ymin, **RtFloat** ymax)

Render only a subrectangle of the image. This command does not affect the mapping from screen to raster coordinates. This command is used to facilitate debugging regions of an image, and to help in generating panels of a larger image. These values are specified as fractions of the raster window defined by **RiFormat** and **RiFrameAspectRatio**, and therefore lie between 0 and 1. By default the entire raster window is rendered. The integer image locations corresponding to these limits are given by

```
rxmin = clamp (ceil ( xresolution*xmin ), 0, xresolution-1);
rxmax = clamp (ceil ( xresolution*xmax -1 ), 0, xresolution-1);
rymin = clamp (ceil ( yresolution*ymin ), 0, yresolution-1);
rymax = clamp (ceil ( yresolution*ymax -1 ), 0, yresolution-1);
```

These regions are defined so that if a large image is generated with tiles of abutting but non-overlapping crop windows, the subimages produced will tile the display with abutting and non-overlapping regions.

RIB BINDING

Cropwindow *xmin xmax ymin ymax*
Cropwindow [*xmin xmax ymin ymax*]

EXAMPLE

RiCropWindow (0.0, 0.3, 0.0, 0.5);

SEE ALSO

RiFrameAspectRatio, **RiFormat**

RiProjection (**RtToken** name, *...parameterlist...*)

The projection determines how camera coordinates are converted to screen coordinates, using the type of projection and the near/far clipping planes to generate a projection matrix. It appends this projection matrix to the *current transformation matrix* and stores this as the screen transformation, then marks the current coordinate system as the camera coordinate system and reinitializes the *current transformation matrix* to the identity camera transformation. The required types of projection are "perspective", "orthographic", and RI_NULL.

"perspective" builds a projection matrix that does a perspective projection along the z-axis, using the **RiClipping** values, so that points on the near clipping plane project to $z = 0$ and points on the far clipping plane project to $z = 1$. "perspective" takes one optional parameter, "fov", a single **RtFloat** that indicates he full angle perspective field of view (in degrees) between screen space coordinates (-1,0) and (1,0) (equivalently between (0,-1) and (0,1)). The default is 90 degrees.

Note that there is a redundancy in the focal length implied by this procedure and the one set by **RiDepthOfField**. The focal length implied by this command is:

$$focallength = \frac{horizontalscreenwidth}{verticalscreenwidth} / \tan\left(\frac{fov}{2}\right)$$

"orthographic" builds a simple orthographic projection that scales z using the **RiClipping** values as above. "orthographic" takes no parameters.

RI_NULL uses an identity projection matrix, and simply marks camera space in situations where the user has generated his own projection matrices himself using **RiPerspective** or **RiTransform**.

This command can also be used to select implementation-specific projections or special projections written in the Shading Language. If a particular implementation does

not support the special projection specified, it is ignored and an orthographic projection is used. If **RiProjection** is not called, the screen transformation defaults to the identity matrix, so screen space and camera space are identical.

RIB BINDING

Projection "perspective" *...parameterlist...*
Projection "orthographic"
Projection *name ...parameterlist...*

EXAMPLE

```
RiProjection (RI_ORTHOGRAPHIC, RI_NULL);

RtFloat fov = 45.0;
RiProjection (RI_PERSPECTIVE, "fov", &fov, RI_NULL);
```

SEE ALSO

RiPerspective, **RiClipping**

RiClipping (**RtFloat** near, **RtFloat** far)

Sets the position of the near and far clipping planes along the direction of view. *near* and *far* must both be positive numbers. *near* must be greater than or equal to RI_EPSILON and less than *far*. *far* must be greater than *near* and may be equal to RI_INFINITY. These values are used by **RiProjection** to generate a screen projection such that depth values are scaled to equal zero at z=*near* and one at z=*far*. Notice that the rendering system will actually clip geometry which lies outside of z=(0,1) in the screen coordinate system, so non-identity screen transforms may affect which objects are actually clipped.

For reasons of efficiency, it is generally a good idea to bound the scene tightly with the near and far clipping planes.

RIB BINDING

Clipping *near far*

EXAMPLE

Clipping 0.1 10000

SEE ALSO

RiBound, **RiProjection**, **RiClippingPlane**

RiClippingPlane (**RtFloat** x, **RtFloat** y, **RtFloat** z, **RtFloat** nx, **RtFloat** ny, **RtFloat** nz)

Adds a user-specified clipping plane. The plane is specified by giving any point on its surface, (*x*, *y*, *z*), and the plane normal, (*nx*, *ny*, *nz*). All geometry on the positive side of the plane (that is, in the direction that the normal points) will be clipped from the scene. The point and normal parameters are interpreted as being in the active local coordinate system at the time that the **RiClippingPlane** statement is issued.

Multiple calls to **RiClippingPlane** will establish multiple clipping planes.

RIB BINDING

ClippingPlane *x y z nx ny nz*

EXAMPLE

ClippingPlane 3 0 0 0 0 -1

SEE ALSO

RiClipping

RiDepthOfField (**RtFloat** fstop, **RtFloat** focallength, **RtFloat** focaldistance)

focaldistance sets the distance along the direction of view at which objects will be in focus. *focallength* sets the focal length of the camera. These two parameters should have the units of distance along the view direction in camera coordinates. *fstop*, or aperture number, determines the lens diameter:

$$lensdiameter = \frac{focallength}{fstop}$$

If *fstop* is RI_INFINITY, a pin-hole camera is used and depth of field is effectively turned off. If the *Depth of Field* capability is not supported by a particular implementation, a pin-hole camera model is always used.

If depth of field is turned on, points at a particular depth will not image to a single point on the view plane but rather a circle. This circle is called the *circle of confusion*. The diameter of this circle is equal to

$$C = \frac{focallength}{fstop} \cdot \frac{focaldistance \cdot focallength}{focaldistance - focallength} \cdot \left| \frac{1}{depth} - \frac{1}{focaldistance} \right|$$

Note that there is a redundancy in the focal length as specified in this procedure and the one implied by **RiProjection**.

RIB BINDING

DepthOfField *fstop focallength focaldistance*
DepthOfField -

The second form specifies a pin-hole camera with infinite *fstop*, for which the *focallength* and *focaldistance* parameters are meaningless.

EXAMPLE

DepthOfField 22 45 1200

SEE ALSO

RiProjection

RiShutter (**RtFloat** min, **RtFloat** max)

This procedure sets the times at which the shutter opens and closes. *min* should be less than *max*. If *min*==*max*, no motion blur is done.

RIB BINDING

Shutter *min max*

EXAMPLE

RiShutter (0.1, 0.9);

SEE ALSO

RiMotionBegin

4.1.2 Displays

The graphics state contains a set of parameters that control the properties of the display process. The complete set of display options is given in Table 4.3, Display Options.

Rendering programs must be able to produce color, coverage (alpha), and depth images, and may optionally be able to produce "images" of arbitrary geometric or shader-computed data. Display parameters control how the values in these images are converted into a displayable form. Many times it is possible to use none of the procedures described in this section. If this is done, the rendering process and the images it produces are described in a completely device-independent way. If a rendering program is designed for a specific display, it has appropriate defaults for all display parameters. The defaults given in Table 4.3, Display Options characterize a file to be displayed on a hypothetical video framebuffer.

The output process is different for color, alpha, and depth information. (See Figure 4.2, Imaging Pipeline). The hidden-surface algorithm will produce a representation of the light incident on the image plane. This color image is either continuous or sampled at a rate that may be higher than the resolution of the final image. The minimum sampling rate can be controlled directly, or can be indicated by the estimated variance of the pixel values. These color values are filtered with a user-selectable filter and filterwidth, and sampled at the pixel centers. The resulting color values are then multiplied by the gain and passed through an inverse gamma function to simulate the exposure process. The resulting colors are then passed to a quantizer which scales the values and optionally dithers them before converting them to a fixed-point integer. It is also possible to interpose a programmable imager (written in the Shading Language) between the exposure process and quantizer. This imager can be used to perform special effects processing, to compensate for nonlinearities in the display media, and to convert to device dependent color spaces (such as CMYK or pseudocolor).

Final output alpha is computed by multiplying the coverage of the pixel (i.e., the subpixel area actually covered by a geometric primitive) by the average of the color opacity components. If an alpha image is being output, the color values will be multiplied by this alpha before being passed to the quantizer. Color and alpha use the same quantizer.

Display Option	Type	Default	Description
Pixel Variance	*float*	–	Estimated variance of the computed pixel value from the true pixel value.
Sampling Rates	2 *floats*	2, 2	Effective sampling rate in the horizontal and vertical directions.
Filter	*function*	RiGaussianFilter	Type of filtering and the width of
Filter Widths	2 *floats*	2, 2	the filter in the horizontal and vertical directions.
Exposure			
gain	*float*	1.0	Gain and gamma of the exposure
gamma	*float*	1.0	process.
Imager	*shader*	"null"	A procedure defining an image or pixel operator.
Color Quantizer			Color and opacity quantization
one	*integer*	255	parameters.
maximum	*integer*	0	
minimum	*integer*	255	
dither amplitude	*float*	0.5	
Depth Quantizer			Depth quantization parameters.
one	*integer*	0	
maximum	*integer*	–	
minimum	*integer*	–	
dither amplitude	*float*	–	
Display Type	*token*	*	Whether the display is a framebuffer or a file.
Display Name	*string*	*	Name of the display device or file.
Display Mode	*token*	*	Image output type.

* Implementation-specific

Table 4.3: Display Options

Output depth values are the camera-space z values. Depth values bypass all the above steps except for the imager and quantization. The depth quantizer has an independent set of parameters from those of the color quantizer.

RiPixelVariance (**RtFloat** variation)

The color of a pixel computed by the rendering program is an estimate of the true pixel value: the convolution of the continuous image with the filter specified by **RiPixelFilter**. This routine sets the upper bound on the acceptable estimated variance of the pixel values from the true pixel values.

RIB BINDING

PixelVariance *variation*

EXAMPLE

RiPixelVariance (.01);

SEE ALSO

RiPixelFilter, **RiPixelSamples**

RiPixelSamples (**RtFloat** xsamples, **RtFloat** ysamples)

Set the effective hider sampling rate in the horizontal and vertical directions. The effective number of samples per pixel is *xsamples***ysamples*. If an analytic hidden surface calculation is being done, the effective sampling rate is RI_INFINITY. Sampling rates less than 1 are clamped to 1.

RIB BINDING

PixelSamples *xsamples ysamples*

EXAMPLE

PixelSamples 2 2

SEE ALSO

RiPixelFilter, **RiPixelVariance**

RiPixelFilter (**RtFilterFunc** filterfunc, **RtFloat** xwidth, **RtFloat** ywidth)

Antialiasing is performed by filtering the geometry (or supersampling) and then sampling at pixel locations. The *filterfunc* controls the type of filter, while *xwidth* and *ywidth* specify the width of the filter in pixels. A value of 1 indicates that the support of the filter is one pixel. RenderMan supports nonrecursive, linear shift-invariant filters. The type of the filter is set by passing a reference to a function that returns a filter kernel value; i.e.,

*filterkernelvalue = (*filterfunc)(x, y, xwidth, ywidth);*

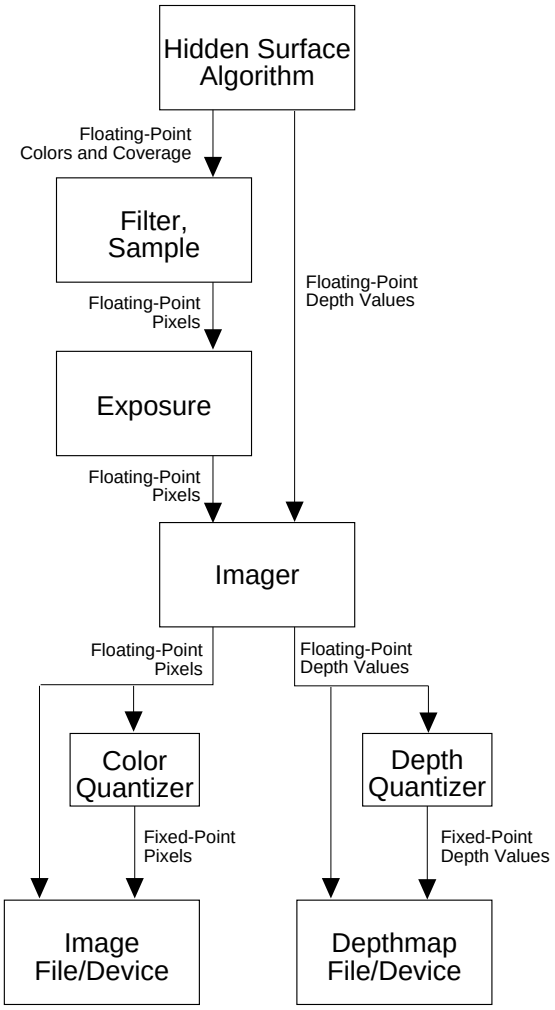


Figure 4.2: Imaging pipeline

(where (*x*,*y*) is the point at which the filter should be evaluated). The rendering program only requests values in the ranges –*xwidth*/2 to *xwidth*/2 and –*ywidth*/2 to *ywidth*/2. The values returned need not be normalized.

The following standard filter functions are available:

RtFloat RiBoxFilter (**RtFloat**, **RtFloat**, **RtFloat**, **RtFloat**);
RtFloat RiTriangleFilter (**RtFloat**, **RtFloat**, **RtFloat**, **RtFloat**);
RtFloat RiCatmullRomFilter (**RtFloat**, **RtFloat**, **RtFloat**, **RtFloat**);
RtFloat RiGaussianFilter (**RtFloat**, **RtFloat**, **RtFloat**, **RtFloat**);
RtFloat RiSincFilter (**RtFloat**, **RtFloat**, **RtFloat**, **RtFloat**);

A particular renderer implementation may also choose to provide additional built-in filters. The standard filters are described in Appendix E.

A high-resolution picture is often computed in sections or panels. Each panel is a subrectangle of the final image. It is important that separately computed panels join together without a visible discontinuity or seam. If the filter width is greater than 1 pixel, the rendering program must compute samples outside the visible window to properly filter before sampling.

RIB BINDING

PixelFilter *type xwidth ywidth*

The *type* is one of: "box", "triangle", "catmull-rom" (cubic), "sinc", and "gaussian".

EXAMPLE

```
RiPixelFilter (RiGaussianFilter, 2.0, 1.0);
PixelFilter "gaussian" 2 1
```

SEE ALSO

RiPixelSamples, **RiPixelVariance**

RiExposure (**RtFloat** gain, **RtFloat** gamma)

This function controls the sensitivity and nonlinearity of the exposure process. Each component of color is passed through the following function:

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

RIB BINDING

Exposure *gain gamma*

EXAMPLE

Exposure 1.5 2.3

SEE ALSO

RiImager

RiImager (**RtToken** name, *...parameterlist...*)

Select an imager function programmed in the Shading Language. *name* is the name of an *imager shader*. If *name* is RI_NULL, no *imager shader* is used.

RIB BINDING

Imager *name ...parameterlist...*

EXAMPLE

RiImager ("cmyk," RI_NULL);

SEE ALSO

RiExposure

RiQuantize (**RtToken** type, **RtInt** one, **RtInt** min, **RtInt** max, **RtFloat** ditheramplitude)

Set the quantization parameters for colors *or* depth. If *type* is "rgba", then color and opacity quantization are set. If *type* is "z", then depth quantization is set. The value *one* defines the mapping from floating-point values to fixed point values. If *one* is 0, then quantization is not done and values are output as floating point numbers.

Dithering is performed by adding a random number to the floating-point values before they are rounded to the nearest integer. The added value is scaled to lie between plus and minus the dither amplitude. If *ditheramplitude* is 0, dithering is turned off.

Quantized values are computed using the following formula:

value = *round*(*one* * *value* + *ditheramplitude* * *random*());
value = *clamp*(*value*, *min*, *max*);

where *random* returns a random number between ± 1.0, and *clamp* clips its first argument so that it lies between *min* and *max*.

By default color pixel values are dithered with an amplitude of 0.5 and quantization is performed for an 8-bit display with a *one* of 255. Quantization and dithering and not performed for depth values (by default).

RIB BINDING

Quantize *type one min max ditheramplitude*

EXAMPLE

RiQuantize (RI_RGBA, 2048, -1024, 3071, 1.0);

SEE ALSO

RiDisplay, **RiImager**

RiDisplay (**RtToken** name, **RtToken** type, **RtToken** mode, *...parameterlist...*)

Choose a display by name and set the type of output being generated. *name* is either the name of a picture file or the name of the framebuffer, depending on *type*. The *type* of display is the display format, output device, or output driver. All implementations must support the type names "framebuffer" and "file", which indicate that the renderer should select the default framebuffer or default file format, respectively. Implementations may support any number of particular formats or devices (for example, "tiff" might indicate that a TIFF file should be written), and may allow the supported formats to be user-extensible in an implementation-specific manner.

The *mode* indicates what data are to be output in this display stream. All renderers must support any combination (string concatenation) of "rgb" for color (usually red, green and blue intensities unless there are more or less than 3 color samples; see the next section, Additional options), "a" for alpha, and "z" for depth values, in that order.

Renderers may additionally produce "images" consisting of arbitrary data, by using a *mode* that is the name of a known geometric quantity or the name of a shader output variable. Note also that multiple display channels can be specified, by prepending the + character to the *name*. For example,

RiDisplay ("out.tif," "file," "rgba", RI_NULL);
RiDisplay ("+normal.tif," "file," "N", RI_NULL);

will produce a four-channel image consisting of the filtered color and alpha in out.tif, and also a second three-channel image file normal.tif consisting of the surface normal of the nearest surface behind each pixel. (This would, of course, only be useful if **RiQuantize** were instructed to output floating point data or otherwise scale the data.)

Display options or device-dependent display modes or functions may be set using the *parameterlist*. One such option is required: "origin", which takes an array of two **RtInt**s, sets the x and y position of the upper left hand corner of the image in the display's coordinate system; by default the origin is set to (0,0). The default display device is renderer implementation-specific.

RIB BINDING

Display *name type mode ...parameterlist...*

EXAMPLE

```
RtInt origin[2] = { 10, 10 };
RiDisplay ("pixar0", "framebuffer", "rgba", "origin", (RtPointer)origin, RI_NULL);
```

SEE ALSO

RiFormat, **RiQuantize**

4.1.3 Additional options

Option	Type	Default	Description
Hider	*token*	"hidden"	The type of hidden surface algorithm that is performed.
Color Samples	*integer*	3	Number of color components in colors. The default is 3 for RGB.
Relative Detail	*float*	1.0	A multiplicative factor that can be used to increase or decrease the effective level of detail used to render an object.

Table 4.4: Additional Options

The hider type and parameters control the hidden-surface algorithm.

RiHider (**RtToken** type, *...parameterlist...*)

The standard types are "hidden" and "null". "hidden", which is the default, performs standard hidden-surface computations. The hider "null" performs no pixel computation and hence produces no output. Other implementation-specific hidden-surface algorithms can also be selected using this routine; for example, an implementation may choose to support *type* "paint", which might draw the objects in the order in which they are defined.

RIB BINDING

Hider *type ...parameterlist...*

EXAMPLE

RiHider "paint"

Rendering programs compute color values in some *spectral color space*. This implies that multiplying two colors corresponds to interpreting one of the colors as a light and the other as a filter and passing light through the filter. Adding two colors corresponds to adding two lights. The default color space is NTSC-standard RGB; this color space has three samples. Color values of 0 are interpreted as black (or transparent) and values of 1 are interpreted as white (or opaque), although values outside this range are allowed.

RiColorSamples (**RtInt** n, **RtFloat** nRGB[], **RtFloat** RGBn[])

This function controls the number of color components or samples to be used in specifying colors. By default, *n* is 3, which is appropriate for RGB color values. Setting *n* to 1 forces the rendering program to use only a single color component. The array *nRGB* is an *n* by 3 transformation matrix that is used to convert *n* component colors to 3 component NTSC-standard RGB colors. This is needed if the rendering program cannot handle multiple components. The array *RGBn* is a 3 by *n* transformation matrix that is used to convert 3 component NTSC-standard RGB colors to *n* component colors. This is mainly used for transforming constant colors specified as color triples in the Shading Language to the representation being used by the RenderMan Interface.

Calling this procedure effectively redefines the type **RtColor** to be

typedef **RtFloat RtColor** [n];

After a call to **RiColorSamples**, all subsequent color arguments are assumed to be this size.

If the *Spectral Color* capability is not supported by a particular implementation, that implementation will still accept multiple component colors, but will immediately convert them to RGB color space and do all internal calculations with 3 component colors.

RIB BINDING

ColorSamples *nRGB RGBn*

The number of color components, *n*, is derived from the lengths of the *nRGB* and *RGBn* arrays, as described above.

EXAMPLE

```
ColorSamples [.3.3 .4] [1 1 1]
RtFloat frommonochr[] = {.3, .3, .4};
RtFloat tomonochr[] = {1., 1., 1.};
RiColorSamples (1, frommonochr, tomonochr);
```

SEE ALSO

RiColor, **RiOpacity**

The method of specifying and using level of detail is discussed in the section on Detail.

RiRelativeDetail (**RtFloat** relativedetail)

The relative level of detail scales the results of all level of detail calculations. The level of detail is used to select between different representations of an object. If *relativedetail* is greater than 1, the effective level of detail is increased, and a more detailed representation of all objects will be drawn. If *relativedetail* is less than 1, the effective level of detail is decreased, and a less detailed representation of all objects will be drawn.

RIB BINDING

RelativeDetail *relativedetail*

EXAMPLE

```
RelativeDetail 0.6
```

SEE ALSO

RiDetail, **RiDetailRange**

4.1.4 Implementation-specific options

Rendering programs may have additional implementation-specific options that control parameters that affect either their performance or operation. These are all set by the following procedure.

RiOption (**RtToken** name, *...parameterlist...*)

Sets the named implementation-specific option. A rendering system may have certain options that must be set before the renderer is initialized. In this case, **RiOption** may be called before **RiBegin** to set those options only.

Although **RiOption** is intended to allow implementation-specific options, there are a number of options that we expect that nearly all implementations will need to support. It is intended that when identical functionality is required, that all implementations use the option names listed in Table 4.5.

RIB BINDING

Option *name ...parameterlist...*

EXAMPLE

Option "limits" "gridsize" [32] "bucketsize" [12 12]

SEE ALSO

RiAttribute

Option name/param	Type	Default	Description
"searchpath" "archive" [s]	*string*	""	List of directories to search for RIB archives.
"searchpath" "texture" [s]	*string*	""	List of directories to search for texture files.
"searchpath" "shader" [s]	*string*	""	List of directories to search for shaders.
"searchpath" "procedural" [s]	*string*	""	List of directories to search for dynamically-loaded **RiProcedural** primitives.
"statistics" "endofframe" [i]	*integer*	""	If nonzero, print runtime statistics when the frame is finished rendering.

Table 4.5: Typical implementation-specific options

4.2 Attributes

Attributes are parameters in the graphics state that may change while geometric primitives are being defined. The complete set of standard attributes is described in two tables: Table 4.6, Shading Attributes, and Table 4.11, Geometry Attributes.

Attributes can be explicitly saved and restored with the following commands. All begin-end blocks implicitly do a save and restore.

RiAttributeBegin ()

RiAttributeEnd ()

Push and pop the current set of attributes. Pushing attributes also pushes the current transformation. Pushing and popping of attributes must be properly nested with respect to various begin-end constructs.

RIB BINDING

AttributeBegin -
AttributeEnd -

EXAMPLE

```
RiAttributeBegin ();
```

SEE ALSO

RiFrameBegin, **RiTransformBegin**, **RiWorldBegin**

The process of shading is described is detail in Part II: The RenderMan Shading Language. The complete list of attributes related to shading are in Table 4.6, Shading Attributes.

The graphics state maintains a list of attributes related to shading. Associated with the shading state are a *current color* and a *current opacity*. The graphics state also contains a *current surface shader*, a *current atmosphere shader*, a *current interior volume shader*, and a *current exterior volume shader*.

All geometric primitives use the *current surface shader* for computing the color (shading) of their surfaces and the *current atmosphere shader* for computing the attenuation of light towards the viewer. Primitives may also attach the *current interior* and *exterior volume shaders* to their interior and exterior, which is used to alter the colors of rays spawned by `trace()` calls in the shaders bound to the primitives (in a renderer that supports this optional feature). The graphics state also contains a *current list of light sources* that are used to illuminate the geometric primitive. Finally, there is a *current area light source*. Geometric primitives can be added to a list of primitives defining this light source.

4.2.1 Color and opacity

All geometric primitives inherit the current color and opacity from the graphics state, unless color or opacity are defined as part of the primitive. Colors are passed in arrays that are assumed to contain the number of color samples being used (see the section on Additional options).

RiColor (**RtColor** color)

Set the current color to color. Normally there are three components in the color (*red, green*, and *blue*), but this may be changed with the colorsamples request.

Shading Attribute	Type	Default	Description
Color	*color*	color ″rgb″ (1,1,1)	The reflective color of the object.
Opacity	*color*	color ″rgb″ (1,1,1)	The opacity of the object.
Texture coordinates	8 *floats*	(0,0)(1,0),(0,1),(1,1)	The texture coordinates (s, t) at the 4 corners of a parametric primitive.
Light Sources	*shader list*	–	A list of light source shaders that illuminate subsequent primitives.
Area Light Source	*shader*	–	An area light source which is being defined.
Surface	*shader*	default surface*	A shader controlling the surface shading model.
Atmosphere	*shader*	–	A volume shader that specifies how the color of light is changed as it travels from a visible surface to the eye.
Interior Volume	*shader*	–	
Exterior Volume	*shader*	–	A volume shader that specifies how the color of light is changed as it traverses a volume in space.
Effective Shading Rate	*float*	1	Minimum rate of surface shading.
Shading Interpolation	*token*	″constant″	How the results of shading samples are interpolated.
Matte Surface Flag	*boolean*	false	A flag indicating the surfaces of the subsequent primitives are opaque to the rendering program, but transparent on output.

* Implementation-dependent

Table 4.6: Shading Attributes

RIB BINDING

Color *c0 c1... cn*
Color [*c0 c1... cn*]

EXAMPLE

RtColor blue = { .2, .3, .9};
RiColor (blue);

Color [.2 .3 .9]

SEE ALSO

RiOpacity, **RiColorSamples**

RiOpacity (**RtColor** color)

Set the current opacity to *color*. The color component values must be in the range [0,1]. Normally there are three components in the color *(red, green*, and *blue*), but this may be changed with **RiColorSamples**. If the opacity is 1, the object is completely opaque; if the opacity is 0, the object is completely transparent.

RIB BINDING

Opacity *c0 c1... cn*
Opacity [*c0 c1... cn*]

EXAMPLE

Opacity 0.5 1 1

SEE ALSO

RiColorSamples, **RiColor**

4.2.2 Texture coordinates

The Shading Language allows precalculated images to be accessed by a set of two-dimensional texture coordinates. This general process is referred to as *texture mapping*. Texture access in the Shading Language is very general since the coordinates are allowed to be any legal expression. However, the texture access functions (in Part II, see the sections on Basic texture maps) often use default texture coordinates related to the surface parameters.

All the parametric geometric primitives have surface parameters (u,v) that can be used as their texture coordinates (s,t). Surface parameters for different primitives are normally defined to lie in the range 0 to 1. This defines a unit square in parameter space. Section 5, Geometric Primitives defines the position on each surface primitive that the corners of this unit square lie. The texture coordinates at each corner of this unit square are given by providing a corresponding set of (s,t) values. This correspondence uniquely defines a 3x3

homogeneous two-dimensional mapping from parameter space to texture space. Special cases of this mapping occur when the transformation reduces to a scale and an offset, which is often used to piece patches together, or to an affine transformation, which is used to map a collection of triangles onto a common planar texture.

The graphics state maintains a *current set of texture coordinates*. The correspondence between these texture coordinates and the corners of the unit square is given by the following table.

Surface Parameters (u,v)	**Texture Coordinates** (s,t)
(0,0)	(s_1, t_1)
(1,0)	(s_2, t_2)
(0,1)	(s_3, t_3)
(1,1)	(s_4, t_4)

By default, the texture coordinates at each corner are the same as the surface parameters (s=u, t=v). Note that texture coordinates can also be explicitly attached to geometric primitives. Note also that polygonal primitives are not parametric, and the current set of texture coordinates do not apply to them.

RiTextureCoordinates (**RtFloat** s1, **RtFloat** t1, **RtFloat** s2, **RtFloat** t2,
RtFloat s3, **RtFloat** t3, **RtFloat** s4, **RtFloat** t4)

Set the current set of texture coordinates to the values passed as arguments according to the above table.

RIB BINDING

TextureCoordinates *s1 t1 s2 t2 s3 t3 s4 t4*
TextureCoordinates [*s1 t1 s2 t2 s3 t3 s4 t4*]

EXAMPLE

```
RiTextureCoordinates (0.0, 0.0, 2.0, -0.5, -0.5, 1.75, 3.0, 3.0);
```

SEE ALSO

texture() and bump() in the Shading Language

4.2.3 Light sources

The graphics state maintains a *current light source list*. The lights in this list illuminate subsequent surfaces. By making this list an attribute different light sources can be used to illuminate different surfaces. Light sources can be added to this list by turning them on and removed from this list by turning them off. Note that popping to a previous graphics state also has the effect of returning the current light list to its previous value. Initially the graphics state does not contain any lights.

An area light source is defined by a shader and a collection of geometric primitives. The association between the shader and the geometric primitives is done by having the graphics state maintain a *single current area light source*. Each time a primitive is defined it is added to the list of primitives that define the *current area light source*. An area light source may be turned on and off just like other light sources.

The RenderMan Interface includes four standard types of light sources: "ambientlight", "pointlight", "distantlight", and "spotlight". The definition of these light sources are given in Appendix A, Standard RenderMan Interface Shaders. The parameters controlling these light sources are given in Table 4.7, Standard Light Source Shader Parameters.

Light Source	Parameter	Type	Default	Description
ambientlight	intensity	*float*	1.0	Light intensity
	lightcolor	*color*	color "rgb" (1,1,1)	Light color
distantlight	intensity	*float*	1.0	Light intensity
	lightcolor	*color*	color "rgb" (1,1,1)	Light color
	from	*point*	point "shader" (0,0,0)	Light position
	to	*point*	point "shader" (0,0,1)	Light direction is from-to
pointlight	intensity	*float*	1.0	Light intensity
	lightcolor	*color*	color "rgb" (1,1,1)	Light color
	from	*point*	point "shader" (0,0,0)	Light position
spotlight	intensity	*float*	1.0	Light intensity
	lightcolor	*color*	color "rgb" (1,1,1)	Light color
	from	*point*	point "shader" (0,0,0)	Light position
	to	*point*	point "shader" (0,0,1)	Light direction is from-to
	coneangle	*float*	radians(30)	Light cone angle
	conedeltaangle	*float*	radians(5)	Light soft edge angle
	beamdistribution	*float*	2.0	Light beam distribution

Table 4.7: Standard Light Source Shader Parameters

RtLightHandle RiLightSource (**RtToken** shadername, *...parameterlist...*)

shadername is the name of a light source shader. This procedure creates a non-area light, turns it on, and adds it to the *current light source list*. An **RtLightHandle** value is returned that can be used to turn the light off or on again.

RIB BINDING

LightSource *name sequencenumber ...parameterlist...*

The sequencenumber is a unique light identification number which is provided by the RIB client to the RIB server. Both client and server maintain independent mappings between the sequencenumber and their corresponding **RtLightHandle**s. The number must be in the range 0 to 65535.

EXAMPLE

LightSource ”spotlight” 2 ”coneangle” [5]
LightSource ”ambientlight” 3 ”lightcolor” [.5 0 0] ”intensity” [.6]

SEE ALSO

RiAreaLightSource, **RiIlluminate**, **RiFrameEnd**, **RiWorldEnd**

RtLightHandle RiAreaLightSource (**RtToken** shadername, *...parameterlist...*)

shadername is the name of a light source shader. This procedure creates an area light and makes it the *current area light source*. Each subsequent geometric primitive is added to the list of surfaces that define the area light. **RiAttributeEnd** ends the assembly of the area light source.

The light is also turned on and added to the *current light source list*. An **RtLightHandle** value is returned which can be used to turn the light off or on again.

If the *Area Light Source* capability is not supported by a particular implementation, this subroutine is equivalent to **RiLightSource**.

RIB BINDING

AreaLightSource *name sequencenumber ...parameterlist...*

The sequencenumber is a unique light identification number which is provided by the RIB client to the RIB server. Both client and server maintain independent mappings between the sequencenumber and their corresponding **RtLightHandle**s. The number must be in the range 0 to 65535.

EXAMPLE

```
RtFloat decay = .5, intensity = .6;
RtColor color = {.5,0,0};
RiAreaLightSource ( ”finite”, ”decayexponent”, (RtPointer)&decay,
                    ”lightcolor”, (RtPointer)color,
                    ”intensity”, (RtPointer)&intensity, RI_NULL);
```

SEE ALSO

RiFrameEnd, **RiLightSource**, **RiIlluminate**, **RiWorldEnd**

RiIlluminate (**RtLightHandle** light, **RtBoolean** onoff)

If *onoff* is RI_TRUE and the light source referred to by the **RtLightHandle** is not currently in the *current light source list*, add it to the list. If *onoff* is RI_FALSE and the light source referred to by the **RtLightHandle** is currently in the *current light source list*, remove it from the list. Note that popping the graphics state restores the onoff value of all lights to their previous values.

RIB BINDING

Illuminate *sequencenumber onoff*

The sequencenumber is the integer light handle defined in a **LightSource** or **AreaLightSource** request.

EXAMPLE

```
LightSource "main" 3
Illuminate 3 0
```

SEE ALSO

RiAttributeEnd, **RiAreaLightSource**, **RiLightSource**

4.2.4 Surface shading

The graphics state maintains a *current surface shader*. The *current surface shader* is used to specify the surface properties of subsequent geometric primitives. Initially the *current surface shader* is set to an implementation-dependent default surface shader (but not "null").

The RenderMan Interface includes six standard types of surfaces: "constant", "matte", "metal", "shinymetal", "plastic", and "paintedplastic". The definitions of these surface shading procedures are given in Appendix A, Standard RenderMan Interface Shaders. The parameters controlling these surfaces are given in Table 4.8, Standard Surface Shader Parameters.

RiSurface (**RtToken** shadername, *...parameterlist...*)

shadername is the name of a surface shader. This procedure sets the *current surface shader* to be shadername. If the surface shader *shadername* is not defined, some implementation-dependent default surface shader (but not "null") is used.

RIB BINDING

Surface *shadername ...parameterlist...*

EXAMPLE

```
RtFloat rough = 0.3, kd = 1.0, width = 0.25;
RiSurface ("wood", "roughness", (RtPointer)&rough,
                "Kd", (RtPointer)&kd, "float ringwidth", &width, RI_NULL);
```

SEE ALSO

RiAtmosphere, **RiDisplacement**

Surface Name	Parameter	Type	Default	Description
constant	–	–	–	–
matte	Ka	*float*	1.0	Ambient coefficient
	Kd	*float*	1.0	Diffuse coefficient
metal	Ka	*float*	1.0	Ambient coefficient
	Ks	*float*	1.0	Specular coefficient
	roughness	*float*	0.1	Surface roughness
shinymetal	Ka	*float*	1.0	Ambient coefficient
	Ks	*float*	1.0	Specular coefficient
	Kr	*float*	1.0	Reflection coefficient
	roughness	*float*	0.1	Surface roughness
	texturename	*string*	""	Environment map name
plastic	Ka	*float*	1.0	Ambient coefficient
	Kd	*float*	0.5	Diffuse coefficient
	Ks	*float*	0.5	Specular coefficient
	roughness	*float*	0.1	Surface roughness
	specularcolor	*color*	color "rgb" (1,1,1)	Specular color
paintedplastic	Ka	*float*	1.0	Ambient coefficient
	Kd	*float*	0.5	Diffuse coefficient
	Ks	*float*	0.5	Specular coefficient
	roughness	*float*	0.1	Surface roughness
	specularcolor	*color*	color "rgb" (1,1,1)	Specular color
	texturename	*string*	""	Texture map name

Table 4.8: Standard Surface Shader Parameters

4.2.5 Displacement shading

The graphics state maintains a *current displacement shader*. Displacement shaders are procedures that can be used to modify geometry before the lighting stage.

The RenderMan Interface includes one standard displacement shader: "bumpy". The definition of this displacement shader is given in Appendix A, Standard RenderMan Interface Shaders. The parameters controlling this displacement is given in Table 4.9.

RiDisplacement (**RtToken** shadername, *...parameterlist...*)

Set the *current displacement shader* to the named shader. *shadername* is the name of a displacement shader.

If a particular implementation does not support the Displacements capability, displacement shaders can only change the normal vectors to generate bump mapping, and the surface geometry itself is not modified (see Displacement Shaders).

RIB BINDING

Displacement *shadername ...parameterlist...*

EXAMPLE

RiDisplacement ("displaceit", RI_NULL);

SEE ALSO

RiDeformation, **RiMakeBump**, **RiSurface**

Surface Name	Parameter	Type	Default	Description
bumpy	amplitude	*float*	1.0	Bump scaling factor
	texturename	*string*	""	Displacement map name

Table 4.9: Standard Displacement Shader Parameters

4.2.6 Volume shading

The graphics state contains a *current interior volume shader*, a *current exterior volume shader*, and a *current atmosphere shader*. These shaders are used to modify the colors of rays traveling through volumes in space.

The interior and exterior shaders define the material properties on the interior and exterior volumes adjacent to the surface of a geometric primitive. The exterior volume relative to a surface is the region into which the natural surface normal points; the interior is the opposite side. Interior and exterior shaders are applied to rays spawned by trace() calls in a surface shader. Renderers that do not support the optional Ray Tracing capability will also not support interior and exterior shaders.

An atmosphere shader is a volume shader that is used to modify rays traveling towards the eye (i.e., camera rays). Even renderers that do not support the optional Ray Tracing capability can still apply atmosphere shaders to any objects directly visible to the camera.

The RenderMan Interface includes two standard volume shaders: "fog" and "depthcue". The definitions of these volume shaders are given in Appendix A, Standard RenderMan Interface Shaders. The parameters controlling these volumes are given in Table 4.10, Standard Volume Shader Parameters.

RiAtmosphere (**RtToken** shadername, *...parameterlist...*)

This procedure sets the *current atmosphere shader*. *shadername* is the name of an atmosphere shader. If *shadername* is RI_NULL, no atmosphere shader is used.

RIB BINDING

Atmosphere *shadername ...parameterlist...*

EXAMPLE

Atmosphere "fog"

SEE ALSO

RiDisplacement, **RiSurface**

Volume Name	Parameter	Type	Default	Description
depthcue	mindistance	*float*	0.0	Distance when brightest
	maxdistance	*float*	1.0	Distance when dimmest
	background	*color*	color "rgb" (0,0,0)	Background color
fog	distance	*float*	1.0	Exponential extinction distance
	background	*color*	color "rgb" (0,0,0)	Background color

Table 4.10: Standard Volume Shader Parameters

RiInterior (**RtToken** shadername, *...parameterlist...*);

This procedure sets the *current interior volume shader*. *shadername* is the name of a volume or atmosphere shader. If *shadername* is RI_NULL, the surface will not have an interior shader.

RIB BINDING

Interior *shadername ...parameterlist...*

EXAMPLE

Interior "water"

SEE ALSO

RiExterior RiAtmosphere

RiExterior (**RtToken** shadername, *...parameterlist...*);

This procedure sets the *current exterior volume shader*. *shadername* is the name of a volume or atmosphere shader. If *shadername* is RI_NULL, the surface will not have an exterior shader.

RIB BINDING

Exterior *shadername ...parameterlist...*

EXAMPLE

RiExterior ("fog," RI_NULL);

SEE ALSO

RiInterior, **RiAtmosphere**

If a particular implementation does not support the *Volume Shading* or *Ray Tracing* capabilities, **RiInterior** and **RiExterior** are ignored; however, **RiAtmosphere** will be available in all implementations.

4.2.7 Shading rate

The number of shading calculations per primitive is controlled by the *current shading rate.* The shading rate is expressed in pixel area. If geometric primitives are being broken down into polygons and each polygon is shaded once, the shading rate is interpreted as the maximum size of a polygon in pixels. A rendering program will shade *at least at this rate*, although it may shade more often. Whatever the value of the shading rate, at least one shading calculation is done per primitive.

RiShadingRate (**RtFloat** size)

Set the *current shading rate* to size. The *current shading rate* is specified as an area in pixels. A shading rate of RI_INFINITY specifies that shading need only be done once per primitive. A shading rate of 1 specifies that shading is done at least once per pixel. This second case is often referred to as *Phong shading*.

RIB BINDING

ShadingRate *size*

EXAMPLE

RiShadingRate (1.0);

SEE ALSO

RiGeometricApproximation

4.2.8 Shading interpolation

Shading calculations are performed at discrete positions on surface elements or in screen space (at a frequency determined by the shading rate). The results can then either be interpolated or constant over some region of the screen or the interior of a surface element corresponding to one shading sample. This is controlled by the following procedure:

RiShadingInterpolation (**RtToken** type)

This function controls how values are interpolated between shading samples (usually across a polygon or over an area of the screen). If *type* is "constant", the color and opacity of all the pixels inside the region are the same. This is often referred to as *flat* or *faceted shading*. If *type* is "smooth", the color and opacity of all the pixels between shaded values are interpolated from the calculated values. This is often referred to as *Gouraud shading*.

RIB BINDING

ShadingInterpolation "constant"
ShadingInterpolation "smooth"

EXAMPLE

ShadingInterpolation "smooth"

4.2.9 Matte objects

Matte objects are the functional equivalent of three-dimensional hold-out mattes. Matte objects are not shaded and are set to be completely opaque so that they hide objects behind them. However, regions in the output image where a matte object is visible are treated as transparent.

RiMatte (**RtBoolean** onoff)

Indicates whether subsequent primitives are matte objects.

RIB BINDING

Matte *onoff*

EXAMPLE

RiMatte (RI_TRUE);

SEE ALSO

RiSurface

Geometry Attribute	Type	Default	Description
Object-to-World	*transform*	identity	Transformation from object or model coordinates to world coordinates.
Bound	6 *floats*	infinite	Subsequent geometric primitives lie inside this box.
Detail Range	4 *floats*	$(0, 0, \infty, \infty)$	Current range of detail. If the *current detail* is in this range, geometric primitives are rendered.
Geometric Approximation	*token value*	–	The largest deviation of an approximation of a surface from the true surface in raster coordinates.
Cubic Basis Matrices	2 *matrices*	Bezier, Bezier	Basis matrices for bicubic patches. There is a separate basis matrix for both the u and the v directions.
Cubic Basis Steps	2 *integers*	3, 3	
Trim Curves	–	–	A list of trim curves which bound NURBS.
Orientation	*token*	"outside"	Whether primitives are defined in a left-handed or right-handed coordinate system.
Number of Sides	*integer*	2	Whether subsequent surfaces are considered to have one or two sides.
Displacement	*shader*	–	A displacement shader that specifies small changes in surface geometry.

Table 4.11: Geometry Attributes

4.2.10 Bound

The graphics state maintains a bounding box called the *current bound*. The rendering program may clip or cull primitives to this bound.

RiBound (**RtBound** bound)

This procedure sets the *current bound* to ***bound***. The bounding box ***bound*** is specified in the current object coordinate system. Subsequent output primitives should all lie within this bounding box. This allows the efficient specification of a bounding box for a collection of output primitives.

RIB BINDING

Bound *xmin xmax ymin ymax zmin zmax*
Bound [*xmin xmax ymin ymax zmin zmax*]

EXAMPLE

Bound [0 0.5 0 0.5 0.9 1]

SEE ALSO

RiDetail

4.2.11 Detail

The graphics state maintains a *relative detail*, a *current detail*, and a *current detail range*. The *current detail* is used to select between multiple representations of objects each characterized by a different range of detail. The *current detail range* is given by 4 values. These four numbers define transition ranges between this range of detail and the neighboring representations. If the *current detail* lies inside the *current detail range*, geometric primitives comprising this representation will be drawn.

Suppose there are two object definitions, *foo1* and *foo2*, for an object. The first contains more detail and the second less. These are communicated to the rendering program using the following sequence of calls.

```
RiDetail ( bound );
RiDetailRange ( 0., 0., 10., 20. );
    RiObjectInstance ( foo1 );
RiDetailRange ( 10., 20., RI_INFINITY, RI_INFINITY);
    RiObjectInstance ( foo2 );
```

The *current detail* is set by **RiDetail**. The detail ranges indicate that object *foo1* will be drawn when the *current detail* is below 10 (thus it is the low detail detail representation) and that

object *foo2* will be drawn when the *current detail* is above 20 (thus it is the high detail representation). If the *current detail* is between 10 and 20, the rendering program will provide a smooth transition between the low and high detail representations.

RiDetail (**RtBound** bound)

Set the *current bound* to *bound*. The bounding box *bound* is specified in the current coordinate system. The *current detail* is set to the area of this bounding box as projected into the *raster coordinate system*, times the *relative detail*. Before computing the raster area, the bounding box is clipped to the near clipping plane but not to the edges of the display or the far clipping plane. The raster area outside the field of view is computed so that if the camera zooms in on an object the detail will increase smoothly. Detail is expressed in raster coordinates so that increasing the resolution of the output image will increase the detail.

RIB BINDING

Detail *minx maxx miny maxy minz maxz*
Detail [*minx maxx miny maxy minz maxz*]

EXAMPLE

```
RtBound box = { 10.0, 20.0, 42.0, 69.0, 0.0, 1.0 };
RiDetail (box);
```

SEE ALSO

RiBound, **RiDetailRange**, **RiRelativeDetail**

RiDetailRange (**RtFloat** minvisible, **RtFloat** lowertransition,
RtFloat uppertransition, **RtFloat** maxvisible)

Set the *current detail range*. Primitives are never drawn if the *current detail* is less than *minvisible* or greater than *maxvisible*. Primitives are always drawn if the *current detail* is between *lowertransition* and *uppertransition*. All these numbers should be nonnegative and satisfy the following ordering:

$$minvisible \leq lowertransition \leq uppertransition \leq maxvisible.$$

RIB BINDING

DetailRange *minvisible lowertransition uppertransition maxvisible*
DetailRange [*minvisible lowertransition uppertransition maxvisible*]

EXAMPLE

DetailRange [0 0 10 20]

SEE ALSO

RiDetail, **RiRelativeDetail**

If the *Detail* capability is not supported by a particular implementation, all object representations which include RI_INFINITY in their detail ranges are rendered.

4.2.12 Geometric approximation

Geometric primitives are typically approximated by using small surface elements or polygons. The size of these surface elements affects the accuracy of the geometry since large surface elements may introduce straight edges at the silhouettes of curved surfaces or cause particular points on a surface to be projected to the wrong point in the final image.

RiGeometricApproximation (**RtToken** type, **RtFloat** value)

The predefined geometric approximation is "flatness". Flatness is expressed as a distance from the true surface to the approximated surface in pixels. Flatness is sometimes called *chordal deviation*.

RIB BINDING

GeometricApproximation "flatness" *value*
GeometricApproximation *type value*

EXAMPLE

GeometricApproximation "flatness" 2.5

SEE ALSO

RiShadingRate

4.2.13 Orientation and sides

The handedness of a coordinate system is referred to as its *orientation*. The initial "camera" coordinate system is left-handed: x points right, y point up, and z points in. Transformations, however, can flip the orientation of the current coordinate system. An example of a transformation that does not preserve orientation is a reflection. (More generally, a transformation does not preserve orientation if its determinant is negative.)

Similarly, geometric primitives have an orientation, which determines whether their surface normals are defined using a right-handed or left-handed rule in their object coordinate system. Defining the orientation of a primitive to be opposite that of the object coordinate system causes it to be turned inside-out. If a primitive is inside-out, its normal will be computed so that it points in the opposite direction. This has implications for culling, shading, and solids (see the section on Solids and Spatial Set Operations). The outside surface of a primitive is the side from which the normal points outward; the inside surface is the opposite side. The interior of a solid is the volume that is adjacent to the inside surface and

the exterior is the region adjacent to the outside. This is discussed further in the section on Geometric Primitives.

The *current orientation* of primitives is maintained as part of the graphics state independent of the orientation of the current coordinate system. The *current orientation* is initially set to match the orientation of the initial coordinate system, and always flips whenever the orientation of the current coordinate system flips. It can also be modified directly with **RiOrientation** and **RiReverseOrientation**. If the *current orientation* is not the same as the orientation of the current coordinate system, geometric primitives are turned inside out, and their normals are automatically flipped.

RiOrientation (**RtToken** orientation)

This procedure sets the *current orientation* to be either "outside" (to match the current coordinate system), "inside" (to be the opposite of the current coordinate system), "lh" (for explicit left-handed orientation) or "rh" (for explicit right-handed orientation).

RIB BINDING

Orientation orientation

EXAMPLE

Orientation "lh"

SEE ALSO

ReverseOrientation

RiReverseOrientation ()

Causes the *current orientation* to be toggled. If the orientation was right-handed it is now left-handed, and vice versa.

RIB BINDING

ReverseOrientation-

EXAMPLE

RiReverseOrientation ();

SEE ALSO

RiOrientation

Objects can be two-sided or one-sided. Both the inside and the outside surface of two-sided objects are visible, whereas only the outside surface of a one-sided object is visible. If the outside of a one-sided surface faces the viewer, the surface is said to be *frontfacing*, and if the outside surface faces away from the viewer, the surface is *backfacing*. Normally closed surfaces should be defined as one-sided and open surfaces should be defined as two-sided.

The major exception to this rule is transparent closed objects, where both the inside and the outside are visible.

RiSides(**RtInt** sides)

If *sides* is 2, subsequent surfaces are considered two-sided and both the inside and the outside of the surface will be visible. If *sides* is 1, subsequent surfaces are considered one-sided and only the outside of the surface will be visible.

RIB BINDING

Sides *sides*

EXAMPLE

Sides 1

SEE ALSO

RiOrientation

4.3 Transformations

Transformations are used to transform points between coordinate systems. At various points when defining a scene the *current transformation* is used to define a particular coordinate system. For example, **RiProjection** establishes the camera coordinate system, and **RiWorldBegin** establishes the world coordinate system.

The *current transformation* is maintained as part of the graphics state. Commands exist to set and to concatenate specific transformations onto the *current transformation*. These include the basic linear transformations translation, rotation, skew, scale and perspective. Concatenating transformations implies that the *current transformation* is updated in such a way that the new transformation is applied to points *before* the old *current transformation*. Standard linear transformations are given by 4x4 matrices. These matrices are premultiplied by 4-vectors in row format to transform them.

The following three transformation commands set or concatenate a 4x4 matrix onto the *current transformation*:

RiIdentity ()

Set the *current transformation* to the identity.

RIB BINDING

Identity-

EXAMPLE

RiIdentity ( );

SEE ALSO

RiTransform

RiTransform(**RtMatrix** transform)

Set the *current transformation* to the transformation *transform*.

RIB BINDING

Transform *transform*

EXAMPLE

Transform [.5 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]

SEE ALSO

RiIdentity, **RiConcatTransform**

RiConcatTransform (**RtMatrix** transform)

Concatenate the transformation *transform* onto the *current transformation*. The transformation is applied before all previously applied transformations, that is, before the *current transformation*.

RIB BINDING

ConcatTransform *transform*

EXAMPLE

```
RtMatrix foo = { 2.0, 0.0, 0.0, 0.0,    0.0, 2.0, 0.0, 0.0,
                 0.0, 0.0, 2.0, 0.0,    0.0, 0.0, 0.0, 1.0 };
RiConcatTransform ( foo );
```

SEE ALSO

RiIdentity, **RiTransform**, **RiRotate**, **RiScale**, **RiSkew**

The following commands perform local concatenations of common linear transformations onto the *current transformation*.

RiPerspective (**RtFloat** fov)

Concatenate a perspective transformation onto the current transformation. The focal point of the perspective is at the origin and its direction is along the z-axis. The field of view angle, *fov*, specifies the full horizontal field of view.

The user must exercise caution when using this transformation, since points behind the eye will generate invalid perspective divides which are dealt with in a renderer-specific manner.

To request a perspective projection from camera space to screen space, an **RiProjection** request should be used; **RiPerspective** is used to request a perspective modeling transformation from object space to world space, or from world space to camera space.

RIB BINDING

Perspective *fov*

EXAMPLE

Perspective 90

SEE ALSO

RiConcatTransform, **RiDepthOfField**, **RiProjection**

RiTranslate (**RtFloat** dx, **RtFloat** dy, **RtFloat** dz)

Concatenate a translation onto the *current transformation*.

RIB BINDING

Translate *dx dy dz*

EXAMPLE

RiTranslate (0.0, 1.0, 0.0);

SEE ALSO

RiConcatTransform, **RiRotate**, **RiScale**

RiRotate (**RtFloat** angle, **RtFloat** dx, **RtFloat** dy, **RtFloat** dz)

Concatenate a rotation of *angle* degrees about the given axis onto the *current transformation*.

RIB BINDING

Rotate *angle dx dy dz*

EXAMPLE

RiRotate (90.0, 0.0, 1.0, 0.0);

SEE ALSO

RiConcatTransform, **RiScale**, **RiTranslate**

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

Concatenate a scaling onto the *current transformation*.

RIB BINDING

Scale *sx sy sz*

EXAMPLE

Scale 0.5 1 1

SEE ALSO

RiConcatTransform, **RiRotate**, **RiSkew**, **RiTranslate**

RiSkew (**RtFloat** angle, **RtFloat** dx1, **RtFloat** dy1, **RtFloat** dz1,
RtFloat dx2, **RtFloat** dy2, **RtFloat** dz2)

Concatenate a skew onto the *current transformation*. This operation shifts all points along lines parallel to the axis vector (*dx2*, *dy2*, *dz2*). Points along the axis vector (*dx1*, *dy1*, *dz1*) are mapped onto the vector (*x, y, z*), where *angle* specifies the angle (in degrees) between the vectors (*dx1*, *dy1*, *dz1*) and (*x, y, z*), The two axes are not required to be perpendicular, however it is an error to specify an angle that is greater than or equal to the angle between them. A negative angle can be specified, but it must be greater than 180 degrees minus the angle between the two axes.

RIB BINDING

Skew *angle dx1 dy1 dz1 dx2 dy2 dz2*
Skew [*angle dx1 dy1 dz1 dx2 dy2 dz2*]

EXAMPLE

RiSkew (45.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0);

SEE ALSO

RiRotate, **RiScale**, **RiTransform**

4.3.1 Named coordinate systems

Shaders often need to perform calculations in non-standard coordinate systems. The coordinate systems with predefined names are: "raster", "NDC", "screen", "camera", "world", and "object". At any time, the current coordinate system can be marked for future reference.

RiCoordinateSystem (**RtToken** name)

This function marks the coordinate system defined by the current transformation with the name space and saves it. This coordinate system can then be referred to by name in subsequent shaders, or in **RiTransformPoints**. A shader cannot refer to a coordinate system that has not already been named. The list of named coordinate systems is global.

RIB BINDING

CoordinateSystem *name*

EXAMPLE

CoordinateSystem "lamptop"

SEE ALSO

RiCoordSysTransform, **RiTransformPoints**

RiCoordSysTransform (**RtToken** name)

This function replaces the current transformation matrix with the matrix that forms the *name* coordinate system. This permits objects to be placed directly into special or user-defined coordinate systems by their names.

RIB BINDING

CoordSysTransform *name*

EXAMPLE

CoordSysTransform "lamptop"

SEE ALSO

RiCoordinateSystem

RtPoint *
RiTransformPoints (**RtToken** fromspace, **RtToken** tospace, **RtInt** n, **RtPoint** points[])

This procedure transforms the array of points from the coordinate system *fromspace* to the coordinate system *tospace*. This array contains *n* points. If the transformation is successful, the array *points* is returned. If the transformation cannot be computed for any reason (e.g., one of the space names is unknown or the transformation requires the inversion of a noninvertable transformation), NULL is returned.

EXAMPLE

```
RtPoint four_points[4];
RiTransformPoints ("current," "lamptop," 4, four_points);
```

SEE ALSO

RiCoordinateSystem, **RiProjection**, **RiWorldBegin**

4.3.2 Transformation stack

Transformations can be saved and restored recursively. Note that pushing and popping the attributes also pushes and pops the current transformation.

RiTransformBegin ()
RiTransformEnd ()

Push and pop the current transformation. Pushing and popping must be properly nested with respect to the various begin-end constructs.

RIB BINDING

TransformBegin-
TransformEnd-

EXAMPLE

RiTransformBegin ();

SEE ALSO

RiAttributeBegin

4.4 Implementation-specific Attributes

Rendering programs may have additional implementation-specific attributes that control parameters that affect primitive appearance or interpretation. These are all set by the following procedure.

RiAttribute (**RtToken** name, *...parameterlist...*);

Set the parameters of the attribute name, using the values specified in the token-value list *parameterlist*.

Although **RiAttribute** is intended to allow implementation-specific attributes, there are a number of attributes that we expect that nearly all implementations will need to support. It is intended that when identical functionality is required, that all implementations use the attribute names listed in Table 4.12.

RIB BINDING

Attribute *name ...parameterlist...*

EXAMPLE

Attribute "displacementbound" "sphere" [2.0]

SEE ALSO

Attribute name/param	Type	Default	Description
"displacementbound" "sphere" [s]	*float*	0	Amount to pad bounding box for displacement.
"displacementbound" "coordinatesystem" [c]	*string*	"object"	The name of the coordinate system that the displacement bound is measured in.
"identifier" "name" [n]	*string*	""	The name of the object (helpful for reporting errors).
"trimcurve" "sense" [n]	*string*	"inside"	If "inside", trim the interior of Trim Curve regions. If "outside", trim the exterior of the trim region.

Table 4.12: Typical implementation-specific attributes

Section 5

GEOMETRIC PRIMITIVES

The RenderMan Interface supports only surface- and solid-defining geometric primitives. Solid primitives are created from surfaces and combined using set operations. The geometric primitives include:

- planar convex polygons, as well as general planar concave polygons with holes,
- collections of planar convex or general planar concave polygons with holes which share vertices (polyhedra),
- bilinear patches and patch meshes,
- bicubic patches and patch meshes with an arbitrary basis,
- non-uniform rational B-spline surfaces of arbitrary degree (NURBS),
- quadric surfaces, tori, and disks,
- subdivion surface meshes,
- implicit surfaces,
- 1D points and 2D curves or ribbons.

Control vertex points are used to construct polygons, patches, NURBS, subdivision meshes, point clouds, and curves. Point positions can be either an (x,y,z) triplet ("P") or an (x,y,z,w) 4-vector ("Pw"). If the vertex is part of a patch mesh, the position may be used to define a height field. In this case the vertex point contains only a (z) coordinate ("Pz"), and the (x,y)s of points of the height field are set equal to the parametric surface parameters of the mesh.

All primitives have well-defined geometric surface normals, so normals need not be provided with any primitive. The surface normal for a polygon is the perpendicular to the plane containing the polygon. The surface normal for a parametric curved surface is computed by taking the cross product of the surface's parametric derivatives: $(\partial P/\partial u) \times (\partial P/\partial v)$. As mentioned in the Section 4.2.13, Orientation and Sides, if the *current orientation* does not match the orientation of the current coordinate system, normals will be flipped. It is also possible to provide additional shading normals ("N") at polygon and bilinear patch vertices to help make the surface appear smooth.

Quadrics, patches and patch meshes, and NURBS primitives have well-defined global two-dimensional surface parameters. All the points on the surface of each primitive are functions of these parameters *(u,v)*. Except for NURBS and polygons, the domain of the surface parameters is the unit square from 0 to 1. Texture coordinates may be attached to primitives by assigning four sets of texture coordinates, one set to each corner of this unit square. This is done by setting the *current set of texture coordinates* or by defining texture coordinates with the geometric primitives as described below.

Subdivision surfaces and implicit surfaces have locally defined parameterizations, but no globally consistent parameterization across an arbitrary surface of that type.

All geometric primitives normally inherit their color and opacity from the graphics state. However, explicit colors and opacities can be provided when defining the primitive ("Cs" and "Os").

Associated with each geometric primitive definition are additional *primitive variables* that are passed to their shaders. These variables may define quantities that are constant over the surface (class *constant*), piecewise-constant but with separate values per subprimitive (class *uniform*), bilinearly interpolated (class *varying*), or fully interpolated (class *vertex*). If the primitive variable is uniform, there is one value per surface facet. If the primitive variable is varying, there are four values per surface facet, one for each corner of the unit square in parameter space (except polygons, which are a special case). On parametric primitives (quadrics and patches), varying primitive variables are bilinearly interpolated across the surface of the primitive. Colors, opacities, and shading normals are all examples of varying primitive variables.

The standard predefined primitive variables are defined in Table 5.1 Standard Geometric Primitive Variables. Other primitive variables may be predefined by specific implementations or defined by the user with the **RiDeclare** function, or may be declared "in-line" as part of the parameter name itself (see Section 3). Primitive variables which are declared to be of type point (including the three predefined position variables), vector, normal, or matrix are specified in object space, and will be transformed by the current transformation matrix. Any vector or normal variables will be transformed by the equivalent transformation matrix for vectors or normals. Primitive variables which are declared to be of type color must contain the correct number of floating point values as defined in **RiColorSamples**. More information about how to use primitive variables is contained in Part II: The RenderMan Shading Language.

5.1 Polygons

The RenderMan Interface supports two basic types of polygons: a convex polygon and a general concave polygon with holes. In both cases the polygon must be planar. Collections of polygons can be passed by giving a list of points and an array that indexes these points.

The geometric normal of the polygon is computed by computing the normal of the plane containing the polygon (unless it is explicitly specified). If the *current orientation* is left-handed, then a polygon whose vertices were specified in clockwise order (from the point of view of the camera) will be a front-facing polygon (that is, will have a normal vector which points toward the camera). If the *current orientation* is right-handed, then polygons

Information	Name	Type	Class	Floats
Position	"P"	point	vertex	3
	"Pz"	float	vertex	1
	"Pw"	h	vertex	4
Normal	"N"	normal	varying	3
Color	"Cs"	color	varying	(3)
Opacity	"Os"	color	varying	(3)
Texture Coordinates	"s"	float	varying	1
	"t"	float	varying	1
	"st"	2 float	varying	2

Table 5.1: Standard Geometric Primitive Variables

whose vertices were specified in counterclockwise order will be front-facing. The shading normal is set to the geometric normal unless it is explicitly specified at the vertices.

The surface parameters of a polygon are its (x,y) coordinates. This is because the height z of a plane is naturally parameterized by its (x,y) coordinates, unless it is vertical. Texture coordinates are set equal to the surface parameters unless texture coordinates are given explicitly, one set per vertex. Polygons do *not* inherit texture coordinates from the graphics state.

The rules for primitive variable interpolation and texture coordinates are different for polygons than for all other geometric primitives. Constant primitive variables are supplied as a single value for the entire aggregate primitive. Uniform primitive variables are supplied for each polygon. Both varying and vertex primitive variables are supplied for each polygon vertex, and are interpolated across the interior without regard to the artificial surface parameters defined above. Note that interpolating values across polygons is inherently ill-defined. However, linearly interpolating values across a triangle is always well defined. Thus, for the purposes of interpolation, polygons are always decomposed into triangles. However, the details of how this decomposition is done is implementation-dependent and may depend on the view.

RiPolygon (**RtInt** nvertices, *...parameterlist...*)

nvertices is the number of vertices in a single closed planar convex polygon. *parameterlist* is a list of token-array pairs where each token is one of the standard geometric primitive variables or a variable which has been defined with **RiDeclare**. The parameter list must include at least position ("P") information. If a primitive variable is of class vertex or varying, the array contains *nvertices* elements of the type corresponding to the token. If the variable is uniform or constant, the array contains a single element. The number of floats associated with each type is given in Table 5.1, Standard Geometric Primitive Variables.

No checking is done by the RenderMan Interface to ensure that polygons are planar, convex and nondegenerate. The rendering program will attempt to render invalid polygons but the results are unpredictable.

RIB BINDING

Polygon *...parameterlist...*

The number of vertices in the polygon is determined implicitly by the number of elements in the required position array.

EXAMPLE

```
RtPoint points[4] = ( 0.0, 1.0, 0.0, 0.0, 1.0, 1.0,
                      0.0, 0.0, 1.0, 0.0, 0.0, 0.0);
RiPolygon(4, RI_P, (RtPointer)points, RI_NULL);
```

SEE ALSO

RiGeneralPolygon, **RiPointsGeneralPolygons**, **RiPointsPolygons**

An example of the definition of a "Gouraud-shaded" polygon is:

```
RtPoint points[4];
RtColor colors[4];
RiPolygon( 4, "P", (RtPointer)points, "Cs", (RtPointer)colors, RI_NULL);
```

A "Phong-shaded" polygon is given by:

```
RtPoint points[4];
RtPoint normals[4];
RiPolygon( 4, "P", (RtPointer)points, "N", (RtPointer)normals, RI_NULL);
```

RiGeneralPolygon (**RtInt** nloops, **RtInt** nvertices[], *...parameterlist...*)

Define a general planar concave polygon with holes. This polygon is specified by giving *nloops* lists of vertices. The first loop is the outer boundary of the polygon; all additional loops are holes. The array *nvertices* contains the number of vertices in each loop, and has length *nloops*. The vertices in all the loops are concatenated into a single vertex array. The length of this array, *n*, is equal to the sum of all the values in the array *nvertices*.

parameterlist is a list of token-array pairs where each token is one of the standard geometric primitive variables or a variable that has been defined with **RiDeclare**. The parameter list must include at least position ("P") information. If a primitive variable is of class vertex or varying, the array contains *n* elements of the type corresponding to the token. If the variable is uniform or constant, there is a single element of that type. The number of floats associated with each type is given in Table 5.1, Standard Geometric Primitive Variables. The interpretation of these variables is the same as for a convex polygon.

No checking is done by the RenderMan Interface to ensure that polygons are planar and nondegenerate. The rendering program will attempt to render invalid polygons but the results are unpredictable.

RIB BINDING

GeneralPolygon *nvertices ...parameterlist...*

The number of loops in the general polygon is determined implicitly by the length of the *nvertices* array.

EXAMPLE

```
GeneralPolygon [4 3] "P" [ 0 0 0 0 1 0 0 1 1 0 0 1
                           0 0.25 0.5 0 0.75 0.75 0 0.75 0.25 ]
```

SEE ALSO

RiPolygon, **RiPointsPolygons**, **RiPointsGeneralPolygons**

RiPointsPolygons (**RtInt** npolys, **RtInt** nvertices[], **RtInt** vertices[], *...parameterlist...*)

Define *npolys* planar convex polygons that share vertices. The array *nvertices* contains the number of vertices in each polygon and has length *npolys*. The array *vertices* contains, for each polygon vertex, an index into the varying primitive variable arrays. The varying arrays are 0-based. *vertices* has length equal to the sum of all of the values in the *nvertices* array. Individual vertices in the *parameterlist* are thus accessed indirectly through the indices in the array *vertices*.

parameterlist is a list of token-array pairs where each token is one of the standard geometric primitive variables or a variable that has been defined with **RiDeclare**. The parameter list must include at least position ("P") information. If a primitive variable is of class vertex or varying, the array contains *n* elements of the type corresponding to the token, where the number *n* is equal to the maximum value in the array *vertices* plus one. If the variable is uniform, the array contains *npolys* elements of the associated type. If the variable is constant, the array contains exactly one element of the associated type. The number of floats associated with each type is given in Table 5.1, Standard Geometric Primitive Variables. The interpretation of these variables is the same as for a convex polygon.

No checking is done by the RenderMan Interface to ensure that polygons are planar, convex and nondegenerate. The rendering program will attempt to render invalid polygons but the results are unpredictable.

RIB BINDING

PointsPolygons *nvertices vertices ...parameterlist...*

The number of polygons is determined implicitly by the length of the *nvertices* array.

EXAMPLE

```
PointsPolygons [3 3 3] [0 3 2 0 1 3 1 4 3]
        "P" [0 1 1 0 3 1 0 0 0 0 2 0 0 4 0]
        "Cs" [0 .3 .4 0 .3 .9 .2 .2 .2 .5 .2 0 .9 .8 0]
```

SEE ALSO

RiGeneralPolygon, **RiPointsGeneralPolygons**, **RiPolygon**

RiPointsGeneralPolygons (**RtInt** npolys, **RtInt** nloops[], **RtInt** nvertices[], **RtInt** vertices[], *...parameterlist...*)

Define *npolys* general planar concave polygons, with holes, that share vertices. The array *nloops* indicates the number of loops comprising each polygon and has a length *npolys*. The array *nvertices* contains the number of vertices in each loop and has a length equal to the sum of all the values in the array *nloops*. The array *vertices* contains, for each loop vertex, an index into the varying primitive variable arrays. All of the arrays are 0-based. *vertices* has a length equal to the sum of all the values in the array *nvertices*. Individual vertices in the *parameterlist* are thus accessed indirectly through the indices in the array *vertices*.

parameterlist is a list of token-array pairs where each token is one of the standard geometric primitive variables or a variable that has been defined with **RiDeclare**. The parameter list must include at least position ("P") information. If a primitive variable is of storage class vertex or varying, the array contains *n* elements of the type corresponding to the token. The number *n* is equal to the maximum value in the array *vertices* plus one. If the variable is uniform, the array contains *npolys* elements of the associated type. If the variable is constant, the array contains a single element of the associated type. The number of floats associated with each type is given in Table 5.1, Standard Geometric Primitive Variables. The interpretation of these variables is the same as for a convex polygon.

No checking is done by the RenderMan Interface to ensure that polygons are planar and nondegenerate. The rendering program will attempt to render invalid polygons but the results are unpredictable.

RIB BINDING

PointsGeneralPolygons *nloops nvertices vertices ...parameterlist...*

The number of polygons is determined implicitly by the length of the nloops array.

EXAMPLE

```
PointsGeneralPolygons [2 2] [4 3 4 3] [0 1 4 3 6 7 8 1 2 5 4 9 10 11]
        "P" [0 0 1 0 1 1 0 2 1 0 0 0 0 1 0 0 2 0
        0 0.25 0.5 0 .75 .75 0 .75 .25
        0 1.25 0.5 0 1.75 .75 0 1.75 .25]
```

SEE ALSO

RiGeneralPolygon, **RiPointsPolygons**, **RiPolygon**

5.2 Patches

Patches can be either *uniform* or *non-uniform* (contain different knot values). Patches can also be *non-rational* or *rational* depending on whether the control points are (x, y, z) or (x, y, z, w). Patches may also be bilinear or bicubic. The graphics state maintains two 4x4 matrices that define the bicubic patch basis matrices. One of these is the *current u-basis* and the other is the *current v-basis*. Basis matrices are used to transform from the power basis to the preferred basis.

RiBasis (**RtBasis** ubasis, **RtInt** ustep, **RtBasis** vbasis, **RtInt** vstep)

Set the *current u-basis* to *ubasis* and the *current v-basis* to *vbasis*. Predefined basis matrices exist for the common types:

RtBasis RiBezierBasis;
RtBasis RiBSplineBasis;
RtBasis RiCatmullRomBasis;
RtBasis RiHermiteBasis;
RtBasis RiPowerBasis;

The variables *ustep* and *vstep* specify the number of control points that should be skipped in the u and v directions, respectively, to get to the next patch in a bicubic patch mesh. The appropriate step values for the predefined cubic basis matrices are:

Basis	Step
RiBezierBasis	3
RiBSplineBasis	1
RiCatmullRomBasis	1
RiHermiteBasis	2
RiPowerBasis	4

The default basis matrix is **RiBezierBasis** in both directions.

RIB BINDING

Basis *uname ustep vname vstep*
Basis *uname ustep vbasis vstep*
Basis *ubasis ustep vname vstep*
Basis *ubasis ustep vbasis vstep*

For each basis, either the name of a predefined basis (as a string) or a matrix may be supplied. If a basis name specified, it must be one of: "bezier", "b-spline", "catmull-rom", "hermite", or "power".

EXAMPLE

Basis "b-spline" 1 [-1 3 -3 1 3 -6 3 0 -3 3 0 0 1 0 0 0] 1

SEE ALSO

RiPatch, **RiPatchMesh**

Note that the geometry vector used with the **RiHermiteBasis** basis matrix must be (point0, vector0, point1, vector1), which is a permutation of the Hermite geometry vector often found in mathematics texts. Using this formulation permits a step value of 2 to correctly increment over data in Hermite patch meshes.

RiPatch (**RtToken** type, *...parameterlist...*)

Define a single patch. *type* can be either "bilinear" or "bicubic". *parameterlist* is a list of token-array pairs where each token is one of the standard geometric primitive variables or a variable which has been defined with **RiDeclare**. The parameter list must include at least position ("P", "Pw" or "Pz") information. Patch arrays are specified such that u varies faster than v.

Four points define a bilinear patch, and 16 define a bicubic patch. The order of vertices for a bilinear patch is (0,0),(1,0),(0,1),(1,1). Note that the order of points defining a quadrilateral is different depending on whether it is a bilinear patch or a polygon. The vertices of a polygon would normally be in clockwise (0,0),(0,1),(1,1),(1,0) order.

Patch primitive variables that are *constant* or *uniform* should supply one value, which is constant over the patch. Primitive variables that are *varying* should supply four values, one for each parametric corner of the patch (the data will be interpolated bilinearly). Primitive variables that are *vertex* should supply four values for a bilinear patch, or 16 values for a bicubic patch — that is, the same number of values as control vertices "P". A *vertex* primitive variable will be interpolated across the surface in the same manner as the surface position "P". In all cases, the actual size of each array is this number of values times the size of the type associated with the variable.

RIB BINDING

Patch *type ...parameterlist...*

EXAMPLE

```
Patch "bilinear" "P" [ -0.08 0.04 0.05    0 0.04 0.05
                       -0.08 0.03 0.05    0 0.03 0.05]
```

SEE ALSO

RiBasis, **RiNuPatch**, **RiPatchMesh**

RiPatchMesh (**RtToken** type, **RtInt** nu, **RtToken** uwrap,
RtInt nv, **RtInt** vwrap, *...parameterlist...*)

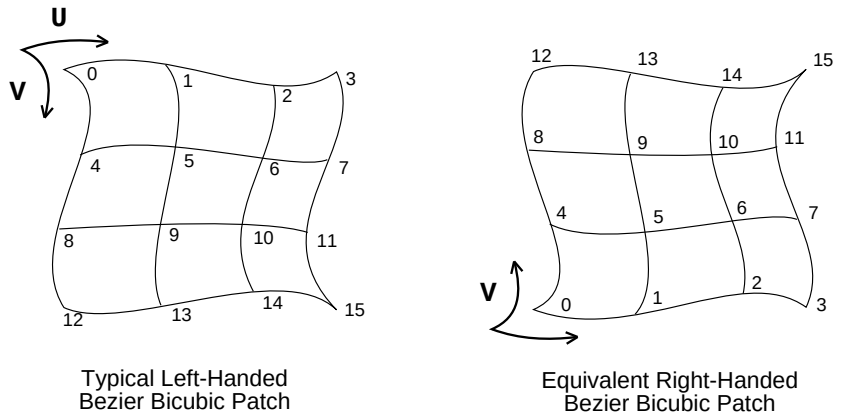


Figure 5.1: Bicubic patch vertex ordering

This primitive is a compact way of specifying a quadrilateral mesh of patches. Each individual patch behaves as if it had been specified with **RiPatch**. *type* can be either "bilinear" or "bicubic". *parameterlist* is a list of token-array pairs where each token is one of the geometric primitive variables or a variable which has been defined with **RiDeclare**. The parameter list must include at least position ("P", "Pw" or "Pz") information. Patch mesh vertex data is supplied in first u and then v order just as for patches. The number of control points in a patch mesh is (*nu*)*(*nv*).

Meshes can wrap around in the u or v direction, or in both directions. If meshes wrap, they close upon themselves at the ends and the first control points will be automatically repeated. As many as three control points may be repeated, depending on the basis matrix of the mesh. The way in which meshes wrap is indicated by giving a wrap mode value of either "periodic" or "nonperiodic".

The actual number of patches produced by this request depends on the type of the patch and the wrap modes specified. For bilinear patches, the number of patches in the u direction, *nupatches*, is given by

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

while for bicubic patches,

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

The same rules hold in the v direction. The total number of patches produced is equal to the product of the number of patches in each direction.

A PatchMesh primitive variable of class *vertex* has the same number of entries as the position "P" (i.e., *nu*×*nv*) and is interpolated using the same order and basis matrices. Any *varying* primitive variables are interpolated piecewise-bilinearly across the patch mesh and contain *n* values, one for each patch corner, where *n* is defined by:

	uwrap	**vwrap**
$n = (nupatches + 1) \cdot nvpatches$	"nonperiodic"	"periodic"
$n = (nupatches + 1) \cdot (nvpatches + 1)$	"nonperiodic"	"nonperiodic"
$n = nupatches \cdot (nvpatches + 1)$	"periodic"	"nonperiodic"
$n = nupatches \cdot nvpatches$	"periodic"	"periodic"

(with *nupatches* and *nvpatches* defined as given above). If a variable is *uniform*, it contains *nupatches* × *nvpatches* elements of its type, one for each patch (see Figure 5.2). Primitive variables of class *constant* have exactly one data element of the appropriate type.

A patch mesh is parameterized by a (*u*,*v*) which goes from 0 to 1 for the entire mesh. Texture maps that are assigned to meshes that wrap should also wrap so that filtering at the seams can be done correctly (see the section on Texture Map Utilities). If texture coordinates are inherited from the graphics state, they correspond to the corners of the mesh.

Height fields can be specified by giving just a z coordinate at each vertex (using "Pz"); the x and y coordinates are set equal to the parametric surface parameters. Height fields cannot be periodic.

RIB BINDING

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

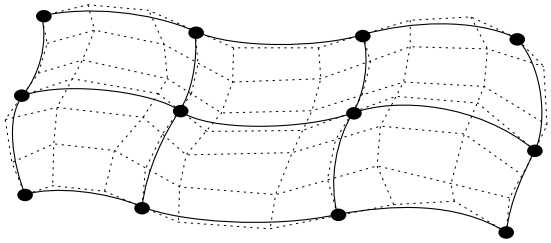
EXAMPLE

```
RtPoint pts[28];
RtFloat foos[2];
RtFloat bars[6];
RiBasis (RiBezierBasis, 3, RiBezierBasis, 3);
RiDeclare ("foo", "uniform float");
RiDeclare ("bar", "varying float");
RiPatchMesh ("bicubic", 7, "nonperiodic", 4, "nonperiodic",
                "P", (RtPointer)pts, "foo", (RtPointer)foos,
                "bar", (RtPointer)bars, RI_NULL);
```

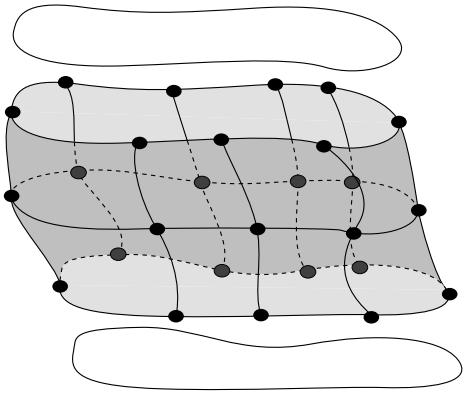
SEE ALSO

RiBasis, **RiNuPatch**, **RiPatch**

Non-uniform B-spline patches are also supported by the RenderMan Interface. Rational quadratic B-splines provide exact representations of many different surfaces including general quadrics, tori, surfaces of revolution, tabulated cylinders, and ruled surfaces.



10 x 7 Aperiodic Bezier Bicubic Patch Mesh
3 x 2 Subpatches
4 x 3 Varying Variable Positions



9 x 5 U-Periodic Catmull-Rom Bicubic Patch Mesh
9 x 2 Subpatches
9 x 3 Varying Variable Positions

Figure 5.2: Patch Meshes

NURBS may contain trim regions and holes that are specified by giving curves in parameter space.

RiNuPatch (**RtInt** nu, **RtInt** uorder, **RtFloat** uknot[], **RtFloat** umin, **RtFloat** umax,
RtInt nv, **RtInt** vorder, **RtFloat** vknot[], **RtFloat** vmin, **RtFloat** vmax,
...parameterlist...)

This procedure creates a tensor product rational or polynomial non-uniform B-spline surface patch mesh. *parameterlist* is a list of token-array pairs where each token is one of the standard geometric primitive variables or a variable that has been defined with **RiDeclare**. The parameter list must include at least position ("P" or "Pw") information.

The surface specified is rational if the positions of the vertices are 4-vectors (x,y,z,w), and polynomial if the positions are 3-vectors (x,y,z). The number of control points in the u direction equals nu and the number in the v direction equals nv. The total number of vertices is thus equal to (nu)*(nv). The *order* must be positive and is equal to the degree of the polynomial basis plus 1. There may be different orders in each parametric direction. The number of control points should be at least as large as the order of the polynomial basis. If not, a spline of order equal to the number of control points is computed. The knot vectors associated with each control point (*uknot[]*, *vknot[]*) must also be specified. Each value in these arrays must be greater than or equal to the previous value. The number of knots is equal to the number of control points plus the order of the spline. The surface is defined in the range *umin* to *umax* and *vmin* to *vmax*. This is different from other geometric primitives where the parameter values are always assumed to lie between 0 and 1. Each *min* must be less than its max. *min* must also be greater than or equal to the corresponding (*order*-1)th knot value. *max* must be less than or equal to the nth knot value.

A NuPatch may be thought of as a nonperiodic uniform B-spline mesh with (1+*nu*−*uorder*) segments in the u parametric direction, and (1+*nv*−*vorder*) segments in the v parametric direction. RiNuPatch primitive variables are therefore defined to have one uniform value per segment and one varying value per segment corner. The number of uniform primitive variables is therefore *nusegments* × *nvsegments*, and the number of varying variables is (*nusegments*+1) × (*nvsegments*+1). Note that this results in redundant parameter values corresponding to repeated knot values, for instance when the knot vector indicates the RiNuPatch is in Bezier form. Primitive variables of class vertex contain *nu* × *nv* values of the appropriate type, and are interpolated using the same methods as the surface position "P". Primitive variables that are of class constant will have a single value for the entire mesh.

If texture coordinates primitive variables are not present, the *current texture coordinates* are assigned to corners defined by the rectangle (*umin*,*umax*) and (*vmin*,*vmax*) in parameter space.

RIB BINDING

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

EXAMPLE

NuPatch 9 3 [0 0 0 1 1 2 2 3 3 4 4 4] 0 4

```
2 2 [ 0 0 1 1 ] 0 1
"Pw" [ 1 0 0 1 1 1 0 1 0 2 0 2
       -1 1 0 1 -1 0 0 1 -1 -1 0 1
       0 -2 0 2 1 -1 0 1 1 0 0 1
       1 0 -3 1 1 1 -3 1 0 2 -6 2
       -1 1 -3 1 -1 0 -3 1 -1 -1 -3 1
       0 -2 -6 2 1 -1 -3 1 1 0 -3 1 ]
```

SEE ALSO

RiPatch, **RiPatchMesh**, **RiTrimCurve**

RiTrimCurve (**RtInt** nloops, **RtInt** ncurves[], **RtInt** order[], **RtFloat** knot[],
RtFloat min, **RtFloat** max, **RtInt** n[], **RtFloat** u[], **RtFloat** v[], **RtFloat** w[])

Set the *current trim curve*. The trim curve contains *nloops* loops, and each of these loops contains *ncurves* curves. The total number of curves is equal to the sum of all the values in *ncurves*. Each of the trimming curves is a non-uniform rational B-spline curve in homogeneous parameter space (u,v,w). The curves of a loop connect in head-to-tail fashion and must be explicitly closed. The arrays *order*, *knot*, *min*, *max*, *n*, *u*, *v*, *w* contain the parameters describing each trim curve. All the trim curve parameters are concatenated together into single large arrays. The meanings of these parameters are the same as the corresponding meanings for a non-uniform B-spline surface.

Trim curves exclude certain areas from the non-uniform B-spline surface definition. The inside must be specified consistently using two rules: an odd winding rule that states that the inside consists of all regions for which an infinite ray from any point in the region will intersect the trim curve an odd number of times, and a curve orientation rule that states that the inside consists of the regions to the "left" as the curve is traced.

Trim curves are typically used to specify boundary representations of solid models. Since trim curves are approximations and not exact, some artifacts may occur at the boundaries between intersecting output primitives. A more accurate method is to specify solids using spatial set operators or constructive solid geometry (CSG). This is described in the section on Solids and Spatial Set Operations, p. 93.

The list of Trim Curves is part of the attribute state, and may be saved and restored using **RiAttributeBegin** and **RiAttributeEnd**.

RIB BINDING

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

The number of loops is determined implicitly by the length of the *ncurves* array.

EXAMPLE

```
RtInt nloops = 1;
RtInt ncurves[1] = { 1 };
RtInt order[1] = { 3 };
```

```
RtFloat knot[12] = { 0,0,0,1,1,2,2,3,3,4,4,4 };
RtFloat min[1] = { 0 };
RtFloat max[1] = { 4 };
RtInt n[1] = { 9 };
RtFloat u[9] = { 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0 };
RtFloat v[9] = { 0.5, 1.0, 2.0, 1.0, 0.5, 0.0, 0.0, 0.0, 0.5 };
RtFloat w[9] = { 1.0, 1.0, 2.0, 1.0, 1.0, 1.0, 2.0, 1.0, 1.0 };
RiTrimCurve (nloops, ncurves, order, knot, min, max, n, u, v, w);
```

SEE ALSO

RiNuPatch, **RiSolidBegin**

5.3 Subdivision Surfaces

The RenderMan Interface includes support for subdivision surfaces. Ordinary cubic B-spline surfaces are rectangular grids of tensor-product patches. Subdivision surfaces generalize these to control grids with arbitrary connectivity. The API for subdivision surfaces looks a lot like **RiPointsPolygons**, with additional parameters to permit the specification of scheme-specific and implementation-specific enhancements.

A subdivision surface, like a parametric surface, is described by its control mesh of points. The surface itself can approximate or interpolate this control mesh while being piecewise smooth. Furthermore, its control mesh is not confined to be rectangular, which is a major limitation of NURBs and uniform B-splines. In this respect, the control mesh is analogous to a polygonal description. But where polygonal surfaces require large numbers of data points to approximate being smooth, a subdivision surface *is* smooth — meaning that polygonal artifacts are never present, no matter how the surface animates or how closely it is viewed.

RiSubdivisionMesh (**RtToken** scheme, **RtInt** nfaces, **RtInt** nvertices[], **RtInt** vertices[],
RtInt ntags, **RtToken** tags[], **RtInt** nargs[],
RtInt intargs[], **RtFloat** floatargs[], *...parameterlist...*)

RiSubdivisionMesh defines a subdivision mesh or surface obeying the subdivision scheme specified by *scheme*. The only standard *scheme* is "catmull-clark", specifying the Catmull-Clark subdivision method. Implementations may also support other schemes. The subdivision mesh is made up of *nfaces* faces. The array *nvertices*, of length *nfaces*, contains the number of vertices in each face. The array *vertices* contains, for each face vertex, an index into the vertex primitive variable arrays. The array vertices has a length equal to the sum of all the values in the array *nvertices*. All the arrays are 0-based.

parameterlist is a list of token-array pairs where each token is one of the standard geometric primitive variables, a variable that has been defined with **RiDeclare**, or is given as an inline declaration. The parameter list must include at least position ("P") information. If a primitive variable is vertex or varying, the array contains n elements of the type corresponding to the token, where n is equal to the maximum value in the array *vertices* plus one. Primitive variables that are vertex will be interpolated according to the subdivision rules (just as "P" is), whereas varying data will be interpolated linearly across faces (as is done for a **PointsPolygons**). If the variable is uniform, the array contains *nfaces* elements of the associated type. If the variable is constant, a single element of the associated type should be provided.

A *component* is either a face, a vertex, or a chain of edges. Components of the subdivision mesh may be tagged by the user to have various implementation-specific properties. The token array *tags*, of length *ntags*, identifies these tags. Each tag has zero or more integer arguments, and zero or more floating-point arguments. The number of arguments provided with each tag is specified by the array *nargs*, which has a length of $ntags \times 2$. For each tag, *nargs* contains an integer specifying the number of integer operands found in the array *intargs*, followed by an integer specifying the number of floating-point operands found in the array *floatargs*. Thus, the length of *intargs* is equal to the sum of all the even-numbered elements of the array *nargs*. The length of *floatargs* is equal to the sum of all the odd-numbered elements of the array *nargs*.

The standard tags are "hole", "crease", "corner", and "interpolateboundary".

The "hole" tag specifies that certain faces are holes. This tag has n integer arguments, one for each face that is a hole, and zero floating-point arguments. Each face is specified by its index in the *nvertices* array.

The "crease" tag specifies that a certain chain of edges should be a sharp crease. This tag has n integer arguments specifying a chain of vertices that make up the crease, and one floating-point argument that is expected to be RI_INFINITY. Each sequential pair of vertices in a crease must be the endpoints of an edge of the subdivision mesh. A mesh may have any number of independent "crease" tags. Individual renderer implementations may choose to expand the functionality of the "crease" tag by making use of tag values other than RI_INFINITY.

The "corner" tag may be used to mark certain vertices as sharp corners. This tag has n integer arguments containing the vertex numbers of the corners and either one or n floating-point arguments that are expected to be RI_INFINITY. Individual renderer implementations may choose to expand the functionality of the "crease" tag by making use of tag values other than RI_INFINITY.

The "interpolateboundary" tag specifies that the subdivision mesh should interpolate all boundary faces to their edges. This tag has zero integer arguments and zero floating-point arguments. It has the same effect as specifying that all the boundary edge-chains are sharp creases and that boundary vertices with exactly two incident edges are sharp corners.

RIB BINDING

SubdivisionMesh *scheme nvertices vertices tags nargs intargs floatargs ...parameterlist...*

The number of faces is determined implicitly by the length of the *nvertices* array.

The number of tags is determined implicitly by the length of the *tags* array, and must match the value as determined from the *nargs* array.

EXAMPLE

SubdivisionMesh "catmull-clark" [4 4 4 4 4 4 4 4 4] [0 4 5 1 1 5 6 2 2 6 7
3 4 8 9 5 5 9 10 6 6 10 11 7 8 12 13 9 9 13 14 10 10 14 15 11]
["interpolateboundary"] [0 0] [0] [0] "P" [-60 60 0 -60 20 0 -60
-20 0 -60 -60 0 -20 60 0 -20 20 45 -20 -20 45 -20 -60 0 20 60 0
20 20 45 20 -20 45 20 -60 0 60 60 0 60 20 0 60 -20 0 60 -60 0]

SEE ALSO

RiPointsPolygons

5.4 Quadrics

Many common shapes can be modeled with quadrics. Although it is possible to convert quadrics to patches, they are defined as primitives because special-purpose rendering programs render them directly and because their surface parameters are not necessarily preserved if they are converted to patches. Quadric primitives are particularly useful in solid and molecular modeling applications.

All the following quadrics are rotationally symmetric about the z axis (see Figure 5.3). In all the quadrics u and v are assumed to run from 0 to 1. These primitives all define a bounded region on a quadric surface. It is not possible to define infinite quadrics. Note that each quadric is defined relative to the origin of the object coordinate system. To position them at another point or with their symmetry axis in another direction requires the use a modeling transformation. The geometric normal to the surface points "outward" from the z-axis, if the *current orientation* matches the orientation of the *current transformation* and "inward" if they don't match. The sense of a quadric can be reversed by giving negative parameters. For example, giving a negative *thetamax* parameter in any of the following definitions will turn the quadric inside-out.

Each quadric has a *parameterlist*. This is a list of token-array pairs where each token is one of the standard geometric primitive variables or a variable which has been defined with **RiDeclare**. For all quadrics, primitive variables of class constant and uniform must supply a single data element of the appropriate type. Primitive variables that are varying or vertex must supply 4 data values, which will be interpolated bilinearly across the quadric surface.

Position variables should not be given with quadrics. All angular arguments to these functions are given in degrees. The trigonometric functions used in their definitions are assumed to also accept angles in degrees.

RiSphere (**RtFloat** radius, **RtFloat** zmin, **RtFloat** zmax, **RtFloat** thetamax, *...parameterlist...*)

Requests a sphere defined by the following equations:

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

Note that if $zmin > -radius$ or $zmax < radius$, the bottom or top of the sphere is open, and that if *thetamax* is not equal to 360 degrees, the sides are also open.

RIB BINDING

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

EXAMPLE

RiSphere (0.5, 0.0, 0.5, 360.0, RI_NULL);

SEE ALSO

RiTorus

RiCone (**RtFloat** height, **RtFloat** radius, **RtFloat** thetamax, *...parameterlist...*)

Requests a cone defined by the following equations:

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

Note that the bottom of the cone is open, and if *thetamax* is not equal to 360 degrees, the sides are open.

RIB BINDING

Cone *height radius thetamax ...parameterlist...*
Cone [*height radius thetamax*] *...parameterlist...*

EXAMPLE

RtColor four_colors[4];
RiCone(0.5, 0.5, 270.0, "Cs", (**RtPointer**)four_colors, RI_NULL);

SEE ALSO

RiCylinder, **RiDisk**, **RiHyperboloid**

RiCylinder (**RtFloat** radius, **RtFloat** zmin, **RtFloat** zmax, **RtFloat** thetamax,
...parameterlist...)

Requests a cylinder defined by the following equations:

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

Note that the cylinder is open at the top and bottom, and if *thetamax* is not equal to 360 degrees, the sides also are open.

RIB BINDING

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

EXAMPLE

Cylinder .5 .2 1 360

SEE ALSO

RiCone, **RiHyperboloid**

RiHyperboloid (**RtPoint** point1, **RtPoint** point2, **RtFloat** thetamax, *...parameterlist...*)

Requests a hyperboloid defined by the following equations:

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

assuming that *point1* $= (x_1, y_1, z_1)$ and *point2* $= (x_2, y_2, z_2)$.

The cone, disk and cylinder are special cases of this surface. Note that the top and bottom of the hyperboloid are open when *point1* and *point2*, respectively, are not on the z-axis. Also, if thetamax is not equal to 360 degrees, the sides are open.

RIB BINDING

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

EXAMPLE

Hyperboloid 0 0 0 .5 0 0 270 "Cs" [1 1 1 .5 .9 1 .2 .9 0 .5 .2 0]

SEE ALSO

RiCone, **RiCylinder**, **RiDisk**

RiParaboloid (**RtFloat** rmax, **RtFloat** zmin, **RtFloat** zmax, **RtFloat** thetamax, *...parameterlist...*)

Requests a paraboloid defined by the following equations:

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

Note that the top of the paraboloid is open, and if *thetamax* is not equal to 360 degrees, the sides are also open.

RIB BINDING

Paraboloid *rmax zmin zmax thetamax ...parameterlist...*
Paraboloid [*rmax zmin zmax thetamax*] *...parameterlist...*

EXAMPLE

Paraboloid .5 .2 .7 270

SEE ALSO

RiHyperboloid

RiDisk (**RtFloat** height, **RtFloat** radius, **RtFloat** thetamax, *...parameterlist...*)

Requests a disk defined by the following equations:

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

Note that the surface normal of the disk points in the positive z direction when *thetamax* is positive.

RIB BINDING

Disk *height radius thetamax ...parameterlist...*
Disk [*height radius thetamax*] *...parameterlist...*

EXAMPLE

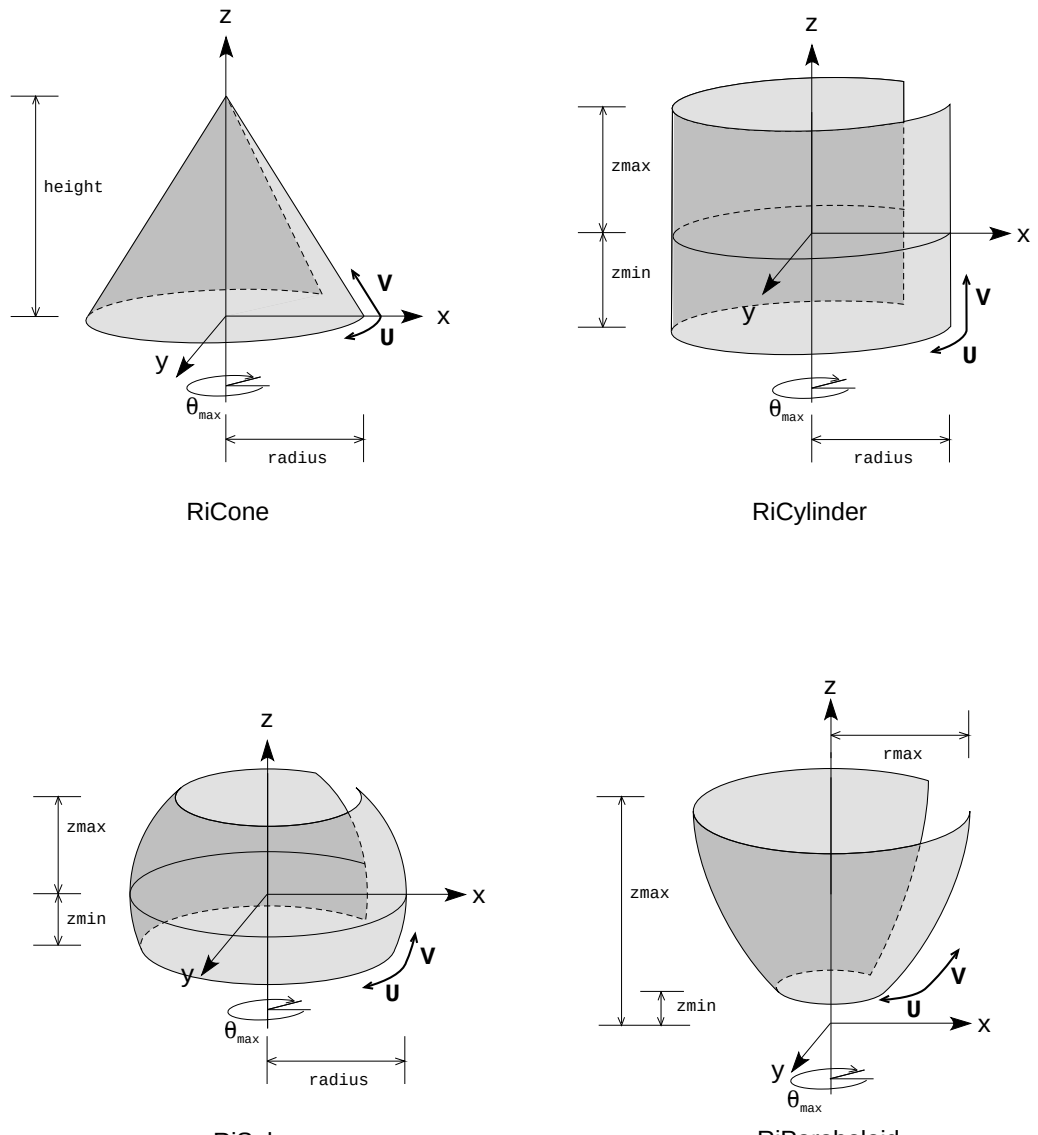
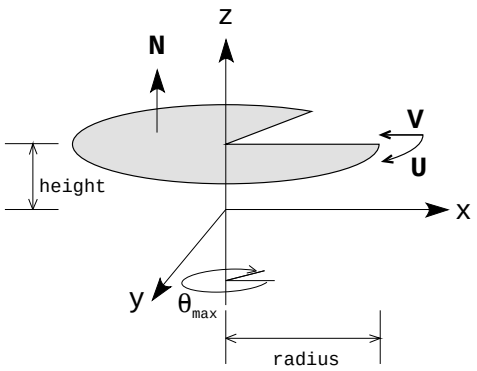
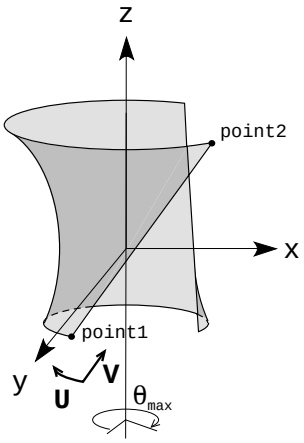


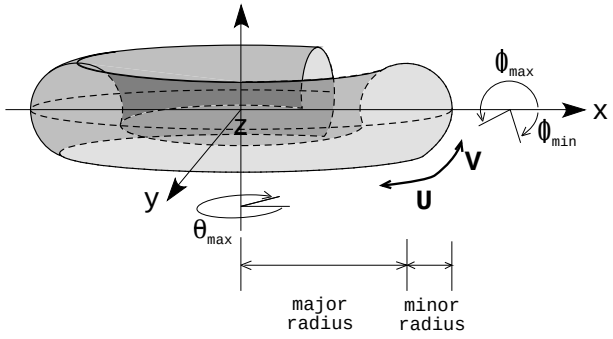
Figure 5.3: Quadric Surface Primitives



RiDisk



RiHyperboloid



RiTorus

Figure 5.4: Quadric Surface Primitives (continued)

RiDisk(1.0, 0.5, 270.0, RI_NULL);

SEE ALSO

RiCone, **RiHyperboloid**

RiTorus (**RtFloat** majorradius, **RtFloat** minorradius, **RtFloat** phimin, **RtFloat** phimax, **RtFloat** thetamax, ...*parameterlist...*)

Requests a torus defined by the following equations:

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

Note that if *phimax* – *phimin* or *thetamax* is not equal to 360 degrees, the torus is open.

RIB BINDING

Torus *rmajor rminor phimin phimax thetamax ...parameterlist...*
Torus [*rmajor rminor phimin phimax thetamax*] *...parameterlist...*

EXAMPLE

Torus 1 .3 60 90 360

SEE ALSO

RiSphere

5.5 Point and Curve Primitives

The RenderMan Interface includes lightweight primitives for specifying point clouds, lines, curves, or ribbons. These primitives are especially useful for representing many particles, hairs, etc.

RiPoints (**RtInt** npoints, ...*parameterlist...*)

Draws *npoints* number of point-like particles. *parameterlist* is a list of token-array pairs where each token is one of the standard geometric primitive variables, a variable that has been defined with **RiDeclare**, or is given as an inline declaration. The parameter list must include at least position ("P") information, one value for each

particle. If a primitive variable is of class varying or vertex, the array contains *npoints* data values of the appropriate type, i.e., one per particle. If the variable is uniform or constant, the array contains a single element.

The size, in object space, of each particle can be specified in the parameter list by using the primitive variable "width", which provides a varying float. If "width" is not specified in the parameter list then it will default to 1.0, meaning that all particles should have an apparent diameter 1.0 units in object space. If all the points are of the same size, the user may specify the variable "constantwidth", which is defined as type constant float to supply a single width value for all points.

Each particle is treated independently. This means a particle is shaded only once and does not have access to derivative information.

RIB BINDING

Points *...parameterlist...*

The number of points is determined implicitly by the length of the "P" array.

EXAMPLE

Points "P" [.5 -.5 0 -.5 -5 0 -.5 .5 0 .5 .5 0] "width" [.1 .12 .05 .02]

RiCurves (**RtToken** type, **RtInt** ncurves, **RtInt** nvertices[], **RtToken** wrap, *...parameterlist...*)

Draws *ncurves* number of lines, curves, or ribbon-like particles of specified width through a set of control vertices. Multiple disconnected individual curves may be specified using one call to **RiCurves**. The parameter *ncurves* is the number of individual curves specified by this command, and *nvertices* is an array of length *ncurves* integers specifying the number of vertices in each of the respective curves.

The interpolation method given by *type* can be either "linear" or "cubic". Cubic curves interpolate using the v basis matrix and step size set by **RiBasis**. The u parameter changes across the width of the curve (if it has any discernable width), whereas the v parameter changes across the length of the curve (i.e., the direction specified by the control vertices). Curves may wrap around in the v direction, depending on whether *wrap* is "periodic" or "nonperiodic". Curves that wrap close upon themselves at the ends and the first control points will be automatically repeated. As many as three control points may be repeated, depending on the basis matrix of the curve.

parameterlist is a list of token-array pairs where each token is one of the standard geometric primitive variables, a variable that has been defined with **RiDeclare**, or is given as an inline declaration. The parameter list must include at least position ("P" or "Pw") information. The width along the curve may be specified with either a special "width" parameter that is a varying float argument, or a "constantwidth" parameter that is a constant float (one value for the entire **RiCurves**). Widths are specified in object space units of the curve. If no "width" vector or "constantwidth" value is given, the default width is 1.0 units in object space.

Primitive variables of class constant should supply a single value of the appropriate type for the entire **RiCurves**. Primitive variables of class uniform should supply a

total of *ncurves* values of the appropriate type. Primitive variables of class varying should supply $\sum(nsegs_i + 1)$ values for nonperiodic curves, and $\sum nsegs_i$ values for periodic curves, where $nsegs_i$ is the number of segments of the ith curve (see below). Primitive variables of class vertex should supply $\sum nvertices_i$ values of the appropriate type, that is, one value for every control vertex "P".

The number of piecewise-linear or piecewise-cubic segments of each individual curve is given by

$$nsegs_i = \begin{cases} nvertices_i - 1 & \text{for linear, nonperiodic curves} \\ nvertices_i & \text{for linear, periodic curves} \\ \frac{nvertices_i - 4}{vstep} + 1 & \text{for cubic, nonperiodic curves} \\ \frac{nvertices_i}{vstep} & \text{for cubic, periodic curves} \end{cases}$$

Since the control vertices only specify the direction of the "spine" of the curves, by default the curves are assumed to always project a cross-section of the specified width (as if it were a hair or a strand of spaghetti). However, if "N" values are supplied, the curves will be interpreted as "flat" ribbons oriented perpendicularly to the supplied normals, thus allowing user-controlled rotation of the ribbon.

RIB BINDING

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

The number of curves is determined implicitly by the length of the *nvertices* array.

EXAMPLE

Curves "cubic" [4] "nonperiodic" "P" [0 0 0 -1 -.5 1 2 .5 1 1 0 -1] "width" [.1 .04]

Curves "linear" [5] "nonperiodic" "P" [0 0 0 3 4 5 -1 -.5 1 2 .5 1 1 0 -1]
"constantwidth" [0.075]

SEE ALSO

RiBasis

5.6 Blobby Implicit Surfaces

The RenderMan Interface allows the use of free-form self-blending implicit-function surfaces in the style of Jim Blinn's blobby molecules, Nishimura et al.'s metaballs and Wyvill, McPheeters and Wyvill's soft objects. Blobby surfaces may be composed of spherical and sausage-like line-segment primitives with extremely flexible control over blending. The surface type also provides for repulsion to avoid intersection with irregular ground planes, represented by depth maps.

RiBlobby (**RtInt** nleaf, **RtInt** ncode, **RtInt** code[], **RtInt** nfloats, **RtFloat** floats[],
RtInt nstrings, **RtString** strings[], *...parameterlist...*)

The *code* array is a sequence of machine language-like instructions describing the object's primitive blob fields and the operations that combine them. Floating point parameters of the primitive fields are stored in the *floats* array. File names of the depth files of repellers are in the *strings* array. The integer *nleaf* is the number of primitive blobs in object, also the number of items in each varying or vertex parameter. Parameters with storage class constant or uniform have exactly one data value for the entire **RiBlobby** primitive.

Each instruction has a numeric opcode followed by a number of operands. Instructions specifying primitive fields start at 1000 and are listed in Table 5.2.

Opcode	Operands	Operation
1000	float	constant
1001	float	ellipsoid
1002	float	segment blob
1003	float, string	repelling plane
1004-1099		*reserved*

Table 5.2: **RiBlobby** opcodes for primitive fields.

For all four of these operators, the operands are indices into the appropriate arrays.

For opcode 1000 (constant) the operand indexes a single floating-point number in the floats array. The index of the first item in the array is zero.

- For opcode 1001 (ellipsoid) the operand indexes the first of 16 floats describing a 4x4 matrix that transforms the unit sphere into the ellipsoidal bump in object space.
- The operand of opcode 1002 (segment blob) indexes 23 floats that give the endpoints and radius of the segment and a 4x4 matrix that transforms the segment into object space.
- Opcode 1003 (repelling ground plane) takes two indices. The first gives the index of the name of a depth map in the strings array. The second indexes the first of 4 float parameters of the repeller's repulsion contour. The value of the field generated by a repeller is a function of the vertical distance from the evaluation point to the z-file, in the view direction in which the z-file was generated. The four float parameters control the shape of the repelling field. Let's call the four parameters A, B, C and D. A controls the overall height of the repeller. The field value is zero whenever the height above the ground plane is larger than A. B controls the sharpness of the repeller. The field looks a lot like $-B/z$ (except that it fades to zero at $z = A$, and remains at a large negative value when $z < 0$), so smaller values of B place the knee in the curve closer to $z = 0$. Added to this negative-going barrier field is a bump that has its peak at $z = C$, and whose maximum value is D. The bump component is exactly zero outside the range $0 \leq z \leq 2C$.

There are several more opcodes that compute composite fields by combining the results of previous instructions in various ways. Every instruction in the code array has a number, starting with zero for the first instruction, that when used as an operand refers to its result. The combining opcodes are given in Table 5.3.

Opcode	Operands	Operation
0	count, ...	add
1	count, ...	multiply
2	count, ...	maximum
3	count, ...	minimum
4	subtrahend, minuend	subtract
5	dividend, divisor	divide
6	negand	negate
7	idempotentate	identity
8-99		*reserved*

Table 5.3: **RiBlobby** opcodes for combining fields.

The *add*, *multiply*, *maximum* and *minimum* operations all take variable numbers of arguments. The first argument is the number of operands, and the rest are indices of results computed by previous instructions. The *identity* operator does nothing useful, and is only included for the convenience of programs that automatically generate RenderMan input.

RiBlobby primitives may be shaded much like ordinary parametric primitives, with the caveat that just like subdivision surfaces, they have no global u and v parameters. Nevertheless, they may be given vertex values by attaching scalar values or reference coordinate fields to primitive sub-objects.

RIB BINDING

Blobby *nleaf* [*floats*] [*strings*] *...parameterlist...*

The number of points is determined implicitly by the length of the "P" array.

EXAMPLE

```
Blobby 2 [ 1001 0     1003 0 16     0 2 0 1 ]
      [1.5 0 0 0 0 1.5 0 0 0 0 1.5 0 0 0 -.1 1     .4 .01 .3 .08] ["flat.zfile"]
```

5.7 Procedural Primitives

Procedural primitives can be specified as follows:

RiProcedural (**RtPointer** data, **RtBound** bound,
RtProcSubdivFunc subdividefunc, **RtProcFreeFunc** freefunc)

This defines a procedural primitive. The *data* parameter is a pointer to an opaque data structure that defines the primitive. (The rendering program does not "look inside" *data*, it simply records it for later use by the procedural primitive.) *bound*

is an array of floats that define the bounding box of the primitive in object space. *subdividefunc* is the routine that the renderer should call (when necessary) to have the primitive subdivided. A bucket-based rendering scheme can potentially save memory space by delaying this call until the bounding box overlaps a bucket that must be rendered. The calling sequence for *subdividefunc* is:

(*subdividefunc)(**RtPointer** data, **RtFloat** detail)

where *data* is the parameter that was supplied in defining the primitive, and *detail* is the screen area of the *bound* of the primitive. When *subdividefunc* is called, it is expected to subdivide the primitive into other smaller procedural primitives or into any number of non-procedural primitives. If the renderer can not determine the true detail of the *bound* (e.g., if the geometric primitive data is being archived to a file), *subdividefunc* may be called with a *detail* value equal to RI_INFINITY. This should be interpreted by the *subdividefunc* as a request for the immediate full generation of the procedural primitive.

freefunc is a procedure that the rendering program calls to free the primitive when the *data* is no longer needed. The calling sequence for *freefunc* is:

(*freefunc)(**RtPointer** data)

Note that the rendering program may call back multiple times with the same procedural primitive, so the data area should not be overwritten or freed until the *freefunc* is called.

The RenderMan Interface provides three standard built-in procedural primitives. Their declarations are:

RtVoid RiProcDelayedReadArchive (**RtPointer** data, **RtFloat** detail);
RtVoid RiProcRunProgram (**RtPointer** data, **RtFloat** detail);
RtVoid RiProcDynamicLoad (**RtPointer** data, **RtFloat** detail);

These built-in procedurals are the *subdivide* routines. All three may use the single built-in *free* function:

RtVoid RiProcFree (**RtPointer** data);

The **RiProcFree** procedure simply calls the standard C free() routine on *data*. The meanings of the standard built-in procedural types are explained below.

RIB BINDING

Procedural *procname* [*args*] [*bound*]

The procedural name *procname* must be a built-in procedural, either one of the standard ones described below or an implementation-specific procedural. The *args* parameter is an array of strings supplying arguments to the built-in procedural. The expected arguments for each type of procedural are explained in the following sections on those primitives.

EXAMPLE

```
RtString args[] = { "sodacan.rib" };
RtBound mybound = { -1, 1, -1, 1, 0, 6 };
```

RiProcedural ((RtPointer)args, mybound, **RiProcDelayedReadArchive**, RiProcFree);

Procedural "DelayedReadArchive" ["sodacan.rib"] [-1 1 -1 1 0 6]

SEE ALSO

RiProcDelayedReadArchive, **RiProcRunProgram**, **RiProcDynamicLoad**

RiProcDelayedReadArchive (**RtPointer** data, **RtFloat** detail)

The built-in procedural **RiProcDelayedReadArchive** operates exactly like **ReadArchive**, except that the reading is delayed until the procedural primitive bounding box is reached (unlike **ReadArchive**, which reads RIB files immediately during parsing). The advantage of the procedural primitive is that because the reading is delayed, memory for the read primitives is not used unless (or until) the contents of the bounding box are actually needed.

The *data* parameter consists of a pointer to an **RtString** array. The first element of the array (that is, ((**RtString** *)data)[0]) is the name of a RIB file to read.

The file can contain any amount of valid RIB, although it is suggested that it either be "flat" (have no hierarchy) or have some balanced hierarchy (matching Begin-End calls). As with all RIB parameters that are bounding boxes, the *boundingbox* is an array of six floating-point numbers, which are *xmin, xmax, ymin, ymax, zmin, zmax* in the current object space.

RIB BINDING

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

The argument string list contains a single string giving the *filename* the file to read when the contents of the *boundingbox* are needed.

EXAMPLE

RtString args[] = { "sodacan.rib" };
RtBound mybound = { -1, 1, -1, 1, 0, 6 };
RiProcedural ((RtPointer)args, mybound, **RiProcDelayedReadArchive**, RiProcFree);

Procedural "DelayedReadArchive" ["sodacan.rib"] [-1 1 -1 1 0 6]

SEE ALSO

RiProcedural, **RiProcRunProgram**, **RiProcDynamicLoad**

RiProcRunProgram (**RtPointer** data, **RtFloat** detail)

The built-in procedural **RiProcRunProgram** will run an external "helper program," capturing its output and interpreting it as RIB input. The *data* parameter consists of a pointer to an **RtString** array. The first element of the array is the name of the program to run (including any command line arguments), and the second element is an argument string to be written to the standard input stream of the helper program.

The helper program generates geometry on-the-fly in response to procedural primitive requests in the RIB stream. Each generated procedural primitive is described by a request to the helper program, in the form of an ASCII datablock that describes the primitive to be generated. This datablock can be anything that is meaningful and adequate to the helper program, such as a sequence of a few floating-point numbers, a filename, or a snippet of code in an interpreted modeling language. In addition, the renderer supplies the *detail size* of the primitive's bounding box, so that the generating program can decide what to generate based on how large the object will appear on-screen.

The generation program reads the request datablocks on its standard input stream and emits RIB commands on its standard output stream. These RIB streams are read into the renderer as though they were read from a file (as with **ReadArchive**) and may include any standard RenderMan attributes and primitives (including procedural primitive calls to itself or other helper programs). As long as any procedural primitives exist in the rendering database that require the same helper program for processing, the socket connection to the program will remain open. This means that the program should be written with a loop that accepts any number of requests and generates a RIB "snippet" for each one.

RIB BINDING

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

The argument list is an array of two strings. The first element is the name of the helper program to execute and may include command line options. The second element is the generation-request data block. It is an ASCII printable string that is meaningful to the helper program and describes the children to be generated. Notice that the data block is a quoted ASCII string in the RIB file, so if it is a complex description that contains quote marks or other special characters, these must be escaped in the standard way. (Similar to C, using backslash metacharacters like \" and \n.) The *boundingbox* is an array of six floating-point numbers, which is *xmin, xmax, ymin, ymax, zmin, zmax* in the current object space.

EXAMPLE

```
RtString args[] = { "perl gencan.pl", "" };
RtBound mybound = { -1, 1, -1, 1, 0, 6 };
RiProcedural ((RtPointer)args, mybound, RiProcRunProgram, RiProcFree);
```

```
Procedural "RunProgram" [ "perl gencan.pl" "" ] [ -1 1 -1 1 0 6 ]
```

The example above presumes that you have a Perl script called gencan.pl that will generate RIB for a model and write that RIB to its standard output stream.

SEE ALSO

RiProcedural, **RiProcRunProgram**, **RiProcDynamicLoad**

RiProcDynamicLoad (**RtPointer** data, **RtFloat** detail)

A more efficient method for accessing subdivision routines is to write them as dynamic shared objects (DSOs)[1], and *dynamically load* them into the renderer executable at run-time. In this case, you write your subdivision and free routines in C, exactly as you would if you were writing them to be linked into the renderer using the C **RiProcedural** interface. DSOs are compiled with special compiler options to make them run-time loadable and are specified in the RIB file by the name of the shared object file. The renderer will load the DSO the first time that the subdivision routine must be called, and from then on, it is called as if (and executes as fast as if) it were statically linked. DSOs are more efficient than external programs because they avoid the overhead of interprocess communication.

When writing a procedural primitive DSO, you must create three specific public subroutine entry points, named Subdivide, Free, and ConvertParameters. Subdivide is a standard RiProcedural() primitive subdivision routine, taking a blind data pointer to be subdivided and a floating-point detail to estimate screen size. Free is a standard **RiProcedural** primitive free routine, taking a blind data pointer to be released. ConvertParameters is a special routine that takes a string and returns a blind data pointer. It will be called exactly once for each DynamicLoad procedural primitive in the RIB file, and its job is to convert a printable string version of the progenitor's blind data (which must be in ASCII in the RIB file), into something that the Subdivide routine will accept.

The C prototypes for these functions are as follows:

```
RtPointer ConvertParameters(char * initialdata );
void Subdivide(RtPointer blinddata, RtFloat detailsize );
void Free( RtPointer blinddata );
```

Note that if the DSO Subdivide routine wants to recursively create child procedural primitives of the same type as itself, it should specify a direct recursive call to itself, with RiProcedural(newdata,newbound,Subdivide,Free), *not* call itself as a DynamicLoad procedural. The latter would eventually just call the former after wasting time checking for and reloading the DSO.

The conventions for how dynamic objects are compiled are implementation-dependent (and also very likely OS-dependent).

RIB BINDING

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

The argument list is an array of two strings. The first element is the name of the shared object file that contains the three required entry points and has been compiled and prelinked as described earlier. The second element is the ASCII printable string that represents the initial data to be sent to the ConvertParameters routine. The *boundingbox* is an array of six floating-point numbers, which is *xmin, xmax, ymin, ymax, zmin, zmax* in the current object space.

EXAMPLE

```
RtString args[] = { "mydso.so", "" };
RtBound mybound = { -1, 1, -1, 1, 0, 6 };
RiProcedural ((RtPointer)args, mybound, RiProcDynamicLoad, RiProcFree);
```

[1] on some systems called dynamically linked libraries, or DLLs

Procedural ”DynamicLoad” [”mydso.so” ””] [-1 1 -1 1 0 6]

SEE ALSO

RiProcedural, **RiProcDelayedReadArchive**, **RiProcRunProgram**

5.8 Implementation-specific Geometric Primitives

Additional geometric primitives can be specified using the following procedure.

RiGeometry (**RtToken** type, *...parameterlist...*)

This procedure provides a standard way of defining an implementation-specific geometric primitive. The values supplied in the parameter list for each primitive is implementation specific.

RIB BINDING

Geometry *name ...parameterlist...*

EXAMPLE

RiGeometry(”teapot”, RI_NULL);

5.9 Solids and Spatial Set Operations

All of the previously described geometric primitives can be used to define a solid by bracketing a collection of surfaces with **RiSolidBegin** and **RiSolidEnd**. This is often referred to as the *boundary representation* of a solid. When specifying a volume it is important that boundary surfaces completely enclose the interior. Normally it will take several surfaces to completely enclose a volume since, except for the sphere, the torus, and potentially a periodic patch or patch mesh, none of the geometric primitives used by the rendering interface completely enclose a volume. A set of surfaces that are closed and non-self-intersecting unambiguously defines a volume. However, the RenderMan Interface performs no explicit checking to ensure that these conditions are met. The inside of the volume is the region or set of regions that have finite volume; the region with infinite volume is considered outside the solid. For consistency the normals of a solid should always point outwards.

RiSolidBegin(**RtToken** operation)
RiSolidEnd ()

SolidBegin the definition of a solid. *operation* may be one of the following tokens: "primitive", "intersection", "union", "difference". Intersection and union operations form the set intersection and union of the specified solids. Difference operations require at least 2 parameter solids and subtract the last *n-1* solids from the first (where *n* is the number of parameter solids).

When the innermost solid block is a "primitive" block, no other **RiSolidBegin** calls are legal. When the innermost solid block uses any other operation, no geometric primitives are legal.

RiSolidEnd terminates the definition of the solid.

RIB BINDING

SolidBegin *operation*
SolidEnd -

EXAMPLE

SolidBegin "union"

SEE ALSO

RiInterior, **RiTrimCurve**

A single solid sphere can be created using

```
SolidBegin ( "primitive" );
    RiSphere ( 1.0, -1.0, 1.0, 360.0, RI_NULL);
RiSolidEnd ();
```

Note that if the same sphere is defined outside of a **RiSolidBegin**-**RiSolidEnd** block, it is not treated as a volume-containing solid. A solid hemisphere can be created with

```
SolidBegin ( "primitive" );
    RiSphere ( 1.0, 0.0, 1.0, 360.0, RI_NULL);
    RiDisk ( 0.0, 1.0, -360.0, RI_NULL);
RiSolidEnd ();
```

(Note that the -360 causes the surface normal of the disk to point towards negative z.)

A composite solid is one formed using spatial set operations. The allowed set operations are "intersection", "union", and "difference". A spatial set operation has n operands, each of which is either a primitive solid defined using **SolidBegin** ("primitive")-**RiSolidEnd**, or a composite solid that is the result of another set operation. For example, a closed cylinder would be subtracted from a sphere as follows:

```
SolidBegin ( "difference" );
    SolidBegin ( "primitive" );
        RiSphere ( 1.0, -1.0, 1.0, 360.0, RI_NULL);
    RiSolidEnd ();
    SolidBegin ( "primitive" );
        RiDisk( 2.0, 0.5, 360.0, RI_NULL);
        RiCylinder( 0.5, -2.0, 2.0, 360.0, RI_NULL);
        RiDisk( -2.0, 0.5, -360.0, RI_NULL);
    RiSolidEnd ();
RiSolidEnd ();
```

When performing a difference the sense of the orientation of the surfaces being subtracted is automatically reversed.

Attributes may be changed freely inside solids. Each section of a solid's surface can have a different surface shader and color. For consistency a single solid should have a single interior and exterior volume shader.

If the *Solid Modeling* optional capability is not supported by a particular implementation, all primitives are rendered as a collection of surfaces, and the spatial set operators are ignored.

5.10 Retained Geometry

A single geometric primitive or a list of geometric primitives may be retained by enclosing them with **RiObjectBegin** and **RiObjectEnd**. The RenderMan Interface allocates and returns an **RtObjectHandle** for each retained object defined in this way. This handle can subsequently be used to reference the object when creating *instances* with **RiObjectInstance**. Objects are not rendered when they are defined within an **RiObjectBegin**-**RiObjectEnd** block; only an internal definition is created.

Transformations, and even Motion blocks, may be used inside an Object block, though they obviously imply a *relative* transformation to the coordinate system active when the Object is instanced. All of an object's attributes are inherited at the time it is instanced, not at the time at which it is created. So, for example, shader assignments or other attributes are not allowed within an Object block. The only exception is **RiBasis**, which may set the interpolation basis matrix used for **RiPatch**, **RiPatchMesh**, or **RiCurves** primitives that are within the Object block.

RtObjectHandle
RiObjectBegin ()

RiObjectEnd ()

RiObjectBegin starts the definition of an *object* and return a handle for later use with **RiObjectInstance**. If the handle returned is NULL, an object could not be created.

RiObjectEnd ends the definition of the current object.

RIB BINDING

ObjectBegin *sequencenumber*

ObjectEnd-

The *sequencenumber* is a unique object identification number which is provided by the RIB client to the RIB server. Both client and server maintain independent mappings between the *sequencenumber* and their corresponding **RtObjectHandle**s. If *sequencenumber* has been used to define a previous object, that object is replaced with the new definition. The number must be in the range 0 to 65535.

EXAMPLE

```
ObjectBegin 2
    Sphere 1 -1 1 360
ObjectEnd
```

SEE ALSO

RiFrameEnd, **RiObjectInstance**, **RiWorldEnd**

RiObjectInstance (**RtObjectHandle** handle)

Create an *instance* of a previously defined object. The object inherits the current set of attributes defined in the graphics state.

RIB BINDING

ObjectInstance *sequencenumber*

The object must have been defined to have a handle *sequencenumber* with a previous **RiObjectBegin**.

EXAMPLE

```
ObjectInstance 2
```

SEE ALSO

RiFrameEnd, **RiObjectBegin**, **RiWorldEnd**

Section 6

MOTION

Some rendering programs are capable of performing temporal antialiasing and motion blur. Motion blur is specified through *moving transformations* and *moving geometric primitives*. Appearance parameters, such as color, opacity, and shader variables can also be changed during a frame. To specify objects that vary over time several copies of the same object are created, each with different parameters at different times within a frame. The times that actually contribute to the motion blur are set with the **RiShutter** command. Parameter values change linearly over the intervals between knots. There is no limit to the number of time values associated with a motion-blurred primitive, although two is usually sufficient.

Rigid body motions and other transformation-based movements are modeled using moving coordinate systems. Moving coordinate systems are created by giving a sequence of transformations at different times and can be concatenated and nested hierarchically. All output primitives are defined in the current object coordinate system and, if that coordinate system is moving, the primitives will also be moving. The extreme case is when the camera is moving, since then all objects in the scene appear to be moving. Moving lights also are handled by placing them in a moving coordinate system. Deforming geometric primitives can also be modeled by giving their parameters at different times.

Moving geometry is created by bracketing the definitions at different times between **RiMotionBegin** and **RiMotionEnd** calls.

RiMotionBegin (**RtInt** n, **RtFloat** t0, **RtFloat** t1,..., **RtFloat** tnminus1)

RiMotionEnd ()

RiMotionBegin starts the definition of a moving primitive. *n* is the number of time steps associated with this moving primitive. The times should be in increasing order. Only one type of RenderMan Interface command can be executed within this sequence and only numerical values may be interpolated.

RiMotionEnd terminates the definition of the moving primitive.

RIB BINDING

MotionBegin [*t0 t1... tn-1*]
MotionEnd -

SEE ALSO

For example, assume the following list of commands creates a static translated sphere:

```
RtFloat Kd = 0.8;
RiSurface( "leather", "Kd", (RtPointer)&Kd, RI_NULL );
RiTranslate( 1., 2., 3. );
RiSphere( 1., -1., 1., 360., RI_NULL );
```

To create a moving, deforming sphere with changing surface qualities, the following might be used:

```
RtFloat Kd[] = { 0.8, 0.7 };
RiMotionBegin( 2, 0., 1. );
    RiSurface( "leather", "Kd", (RtPointer)Kd, RI_NULL );
    RiSurface( "leather", "Kd", (RtPointer)(Kd+1), RI_NULL );
RiMotionEnd();
RiMotionBegin( 2, 0., 1. );
    RiTranslate( 1., 2., 3. );
    RiTranslate( 2., 3., 4. );
RiMotionEnd();
RiMotionBegin( 2, 0., 1. );
    RiSphere( 1., -1., 1., 360., RI_NULL );
    RiSphere( 2., -2., 2., 360., RI_NULL );
RiMotionEnd();
```

Table 6.1, Moving Commands, shows which commands may be specified inside a **RiMotionBegin**-**RiMotionEnd** block. If the *Motion Blur* capability is not supported by a particular implementation, only the transformations, geometry and shading parameters from t_0 are used to render each moving object.

Transformations	Geometry	Shading
RiTransform	**RiBound**	**RiColor**
RiConcatTransform	**RiDetail**	**RiOpacity**
RiPerspective	**RiPolygon**	**RiLightSource**
RiTranslate	**RiGeneralPolygon**	**RiAreaLightSource**
RiRotate	**RiPointsPolygons**	
RiScale	**RiPointsGeneralPolygons**	**RiSurface**
RiSkew	**RiPatch**	**RiInterior**
	RiPatchMesh	**RiExterior**
RiProjection	**RiNuPatch**	**RiAtmosphere**
RiDisplacement	**RiSphere**	
RiDeformation	**RiCone**	
	RiCylinder	
	RiHyperboloid	
	RiParaboloid	
	RiDisk	
	RiTorus	
	RiPoints	
	RiCurves	
	RiSubdivisionMesh	
	RiBlobby	

Table 6.1: Moving Commands

Section 7

EXTERNAL RESOURCES

7.1 Texture Map Utilities

The format of the various texture map files is implementation dependent. However, there are standard utilities that convert image files into texture map files.

During two-dimensional texture access, texture coordinates (*s, t*) are mapped onto the texture such that *s=0* maps to *xmin*, *s=1* maps to *xmax+1*, *t=0* maps to *ymin*, and *t=1* maps to *ymax+1*. To be precise, all accesses to the half-open interval [0,1) in *s* and *t* will lie within the picture data.

A *wrapmode* describes how the texture is accessed if the texture coordinates are outside the unit square (less than zero, or greater than or equal to one). The *swrap* and *twrap* strings specify the wrapping behavior of the *s* and *t* coordinates. The standard wrapping behavior for *s* and *t*, "black", is to return the value zero for all accesses outside the unit square. (Thus an RGBa texture will be transparent black, zero on all four channels.) The keyword "periodic" indicates that values of *s* (or *t*) outside [0,1) will be mapped into [0,1) by subtracting the largest integer less than or equal to the coordinate (the "floor" of the coordinate). This will wrap the value 1 back to 0, the value 1.25 to 0.25, and the value -0.1 to 0.9. The result will be to repeat the texture as a tile that fills texture space in the s (or t) direction. The keyword "clamp" indicates that values of *s* (or *t*) outside [0,1) will be mapped into [0,1) by clamping them at their minimum and maximum values. All values below zero will be clamped to zero and all values greater than or equal to one will be clamped to a value slightly less than one (at the last texture pixel).

Textures are often prefiltered so that subsequent antialiasing calculations can be done more quickly at run-time. This is controlled by giving a *filterfunc*, which is the same as the *filterfunc* used in **RiPixelFilter**, and an *swidth* and *twidth*.

7.1.1 Making texture maps

Surface textures are used to modify the properties of a surface, such as color and opacity. A surface texture is accessed using the surface texture coordinates (see the section on Texture coordinates) or any other two-dimensional coordinates computed by a user-defined shader. A surface texture consists of one or more *channels*. A single channel or a group of *n*

channels (usually an RGB color) can be accessed using the *texture* function of the Shading Language. The *texture* function requires the name of a *texture file* containing the texture.

RiMakeTexture (char *picturename, char *texturename, **RtToken** swrap, **RtToken** twrap,
RtFilterFunc filterfunc, **RtFloat** swidth, **RtFloat** twidth, *...parameterlist...*)

Convert an image in a standard picture file whose name is *picturename* into a texture file whose name is *texturename*. All channels of the picture file will be converted (in order) to texture *channels*. The storage format of the texture file and the precision of stored texture channels are implementation-dependent.

The picture file used as input is not changed or otherwise affected by **RiMakeTexture**.

RIB BINDING

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

The *filter* parameter should be one of "box", "triangle", "catmull-rom", "b-spline", "gaussian" and "sinc". These correspond to the predefined filter functions described in **RiPixelFilter**.

EXAMPLE

```
RiMakeTexture("globe.pic", "globe.tx", "periodic", "clamp",
              RiGaussianFilter, 2.0, 2.0, RI_NULL);
```

SEE ALSO

RiTextureCoordinates, *texture()* in the Shading Language

7.1.2 Making environment maps

Environment maps are images representing the color of an environment in a particular direction. An environment map is accessed using a direction vector; this vector is often the direction of a mirror reflection and hence environment maps are often referred to as reflection maps. However, any direction can be computed by a user-defined shader. An environment map image consists of one or more *channels*. A single channel or a group of *n* channels (usually an RGB color) can be accessed using the *environment* function in the Shading Language. Environment maps can be input in two formats. The first is as a single latitude-longitude image. Environment maps in this form are fairly easy to create using a paint system. The second format is a set of six cube face projections. Environment maps in this form are naturally created by the rendering program.

RiMakeLatLongEnvironment (char *picturename, char *texturename,
RtFilterFunc filterfunc, **RtFloat** swidth, **RtFloat** twidth, *...parameterlist...*)

Convert an image in a standard picture file representing a latitude-longitude map whose name is *picturename* into an environment map whose name is *texturename*. The storage format of the texture file and the precision of stored texture channels are implementation-dependent.

This image has longitude equal to 0 degrees at the left, and 360 degrees at the right. The latitude at the bottom is -90 degrees and at the top is 90 degrees. The bottom of the picture is at the south pole and the top the north pole. The direction in space corresponding to each of the points on the image is given by:

$$\begin{aligned} x &= \cos(longitude) \cdot \cos(latitude) \\ y &= \sin(longitude) \cdot \cos(latitude) \\ z &= \sin(latitude) \end{aligned}$$

Notice that latitude-longitude environment maps are sensitive to the handedness of the coordinate system in which they will be accessed. Environment maps which are intended to be accessed in a right-handed coordinate system will, if displayed, appear as a mirror image of those intended to be accessed in a left-handed coordinate system.

RIB BINDING

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

The *filter* parameter should be one of "box", "triangle", "catmull-rom", "b-spline", "gaussian" and "sinc". These correspond to the predefined filter functions described with **RiPixelFilter**.

EXAMPLE

MakeLatLongEnvironment "long.pic" "long.tx" "catmull-rom" 3 3

SEE ALSO

RiMakeCubeFaceEnvironment, environment() in the Shading Language

---

RiMakeCubeFaceEnvironment (char *px, char *nx, char *py, char *ny, char *pz, char *nz, char *texturename, **RtFloat** fov, **RtFilterFunc** filterfunc, **RtFloat** swidth, **RtFloat** twidth, *...parameterlist...*)

Convert six images in standard picture files representing six viewing directions into an environment map whose name is *texturename*. The image *pz* (*nz*) is the image as viewed in the positive (negative) z direction. The remaining images are those viewed along the positive and negative x and y directions. The storage format of the texture file and the precision of stored texture channels are implementation-dependent.

Each image is normally produced by a rendering program by placing the eye at the center of the environment (usually the origin) and generating a picture in each of the six directions. These pictures are the projection of the environment onto a set of cube faces. Each face is usually assumed to be unit distance from the eye point. Cube face environment maps should be generated with the following orientations:

Image	Forward Axis	Up Axis	Right Axis
px	+X	+Y	-Z
nx	-X	+Y	+Z
px	+Y	-Z	+X
nx	-Y	+Z	+X
px	+Z	+Y	+X
nx	-Z	+Y	-X

Notice that cube face environment maps are sensitive to the handedness of the coordinate system in which they will be accessed. Environment maps which are intended to be accessed in a right-handed coordinate system will, if displayed, appear as a mirror image of those intended to be accessed in a left-handed coordinate system.

The *fov* is the full horizontal field of view used to generate these images. A value of 90 degrees will cause the cube face edges to meet exactly. Using a slightly larger value will cause the cube faces to intersect. Having a slight overlap helps remove artifacts along the seams where the different pictures are joined.

RIB BINDING

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

The *filter* parameter should be one of "box", "triangle", "catmull-rom", "b-spline", "gaussian" and "sinc". These correspond to the predefined filter functions described with **RiPixelFilter**.

EXAMPLE

```
RiMakeCubeFaceEnvironment("foo.x", "foo.nx", "foo.y", "foo.ny",
                          "foo.z", "foo.nz", "foo.env", 95.0,
                          RiTriangleFilter, 2.0, 2.0, RI_NULL);
```

SEE ALSO

RiMakeLatLongEnvironment, environment() in the Shading Language

---

7.1.3 Making shadow maps

Shadow maps are depth buffer images from a particular view. They are generally used in light source shaders to cast shadows onto objects. A shadow map is accessed by point in the camera coordinate system corresponding to that view. This point must be computed in the shader. A shadow map texture can be accessed using the *shadow* function of the Shading Language. The *shadow* function requires the name of a *texture file* containing the texture.

RiMakeShadow (char *picturename, char *texturename, *...parameterlist...*)

Create a depth image file named *picturename* into a shadow map whose name is *texturename*. The storage format of the shadow map texture file and the precision of stored texture channels are implementation-dependent.

RIB BINDING

MakeShadow *picturename texturename ...parameterlist...*

EXAMPLE

MakeShadow "shadow.pic" "shadow.tex"

SEE ALSO

shadow() in the Shading Language

7.2 Errors

RenderMan Interface procedures do not return error status codes. Instead, the user may specify an error handling routine that will be called whenever an error is encountered.

RiErrorHandler (**RtErrorHandler** handler)

This procedure sets the error handling procedure invoked by the renderer when an error is detected. Error handling procedures have the following form:

RtVoid handler (**RtInt** code, **RtInt** severity, char *message)

code indicates the type of error, and *severity* indicates how serious the error is. Values for *code* and severity are defined in "ri.h". The *message* is a character string containing an error message formatted by the renderer which can be printed or displayed, as the handler desires.

The following standard error handlers are defined:

RtVoid RiErrorIgnore;
RtVoid RiErrorPrint;
RtVoid RiErrorAbort;
RtInt RiLastError;

If **RiErrorIgnore** is specified, all errors are ignored and no diagnostic messages are generated. If **RiErrorPrint** is specified, a diagnostic message is generated for each error. The rendering system will attempt to ignore the erroneous information and continue rendering. If **RiErrorAbort** is specified, the first error will cause a diagnostic message to be generated and the rendering system will immediately terminate. Each of the standard error handlers saves the last error code in the global variable **RiLastError**. This procedure can be called outside an **RiBegin**-**RiEnd** block.

RIB BINDING

ErrorHandler "ignore"
ErrorHandler "print"
ErrorHandler "abort"

If "ignore", "print", or "abort" is specified, the equivalent predefined error handling procedure will be invoked in the RIB server. Notice that the RIB parser process may detect RIB stream syntax errors which make it impossible to correctly parse a request. In this case, the error procedure will be invoked and the parser will do its best to resynchronize the input stream by scanning for the next recognizable token.

EXAMPLE

ErrorHandler "ignore"

7.3 Archive Files

One important use of the RIB protocol is to store a scene description in an archive file for rendering at a later time or in a remote location from the modeling application. Appendix D, RenderMan Interface Bytestream Conventions, outlines a structuring conventions to make these archives as portable and useful as possible.

RiArchiveRecord (**RtToken** type, char *format [, arg ...])

This call writes a user data record (data which is outside the scope of the requests described in the rest of Part I of this document) into a RIB archive file or stream. *type* is one of "comment", or "structure", or "verbatim". "comment" begins the user data record with a RIB comment marker and terminates it with a newline. "structure" begins the user data record with a RIB structuring convention preface and terminates it with a newline. "verbatim" just outputs the data as-is, with no leading comment character and no trailing newline. The user data record itself is supplied as a printf() format string with optional arguments. It is an error to embed newline characters in the format or any of its string arguments.

RiReadArchive (**RtToken** name, RtVoid (*callback)(**RtToken**,char*,...), *...parameterlist...*)

This function will read the named file. Each RIB command in the archive will be parsed and executed exactly as if it had been called by the application program directly, or been in-line in the calling RIB file, as the case may be. This is essentially a RIB-file include mechanism.

In the C API version, the *callback* parameter is a function which will be called for any RIB user data record or structure comment which is found in the file. This routine

has the same prototype as **RiArchiveRecord**, and allows the application routine to notice user data records and then execute special behavior based on them as the file is being read into the renderer. If a NULL value is passed for *callback*, comments and structures in the RIB file will simply be ignored.

RIB BINDING

ReadArchive *filename*

EXAMPLE

ReadArchive "sodacan.rib"

SEE ALSO

RiArchiveRecord

Part II

The RenderMan Shading Language

Section 8

INTRODUCTION TO THE SHADING LANGUAGE

Remarkably realistic images can be produced with a few fairly simple shapes made from interesting materials and lighted in a natural way. Creating a photorealistic image requires the specification of these material and lighting properties. This part of the document describes the Shading Language, which is used to write custom shading and lighting procedures called *shaders*. Providing a language allows a user to extend shading models or to create totally new ones. Models of light sources with special lenses, concentrators, flaps or diffusers can be created. The physics of materials can be simulated or special materials can be created. This is done by modeling the interaction of light at the surface and in the interior of a region of space. Material types can also be combined, simulating the many coats of paint or finish applied to a surface. Providing a shading language also allows many of the tricks and shortcuts commonly performed during production rendering to be accommodated without destroying the conceptual integrity of the shading calculations. Visualizing the results of scientific simulations is also easier because shaders can be written that produce a surface color that is based directly on the results of a computation. For example, it is possible to write a shader that sets the surface color based on temperature and surface curvature. Shaders can also be used to modify the final pixel values before they are written to the display.

The Shading Language is a C-like language with extensions for handling color and point data types. A large number of trigonometric and mathematical functions, including interpolation and noise functions, are provided. Color operators are provided that simulate the mixing and filtering of light. Point and vector operators perform common geometric operations such as dot and cross product. A collection of commonly used geometric functions is also provided. These include functions to transform points to specific coordinate systems. Common lighting and shading formulas, such as *ambient*, *diffuse*, *specular*, or *phong*, are available as built-in functions. Built-in texture access functions return values from images representing texture maps, environment maps, and shadow depth maps. The texture coordinates given to these functions can be either the presupplied texture coordinates or values computed in the Shading Language. Since texture map values are just like any other value in the language, they can be used to control any aspect of the shading calculation. There is in principle no limit to the number of texture maps per surface.

The Shading Language can be used for specifying surface displacement functions such as ripples or nubs. Shading Language functions are also used for pixel operations. This type of shader is referred to as an *imager*. Imagers are used to do special effects processing, to compensate for non-linearities in display media, and to convert to device dependent color

spaces (such as CMYK or pseudocolor).

Section 9

OVERVIEW OF THE SHADING PROCESS

In this document, *shading* includes the entire process of computing the color of a point on a surface or at a pixel. The shading process requires the specification of *light sources*, *surface material properties*, *volume* or *atmospheric effects*, and pixel operations. The interpolation of color across a primitive, in the sense of Gouraud or Phong interpolation, is not considered part of the shading process. Each part of the shading process is controlled by giving a function which mathematically describes that part of the shading process. Throughout this document the term shader refers to a procedure that implements one of these processes. There are thus five major types of shaders:

- *Light source shaders*. Lights may exist alone or be attached to geometric primitives. A light source shader calculates the color of the light emitted from a point on the light source towards a point on the surface being illuminated. A light will typically have a color or spectrum, an intensity, a directional dependency and a fall-off with distance.
- *Displacement shaders*. These shaders change the position and normals of points on the surface, in order to place bumps on surfaces.
- *Surface shaders*. Surface shaders are attached to all geometric primitives and are used to model the optical properties of materials from which the primitive was constructed. A surface shader computes the light reflected in a particular direction by summing over the incoming light and considering the properties of the surface
- *Volume shaders*. Volume shaders modulate the color of a light ray as it travels through a volume. Volumes are defined as the insides of solid objects. The atmosphere is the initial volume defined before any objects are created.
- *Imager shader*. Imager shaders are used to program pixel operations that are done before the image is quantized and output.

Conceptually, it is easiest to envision the shading process using ray tracing (see Figure 9.1). In the classic recursive ray tracer, rays are cast from the eye through a point on the image plane. Each ray intersects a surface which causes new rays to be spawned and traced recursively. These rays are typically directed towards the light sources and in the directions of maximum reflection and transmittance. Whenever a ray travels through space, its color and intensity is modulated by the volume shader attached to that region of space. If that region is inside a solid object, the volume shader is the one associated with the interior of

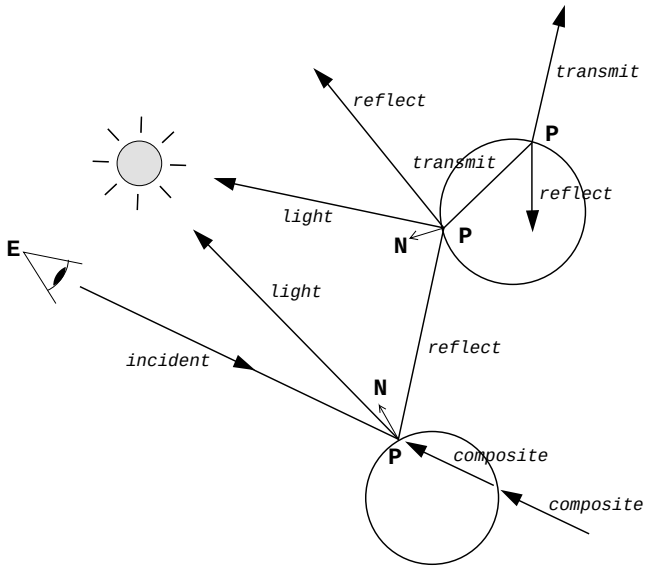


Figure 9.1: The ray tracing paradigm

that solid; otherwise, the exterior shader of the spawning primitive is used. Whenever an incident ray intersects a surface, the surface shader attached to that geometric primitive is invoked to control the spawning of new rays and to determine the color and intensity of the incoming or incident ray from the color and intensity of the outgoing rays and the material properties of the surface. Finally, whenever a ray is cast to a light source, the light source shader associated with that light source is evaluated to determine the color and intensity of the light emitted. The shader evaluation pipeline is illustrated in Figure 9.2.

This description of the shading process in terms of ray tracing is done because ray tracing provides a good metaphor for describing the optics of image formation and the properties of physical materials. However, the Shading Language is designed to work with any rendering algorithm, including scanline and z-buffer renderers, as well as radiosity and other global illumination programs.

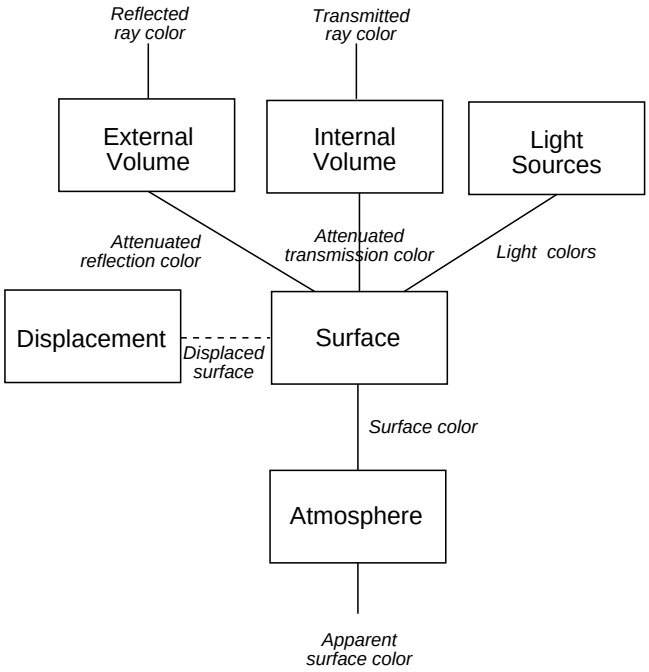


Figure 9.2: Shader evaluation pipeline

Section 10

RELATIONSHIP TO THE RENDERMAN INTERFACE

The Shading Language is designed to be used with the RenderMan Interface described in Part I of this document. This interface is used by modeling systems to describe scenes to a rendering system. Like most graphics systems, the RenderMan Interface maintains a *current graphics state*. This state contains all the information needed to render a geometric primitive.

The graphics state contains a set of attributes that are attached to the surface of each geometric primitive. These shading attributes include the *current color* (set with **RiColor** and referred to as Cs) and *current opacity* (set with **RiOpacity** and referred to as Os). The geometric primitive also contains a *current surface shader* (**RiSurface**) and several volume shaders: the *current atmosphere shader* (**RiAtmosphere**), *current interior shader* (**RiInterior**) and a *current exterior shader* (**RiExterior**). All geometric primitives use the *current surface shader* for computing the surface shading at their surfaces. Light directed toward the viewer is attenuated with the *current atmosphere shader*. If the surface shader of a primitive causes rays to be traced (with the trace() function), the ray color will be attenuated with either the *current exterior shader* or *current interior shader*, depending on whether the traced ray is in the direction of, or opposite to, the surface normal.

The graphics state also contains a *current list of light sources* that contains the light sources that illuminate the geometric primitives. Light sources may be added to or removed from this list using **RiIlluminate**. Light sources can be attached to geometric primitives to define area light sources (**RiAreaLightSource**) or procedurally define their geometric properties (**RiLightSource**). The *current area light source* contains a list of geometric primitives that define its geometry. Defining a geometric primitive adds that primitive to this list.

The graphics state also maintains the *current transformation* that is the transformation corresponding to the modeling hierarchy. The graphics state also contains a *current displacement shader* (**RiDisplacement**) and an *imager* (**RiImager**).

The RenderMan Interface predefines standard shaders for light sources, surfaces, and volumes. These standard shaders are available in all rendering programs that implement the RenderMan Interface, even if some algorithmic limitation prevents them from supporting programmable shaders. Standard and implementation-dependent shaders should always be specified in the Shading Language, even if they are built in. The predefined shaders provided by the Shading Language are listed in Table 10.1, Standard Shaders. There is also a *null* shader that is used as a placeholder. Shading Language definitions for these shaders are given in Appendix A.

Type	Shader
Light sources	ambientlight
	distantlight
	pointlight
	spotlight
Surfaces	constant
	matte
	metal
	shinymetal
	plastic
	paintedplastic
Atmosphere	fog
	depthcue
Displacement	bumpy
Imager	background

Table 10.1: Standard Shaders

Shaders contain *instance* variables that customize a particular shader of that type. For a surface shader these variables may denote material properties; for a light source shader these variables may control its intensity or directional properties. All instance variables have default values that are specified in the definition of the shader. When a shader is added to the graphics state, these default values may be overridden by user-supplied values. This is done by giving a parameter list consisting of name-value pairs. The names in this list are the same as the names of the instance variables in the shader definition. Note that many different versions of the same shader can be instanced by giving different values for its instance variables. The instance variables associated with a shader effectively enlarge the current graphics state to include new appearance attributes. Because the attributes in the graphics state are so easily extended in this way, the number of "built-in" or "predefined" shading-related variables in the graphics state has been kept to a minimum.

There are several steps involved in using a shader defined in the Shading Language. First, a text file containing the source for the shader is created and edited. Second, this file is then compiled using the Shading Language compiler to create an object file. Third, the object file is placed in a standard place to make it available to the renderer. At this point, a shader programmed in the Shading Language is equivalent to any other shader used by the system. When a RenderMan Interface command involving a programmed shader (that is, one that is not *built-in*) is invoked, the shader is looked up by name in the table of available shaders, read into the rendering program, and initialized. This shader is then instanced using the instance variables supplied in the RenderMan Interface procedure. Finally, when a geometric primitive is being shaded, the shaders associated with it are executed.

Section 11

TYPES

The Shading Language is strongly typed and supports the following basic types:

11.1 Floats

Floats are used for all scalar calculations. They are also used for integer calculations.

Floating-point variables are defined as follows:

```
float a, b=1;
```

The initialization value may be any scalar expression.

11.2 Colors

The Shading Language implements color as an abstract data type independent of the number of samples and the color space. The major operations involving color are color addition ('+' operator) corresponding to the mixing of two light sources, and color filtering ('*' operator) corresponding to the absorption of light by a material. In each case these color operations proceed on a component by component basis.

The number of color samples used in the rendering program is set through the RenderMan Interface. Once the number of color samples has been specified, colors with the appropriate number of samples must be passed to a shader. When setting the number of samples, the user can also specify a transformation from RGB space to this n sample color space. This allows a shader to convert color constants to the specified color space.

Color component values of 0 correspond to minimum intensity, while values of 1 correspond to maximum intensity. A color constant of 0 is equivalent to black, and of 1 is equivalent to white. However, values outside that range are allowed (values less than zero decrement intensity from other sources). Note that if you are using a `color` variable to represent reflectivity, only component values between 0 and 1 are physically meaningful, whereas `color` variables that represent radiance may be unbounded.

Color variables may be declared with:

```
color c, d=1, e=color(1,0,0);
```

The initialization value may be any scalar or color expression. If a scalar expression is used, its value is promoted to a color by duplicating its value into each component.

Colors may be specified in a particular color space by:

```
color [space] (u,v,w)
```

The optional specifier *space*, which must be a string literal, indicates the color coordinate system of the 3-tuple. The default color coordinate system is "rgb". This construct lets you specify the color *u*,*v*,*w* in a particular color space, but this statement implicitly converts the color into its "rgb" equivalent. Table 11.1, Color Coordinate Systems, lists the color coordinate systems that are supported in the Shading Language.

Coordinate System	Description
"rgb"	Red, green, and blue.
"hsv"	Hue, saturation, and value.
"hsl"	Hue, saturation, and lightness.
"xyz", "XYZ"	CIE *XYZ* coordinates.
"YIQ"	NTSC coordinates.

Table 11.1: Color Coordinate Systems.

11.3 Points, Vectors, and Normals

Point-like variables are *(x,y,z)* triples of floats that are used to store locations, direction vectors, and surface normals.

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 special type of vector that is perpendicular to a surface, and thus describes the surface's orientation.

All calculations involving points, vectors, and normals are assumed to take place in an implementation-dependent coordinate system, usually either the *camera* or *world coordinate system*. Procedures exist to transform points, vectors, and normals from the shading coordinate system to various named coordinate systems, or to define a point, vector, or normal in one of several coordinate systems and transform it to the shading coordinate system. It should be noted that point locations, direction vectors, and normals do not transform in the same way, and therefore it is important to use the correct transformation routine for each type.

A number of standard coordinate systems are known to a shader. These include: "raster", "NDC", "screen", "camera", "world", and "object". These are discussed in the section on Camera in Part I. In addition, a shader knows the coordinate systems shown in Table 11.2, Point Coordinate Systems.

Point-like variables are declared:

```
point u, v=1, w=point(1,1,1);
```

Coordinate System	Description
"shader"	The coordinate system in which the shader was defined. This is the "object" coordinate system when the shader is defined.
"current"	The coordinate system in which the shading calculations are being performed. This is implementation-dependent, but is usually either the "camera" or "world" coordinate system.
string	A named coordinate system established using **RiCoordinateSystem**.

Table 11.2: Point coordinate systems.

```
vector R;
normal Nf;
```

The initialization value may be any scalar or point-like expression. If a scalar expression is used, the value is promoted to a point (or vector or normal) by duplicating its value into each component.

Point, vector, and normal constants default to be in the "current" coordinate system. Points, vectors, and normals can be specified in any known coordinate system with:

```
point [space] (x,y,z)
vector [space] (x,y,z)
normal [space] (x,y,z)
```

where the *space* is a string literal containing the name of the coordinate system. For example,

```
point "world" (0,0,0)
```

defines a point at the position (0,0,0) in world coordinates. This point is implicitly transformed to the "current" coordinate system. And similarly,

```
vector "object" (0,0,1)
```

defines a vector pointing toward the $+z$ axis in "object" space, which is implicitly transformed into the equivalent vector expressed in the "current" coordinate system. Points, vectors, and normals passed through the RenderMan Interface are interpreted to be in "shader" or "object" space, depending on whether the variable was set using a shader command or a geometric primitive command, respectively. All points, vectors, and normals passed through the RenderMan Interface are transformed to the "current" coordinate system before shader execution begins.

11.4 Transformation Matrices

The Shading Language has a matrix type that represents the transformation matrix required to transform points and vectors from one coordinate system and another. Matrices are represented internally by 16 floats (a 4×4 homogeneous transformation matrix).

A matrix can be declared:

```
matrix zero = 0;
matrix ident = 1;
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:

```
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.

11.5 Strings

Strings are used to name external objects (texture maps, for example). String literals (or constants) are enclosed in double quotes, as in the C language, and may contain any of the standard C "escape sequences" (such as \n for newline or \" for a quote).

11.6 Arrays

The Shading Language supports 1D arrays of all the basic data types. Local variable arrays must be of fixed, predeclared (compile-time) lengths, though arrays can be of indeterminate length for shader instance variable and function formal parameter declarations. Zero and negative-length arrays are not permitted.

As in C, the syntax for declaring an array of any data type uses square brackets, as *datatype*[*length*]. *length* is a float constant, which is rounded down to generate an integer length. The syntax for specifying the data of a constant array uses curly braces, as {value1, value2, ...}. For example:

```
float a[10];
uniform color C[4]; float b[4] = { 3.14, 2.17, 0, -1.0 };
```

As in C, individual array elements may referenced using the familiar square bracket notation. An array element index is a float expression, which is rounded down to generate and integer index. Shading Language arrays perform over/underrun checking at run-time. Also as in C, arrays themselves are not atomic objects — in other words, you may not assign an entire array to another array, or compare entire arrays.

11.7 Uniform and Varying Variables

A renderer implementation may choose to shade many points, or even large regions of a surface, at once. How large a such a region may be is implementation-dependent.

Shaders contain two classes of variables: uniform variables are those whose values are constant over whatever portion of the surface is being shaded, while varying variables are those that may take on different values at different locations on the surface being shaded. For example, shaders inherit a color and a transparency from the graphics state. These values do not change from point to point on the surface and are thus uniform variables. Color and opacity can also be specified at the vertices of geometric primitives (see Section 5, Geometric Primitives). In this case they are bilinearly interpolated across the surface, and therefore are varying variables.

Local variables and arguments to shaders are declared to be either uniform or varying by specifying a storage *modifier*:

```
varying point p;
uniform point q;
```

Variables declared in the argument list of a shader are assumed to be uniform variables by default. These are sometimes referred to as *instance* variables. If a variable is provided only when a shader is instanced, or if it is attached to the geometric primitive as a whole, it should be declared a uniform variable. However, if a variable is to be attached to the vertices of geometric primitive, it should be declared as a varying variable in the shader argument list.

Variables declared locally in the body of a shader, as arguments to a function, or as local variables are assumed to be varying. Declaring a variable to be uniform inside a shader or function definition is never necessary, but may allow the compiler to generate more efficient code, particularly for renderer implementations that can shade large regions of a surface at once.

If a uniform value (or a constant) is assigned to a varying variable or is used in a varying expression, it will be promoted to varying by duplication. It is an error to assign a varying value to a uniform variable or to use a varying value in a uniform expression.

Section 12

SHADER EXECUTION ENVIRONMENT

When a shader is attached to a geometric primitive it inherits a set of varying variables that completely defines the environment in the neighborhood of the surface element being shaded. These state variables are predefined and should not be declared in a Shading Language program. It is the responsibility of the rendering program to properly initialize these variables before a shader is executed.

All the predefined variables which are available to each type of shader are shown in Table 12.1, *Predefined Surface Shader Variables*, through Table 12.5, *Predefined Imager Shader Variables*. In these tables the top section describes state variables that can be read by the shader. The bottom section describes the state variables that are the expected results of the shader. By convention, capitalized variables refer to points and colors, while lower-case variables are floats. If the first character of a variable's name is a C or O, the variable refers to a color or opacity, respectively. Colors and opacities are normally attached to light rays; this is indicated by appending a lowercase subscript. A lowercase *d* prefixing a variable name indicates a derivative.

All predefined variables are considered to be read-only, with the exception of the result variables, which are read-write in the appropriate shader type, and Cs, Os, N, s and t, which are read-write in any shader in which they are readable. Vectors are not normalized by default.

12.1 Surface Shaders

The geometry is characterized by the surface position P which is a function of the surface parameters (u,v). The rate of change of surface parameters are available as (du,dv). The parametric derivatives of position are also available as dPdu and dPdv. The actual change in position between points on the surface is given by P(u+du)=P+dPdu*du and P(v+dv)=P+dPdv*dv. The calculated geometric normal perpendicular to the tangent plane at P is Ng. The shading normal N is initially set equal to Ng unless normals are explicitly provided with the geometric primitive. The shading normal can be changed freely; the geometric normal is automatically recalculated by the renderer when P changes, and cannot be changed by shaders. The texture coordinates are available as (s,t). Figure 12.1 shows a small surface element and its associated state.

The optical environment in the neighborhood of a surface is described by the incident ray

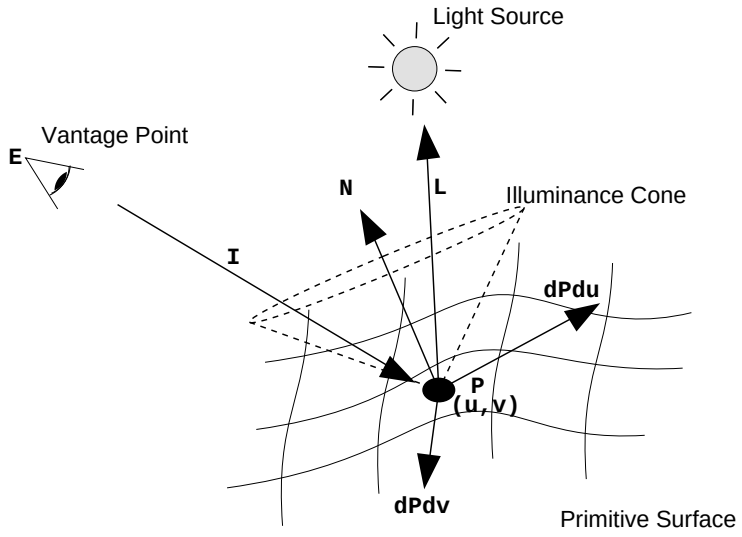


Figure 12.1: Surface shader state

I and light rays L. The incoming rays come either directly from light sources or indirectly from other surfaces. The direction of each of these rays is given by L; this direction points from the surface towards the source of the light. A surface shader computes the outgoing light in the direction $-$I from all the incoming light. The color and opacity of the outgoing ray is Ci and Oi. (Rays have an opacity so that compositing can be done after shading. In a ray tracing environment, opacity is normally not computed.) If either Ci or Oi are not set, they default to black and opaque, respectively.

12.2 Light Source Shaders

A light source shader is slightly different (see Figure 12.2, Light source shader state). It computes the amount of light cast along the direction L which arrives at some point in space Ps. The color of the light is Cl while the opacity is Ol. The geometric parameters described above (P, du, N, etc.) are available in light source shaders; however, they are the parameters of the light emitting surface (e.g., the surface of an area light source)–*not* the parameters of any primitive being illuminated. If the light source is a point light, P is the origin of the light source shader space and the other geometric parameters are zero. If either Cl or Ol are not set, they default to black and opaque, respectively.

Name	Type	Storage Class	Description
Cs	color	varying	Surface color
Os	color	varying	Surface opacity
P	point	varying	Surface position
dPdu	vector	varying	Derivitive of surface position along u
dPdv	vector	varying	Derivitive of surface position along v
N	normal	varying	Surface shading normal
Ng	normal	varying	Surface geometric normal
u,v	float	varying	Surface parameters
du,dv	float	varying	Change in surface parameters
s,t	float	varying	Surface texture coordinates
L	vector	varying	Incoming light ray direction*
Cl	color	varying	Incoming light ray color*
Ol	color	varying	Incoming light ray opacity*
E	point	uniform	Position of the eye
I	vector	varying	Incident ray direction
ncomps	float	uniform	Number of color components
time	float	uniform	Current shutter time
dtime	float	uniform	The amount of time covered by this shading sample.
dPdtime	vector	varying	How the surface position P is changing per unit time, as described by motion blur in the scene.
Ci	color	varying	Incident ray color
Oi	color	varying	Incident ray opacity

* Available only inside illuminance statements.

Table 12.1: Predefined Surface Shader Variables

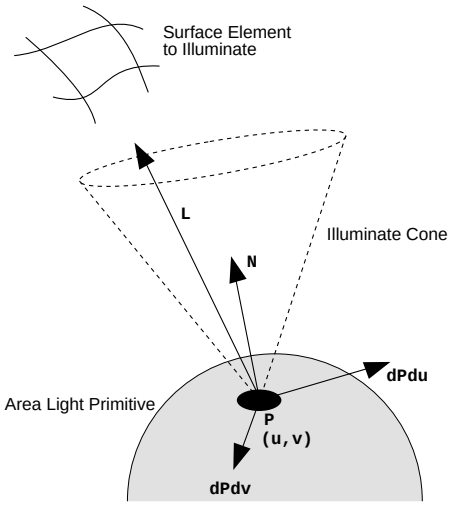


Figure 12.2: Light Source Shader State

12.3 Volume Shaders

A volume shader is not associated with a surface, but rather attenuates a ray color as it travels through space. As such, it does not have access to any geometric surface parameters, but only to the light ray I and its associated values. The shader computes the new ray color at the ray origin P $-$I. The length of I is the distance traveled through the volume from the origin of the ray to the point P.

12.4 Displacement Shaders

The displacement shader environment is very similar to a surface shader, except that it only has access to the geometric surface parameters. It computes a new P and/or a new N. In rendering implementations that do not support the *Displacement* capability, modifications to P will not actually move the surface (change the hidden surface elimination calculation); however, modifications to N will still occur correctly.

12.5 Imager Shaders

An imager shader manipulates a final pixel color after all of the geometric and shading processing has concluded. In the context of an imager shader, P is the position of the pixel center in *raster space*, with the z component being 0.

Name	Type	Storage Class	Description
P	point	varying	Surface position on the light
dPdu	vector	varying	Derivitive of surface position along u
dPdv	vector	varying	Derivitive of surface position along v
N	normal	varying	Surface shading normal on the light
Ng	normal	varying	Surface geometric normal on the light
u,v	float	varying	Surface parameters
du,dv	float	varying	Change in surface parameters
s,t	float	varying	Surface texture coordinates
L	vector	varying	Outgoing light ray direction*
Ps	point	varying	Position being illuminated
E	point	uniform	Position of the eye
ncomps	float	uniform	Number of color components
time	float	uniform	Current shutter time
dtime	float	uniform	The amount of time covered by this shading sample.
Cl	color	varying	Outgoing light ray color
Ol	color	varying	Outgoing light ray opacity

* Available only inside solar or illuminate statements.

Table 12.2: Predefined Light Source Variables

Name	Type	Storage Class	Description
P	point	varying	Light ray endpoint
I	vector	varying	Ray direction (pointing toward P)
E	point	uniform	Position of the eye
Ci	color	varying	Ray color
Oi	color	varying	Ray opacity
ncomps	float	uniform	Number of color components
time	float	uniform	Current shutter time
dtime	float	uniform	The amount of time covered by this shading sample.
Ci	color	varying	Attenuated ray color
Oi	color	varying	Attenuated ray opacity

Table 12.3: Predefined Volume Shader Variables

Name	Type	Storage Class	Description
P	point	varying	Surface position
dPdu	vector	varying	Derivitive of surface position along u
dPdv	vector	varying	Derivitive of surface position along v
N	normal	varying	Surface shading normal
Ng	normal	varying	Surface geometric normal
I	vector	varying	Incident ray direction
E	point	uniform	Position of the eye
u,v	float	varying	Surface parameters
du,dv	float	varying	Change in surface parameters
s,t	float	varying	Surface texture coordinates
ncomps	float	uniform	Number of color components
time	float	uniform	Current shutter time
dtime	float	uniform	The amount of time covered by this shading sample.
dPdtime	vector	varying	How the surface position P is changing per unit time, as described by motion blur in the scene.
P	point	varying	Displaced surface position
N	normal	varying	Displaced surface shading normal

Table 12.4: Predefined Displacement Shader Variables

Name	Type	Storage Class	Description
P	point	varying	Pixel raster position
Ci	color	varying	Pixel color
Oi	color	varying	Pixel opacity
alpha	float	uniform	Fractional pixel coverage
ncomps	float	uniform	Number of color components
time	float	uniform	Shutter open time.
dtime	float	uniform	The amount of time the shutter was open.
Ci	color	varying	Output pixel color
Oi	color	varying	Output pixel opacity

Table 12.5: Predefined Imager Shader Variables

Section 13

LANGUAGE CONSTRUCTS

13.1 Expressions

Expressions are built from arithmetic operators, function calls, and variables. The language supports the common arithmetic operators (+, -, *, and /) plus the vector operators ^ (cross product) and . (dot product), and the C conditional expression (*binary relation* ? *expr1* : *expr2*).

When operating on points, vectors, normals, or colors, an arithmetic operation is performed in parallel on each component. If a binary operator involves a float and a multi-component type (such as a point, vector, normal, matrix, or color), the float is promoted to the appropriate type by duplicating its value into each component. It is illegal to perform a binary operation between a point and a color. Cross products only apply to vectors; dot products apply to both vectors and colors. Two points, two colors, or two strings can be compared using == and !=. Points cannot be compared to colors.

The usual common-sense mathematical rules apply to point/vector/normal arithmetic. For example, a vector added to a point yields a point, a vector added to a vector yields a vector, and a point subtracted from a point yields a vector. Mixing the types of vectors, normals, and points (for example, taking a cross product of two points, rather than two vectors) is allowed, but is discouraged. A particular implementation may choose to issue a compiler warning in such cases. Note that vectors and normals may be used nearly interchangably in arithmetic expressions, but care should be taken to distinguish between them when performing coordinate system transformations.

Matrix variables can be tested for equality and inequality with the == and != boolean operators. 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`.

13.2 Standard Control Flow Constructs

The basic explicit control flow constructs are:

- block-structured statement grouping,

- conditional execution,
- loops, and
- function calls.

These constructs are all modeled after C. Statement grouping allows a single statement to be replaced with a sequence of statements.

```
{
    stmt;
    ...
    stmt;
}
```

Any variables declared within such a brace-delimited series of statements are only visible within that block. In other words, variables follow the same local scoping rules as in the C language.

Conditional execution is controlled by

if (*boolean expression*) *stmt* **else** *stmt*

There are two loop statements,

while (*boolean expression*) *stmt*

and

for (*expr* ; *boolean expression* ; *expr*) *stmt*

A boolean expression is an expression involving a relational operator, one of: <, >, <=, >=, ==, and !=. It is not legal to use an arbitrary float, point or color expression directly as a boolean expression to control execution. A boolean expression can not be used as a floating point expression.

The

break [*n*]

and

continue [*n*]

statements cause either the for or the while loop at level *n* to be exited or to begin the next iteration. The default value for *n* is 1 and refers to the immediately enclosing loop.

Built-in and user-progrmamed functions are called just as in C. The

return *expr*

statement is used to return a value to the caller. Any functions that do not return a value should be declared as **void**, just as in ANSI C, and should not contain a **return** statement.

13.3 Illuminance and Illuminate Statements

The Shading Language contains three new block statement constructs: illuminance, illuminate, and solar. illuminance is used to control the integration of incoming light over a hemisphere centered on a surface in a surface shader. illuminate and solar are used to specify the directional properties of light sources in light shaders. If a light source does not have an illuminate or solar statement, it is a non-directional ambient light source.

Unlike other control statements, illuminance, illuminate, and solar statements cannot be nested. However, multiple illuminance, illuminate, or solar statements may be given sequentially within a single shader.

The illuminance statement controls integration of a procedural reflectance model over the incoming light. Inside the illuminance block two additional variables are defined: Cl or light color, and L or light direction. The vector L points towards the light source, but may not be normalized (see Figure 12.2). The arguments to the illuminance statement specify a three-dimensional solid cone. The two forms of an illuminance statement are:

illuminance([string category,] point *position*)
    statements

illuminance([string category,] point *position*, vector *axis*, float *angle*)
    statements

The first form specifies that the integral extends over the entire sphere centered at *position*. The second form integrates over a cone whose apex is on the surface at *position*. This cone is specified by giving its centerline, and the angle between the side and the axis in radians. If *angle* is PI, the cone extends to cover the entire sphere and these forms are the same as the first form. If *angle* is PI/2, the cone subtends a hemisphere with the north pole in the direction *axis*. Finally, if *angle* is 0, the cone reduces to an infinitesimally thin ray.

Light shaders can specify "categories" to which they belong by declaring a string parameter named __category (this name has two underscores), whose value is a comma-separated list of categories into which the light shader falls. When the illuminance statement contains an optional string parameter *category*, the loop will only consider lights for which the *category* is among those listed in its comma-separated __category list. If the illuminance *category* begins with a - character, then only lights *not* containing that category will be considered.

When the optional *category* string is omitted, an illuminance loop will execute its body for every nonambient light source.

A Lambertian shading model is expressed simply using the illuminance statement:

```
Nn = normalize(N);
illuminance( P, Nn, PI/2 ) {
    Ln = normalize(L);
    Ci += Cs * Cl * Ln.Nn;
}
```

This example integrates over a hemisphere centered at the point on the surface with its north pole in the direction of the normal. Since the integral extends only over the upper hemisphere, it is not necessary to use the customary max(0,Ln.Nn) to exclude lights that are locally occluded by the surface element.

The illuminate and solar statements are inverses of the illuminance statement. They control the casting of light in different directions. The point variable L corresponding to a particular light direction is available inside this block. This vector points outward from the light source. The color variable Cl corresponds to the color in this direction and should be set. Like the illuminance statements, illuminate and solar statements cannot be nested.

The illuminate statement is used to specify light cast by local light sources. The arguments to the illuminate statement specify a three-dimensional solid cone. The general forms are:

illuminate(*position*) *stmt*
illuminate(*position, axis, angle*) *stmt*

The first form specifies that light is cast in all directions. The second form specifies that light is cast only inside the given cone. The length of L inside an illuminate statement is equal to the distance between the light source and the surface currently being shaded.

The solar statement is used to specify light cast by distant light sources. The arguments to the solar statement specify a three-dimensional cone. Light is cast from distant directions inside this cone. Since this cone specifies only directions, its apex need not be given. The general forms of the solar statement are:

solar( ) *stmt*
solar(*axis, angle*) *stmt*

The first form specifies that light is being cast from all points at infinity (e.g., an illumination map). The second form specifies that light is being cast from only directions inside a cone.

An example of the solar statement is the specification of a distant light source:

```
solar( D, 0 )
    Cl = intensity * lightcolor;
```

This defines a light source at infinity that sends light in the direction D. Since the angle of the cone is 0, all rays from this light are parallel.

An example of the illuminate statement is the specification of a standard point light source:

```
illuminate( P )
    Cl = (intensity * lightcolor) / (L.L)
```

This defines a light source at position P that casts light in all directions. The 1/L.L term indicates an inverse square law fall off.

Section 14

SHADERS AND FUNCTIONS

The Shading Language distinguishes between *shaders* and *functions*. Shaders are procedures that are referred to by RenderMan Interface procedures. Functions are procedures that can be called from within the Shading Language. The distinction between shaders and functions is primarily a consequence of different argument passing conventions.

14.1 Shaders

Shaders are introduced by the keywords light, surface, volume, displacement, or imager and are followed by the name of the shader and the statements that comprise its definition. These keywords indicate the type of RenderMan Interface shader that is being defined. The RenderMan Interface uses the shader type to enforce a type match in subsequent calls to RenderMan Interface procedures. For example, it is illegal to declare a shader to be of type light and then instance it using **RiSurface**.

The arguments to a shader are referred to as its *instance variables*. All of these variables are required have default values, and are assumed to be uniform unless declared otherwise. Shader instance variable values can be changed when a particular shader is instanced from the RenderMan Interface. For example, consider the shader weird:

```
surface
weird( float a=0.5; varying float b=0.25 )
{
    Ci = color (mod(s,a), abs(sin(a+b)), mod(b,t));
}
```

This surface shader may be referenced through the RenderMan Interface with the **RiSurface** command. For example,

```
RiSurface ( "weird", RI_NULL );
```

instances the above shader with all its defaults.

Shader instance variable values can be changed from their defaults by passing their new values through the RenderMan Interface. This first requires that the type of the variable be declared. The declaration for weird would be:

```
RiDeclare( "a", "uniform float" );
RiDeclare( "b", "varying float" );
```

Of course, the **RiDeclare** can be eliminated if "in-line declarations" are used (see Section 3). In either case, uniform instance variables can be set by passing them in the *parameterlist* of a shader instance. In the following example, a is redefined while b remains equal to its default:

```
RtFloat a = 0.3;
RiSurface( "weird", "a", (RtPointer)&a, RI_NULL );
```

Shader variables can also be set in geometric primitives. For example, the weird shader variables could be set when defining a primitive:

```
RtFloat a;
RtFloat bs[4];
RtPoint Ps[4];
RiPolygon(4, "P", Ps, "a", (RtPointer)&a, "b", (RtPointer)bs, RI_NULL)
```

a is a single float and b is an array containing four values, since it is a varying variable. The standard variable "P" is predeclared to be of type varying point.

If a geometric primitive sets a shader variable that is defined in none of the shaders associated with that primitive, it is ignored. Variables that are set on geometric primitives override the values set by the shader instance.

Shader instance variables are read-only in the body of the shader, unless they are declared using the **output** keyword:

```
displacement
lumpy ( float a=0.5; output varying float height=0 )
{
    float h = a * noise(P);
    P += h * normalize(N);
    N = calculatenormal(N);
    height = h;
}
```

This displacement shader produces a lumpy surface and also saves the displacement amount in an output variable *height*. Output variables of one shader may be read by other shaders on the same primitive (see Section 15.8) in order to pass information between them.

14.2 Functions

Functions are similar to shaders except they can be called from shaders or other functions, but cannot be instanced from the RenderMan Interface. Functions in the Shading Language are also much like functions in C. For example, the following defines a function normalize that returns a unit vector in the direction V:

```
vector
normalize ( vector V )
{
    return V/length(V);
}
```

Function calls use the traditional C syntax, except for the issues noted below.

Functions may return any of the basic types (float, color, point, vector, normal, matrix, string). Functions may not return arrays, though they make take arrays as parameters. Functions must have exactly one return statement, except for those functions declared with type **void**, which may not have any return statement.

Function parameters are passed *by reference*; in other words, parameters are not copied into private data areas of the function. Nevertheless, function parameters are not writable unless specified with the **output** keyword. For example:

```
void
normalize_inplace ( output vector V )
{
    V = V/length(V);
}
```

Functions in the Shading Language cannot be called recursively.

Functions may be polymorphic — that is, you may define multiple functions with the same name, as long as they take different types or numbers of arguments. Functions may even be polymorphic based on return type. For example:

```
float snoise (point p)
{
    return 2 * (float noise(p)) - 1;
}

float snoise (float s, float t)
{
    return 2 * (float noise(s, t)) - 1;
}

vector snoise (point p)
{
```

```
    return 2 * (vector noise(p)) - 1;
}
```

Functions may be declared anyplace that it is legal to declare a variable or have any other language statement, including inside the body of a shader, inside another function, or inside any other code block. A function may only be called within the lexical scope in which it is declared, and only after its declaration (the same rules that apply to using a declared variable). The only data that functions may access are those passed as parameters, except for variables declared with the **extern** keyword, which indicates that the variable is in an outer scope rather than that local storage should be allocated. For example:

```
surface mysurf (float Knoise = 0.5;) /* parameter */
{
    color shadernoise (point p)
    {
        extern float Knoise;
        return Knoise * color noise (transform (”shader”, p));
    }
    /* code for the shader */
    Ci = shadernoise(P) * diffuse(faceforward(normalize(N),I));
}
```

Section 15

BUILT-IN FUNCTIONS

The following built-in functions are provided in the Shading Language.

15.1 Mathematical Functions

The following library of math functions is provided. This library includes most of the routines normally found in the standard C math library as well as functions for interpolation and computing derivatives.

The following mathematical functions are provided:

float **PI** = 3.14159... ;

float **radians** (float degrees)
float **degrees** (float radians)

float **sin** (float a)
float **asin** (float a)

float **cos** (float a)
float **acos** (float a)

float **tan** (float a)
float **atan** (float yoverx), **atan**(float y, x)

> The predefined float constant PI is the value of π. The function radians converts from degrees to radians; and conversely, the function degrees converts from radians to degrees. sin, cos, and tan are the standard trigonometric functions of radian arguments. asin returns the arc sine in the range $-\pi/2$ to $\pi/2$. acos returns the arc cosine in the range 0 to π. atan with one argument returns the arc tangent in the range $-\pi/2$ to $\pi/2$. atan with two arguments returns the arc tangent of y/x in the range $-\pi$ to π.

float **pow**(float x, y)
float **exp**(float x)
float **sqrt**(float x)
float **inversesqrt**(float x)
float **log**(float x), **log**(float x, base)

These functions compute power and inverse power functions. pow returns x^y, exp returns e^x. sqrt returns the positive square root of *x*, and inversesqrt(x) returns 1/sqrt(x). log with one argument returns the natural logarithm of *x* (x = log(exp(x))). log with two arguments returns the logarithm in the specified base (x = log(pow(*base*, x), *base*)).

float **mod** (float a, b)
float **abs** (float x)
float **sign** (float x)

mod returns a value greater than 0 and less than or equal to *b* such that mod*(a,b)* = *a - n*b* for some integer n. abs returns the absolute value of its argument and sign returns -1 if its argument is negative, 1 if its argument is positive, and 0 if its argument is zero.

type **min** (*type* a, b, ...)
type **max** (*type* a, b, ...)
type **clamp** (*type* a, min, max)

min takes a list of two or more arguments of identical type and returns the argument with minimum value; max returns the argument with maximum value. clamp(*a*,*min*,*max*) returns *min* if *a* is less than *min*, *max* if *a* is greater than *max*; otherwise it returns *a*. 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 (e.g., separately for x, y, and z).

float **mix** (float 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)
color **mix** (color x, y; float alpha)

mix returns $x \cdot (1 - \alpha) + y \cdot \alpha$, that is, it performs a linear blend between values *x* and *y*. The types of *x* and *y* must be identical, but 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 (e.g., separately for x, y, and z).

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

floor returns the largest integer (expressed as a float) not greater than *x*. ceil returns the smallest integer (expressed as a float) not smaller than *x*. round returns the integer closest to *x*.

float **step** (float min, value)
float **smoothstep** (float min, max, value)

step returns 0 if *value* is less than *min*; otherwise it returns 1. smoothstep returns 0 if *value* is less than *min*, 1 if *value* is greater than or equal to *max*, and performs a smooth Hermite interpolation between 0 and 1 in the interval *min* to *max*.

float **filterstep** (float edge, s1; ...parameterlist...)
float **filterstep** (float edge, s1, s2; ...parameterlist...)

This function provides an analytically antialiased step function. In its two argument form, it takes parameters identical to step, but returns a result which is filtered over the area of the surface element being shaded. If the optional *s2* parameter is provided, the step function is filtered in the range between the two values. This low-pass filtering is similar to that done for texture maps (for reference, see Section 15.7, Texture Mapping Functions). The *parameterlist* provides control over the filter function, and may include the following parameters: "width" (aka "swidth"), the amount to "overfilter" is s; "filter", the name of the filter kernel to apply. The filter may be any of the following: "box", "triangle", "catmull-rom", or "gaussian". The default filter is "catmull-rom".

float **spline** ([string basis;] float value; float f1, f2, ..., fn, fn1)
float **spline** ([string basis;] float value; float fvals[])
color **spline** ([string basis;] float value; color c1, c2, ..., cn, cn1)
color **spline** ([string basis;] float value; color cvals[])
point **spline** ([string basis;] float value; point p1, p2, ..., pn, pn1)
point **spline** ([string basis;] float value; point pvals[])
vector **spline** ([string basis;] float value; vector v1, v2, ..., vn, vn1)
vector **spline** ([string basis;] float value; vector vvals[])

spline fits a spline to the control points given. At least four control points must always be given. If *value* equals 0, *f2* (or *c2*, *p2*, *v2*) is returned; if *value* equals 1, *fn* (or *cn*, *pn*, *vn*) is returned. The type of the result depends on the type of the arguments.

If the first argument to spline is a string, it is interpreted as the name of a cubic spline basis function. The basis may be any one of "catmull-rom", "bezier", "bspline", "hermite", or "linear". If the optional basis name is not supplied, "catmull-rom" is assumed, resulting in the control points being interpolated by a Catmull-Rom spline. In the case of Bezier and Hermite spline bases, the number of spline knots must be 4n+3 and 4n+2, respectively. In the case of linear spline basis, the first and last knot are unused, but are nonetheless required to maintain consistency with the cubic bases.

For all spline types, an array of values may be used instead of a list of values to specify the control points of a spline.

float **Du** (float p), **Dv** (float p), **Deriv** (float num; float den)
color **Du** (color p), **Dv** (color p), **Deriv** (color num; float den)
vector **Du** (point p), **Dv** (point p), **Deriv** (point num; float den)
vector **Du** (vector p), **Dv** (vector p), **Deriv** (vector num; float den)

These functions compute the derivatives of their arguments. The type returned depends on the type of the first argument. Du and Dv compute the derivatives in the *u* and *v* directions, respectively. Deriv computes the derivative of the first argument with respect to the second argument.

The actual change in a variable is equal to its derivative with respect to a surface parameter times the change in the surface parameter. Thus, assuming forward differencing,

```
function(u+du)-function(u) = Du( function(u) ) * du;
function(v+dv)-function(v) = Dv( function(v) ) * dv;
```

float **random**()
color **random** ()
point **random** ()

random returns a float, color, or point whose components are a random number between 0 and 1.

float **noise** (float v), **noise** (float u, v), **noise** (point pt) **noise** (point pt, float t)
color **noise** (float v), **noise** (float u, v), **noise** (point pt), **noise** (point pt, float t)
point **noise** (float v), **noise** (float u, v), **noise** (point pt), **noise** (point pt, float t)
vector **noise** (float v), **noise** (float u, v), **noise** (point pt), **noise** (point pt, float t)

noise returns a value which is a pseuodrandom function of its arguments; its value is always between 0 and 1. The domain of this noise function can be 1-D (one float), 2-D (two floats), 3-D (one point), or 4-D (one point and one float). These functions can return any type. The type desired is indicated by *casting* the function to the type desired. The following statement causes noise to return a color.

```
c = 2 * color noise(P);
```

float **pnoise** (float v, uniform float period),
pnoise (float u, v, uniform float uperiod, uniform float vperiod),
pnoise (point pt, uniform point pperiod),
pnoise (point pt, float t, uniform point pperiod, uniform float tperiod)
color **pnoise** (float v, uniform float period),
pnoise (float u, v, uniform float uperiod, uniform float vperiod),
pnoise (point pt, uniform point pperiod),
pnoise (point pt, float t, uniform point pperiod, uniform float tperiod)
point **pnoise** (float v, uniform float period),
pnoise (float u, v, uniform float uperiod, uniform float vperiod),
pnoise (point pt, uniform point pperiod),
pnoise (point pt, float t, uniform point pperiod, uniform float tperiod)
vector **pnoise** (float v, uniform float period),
pnoise (float u, v, uniform float uperiod, uniform float vperiod),
pnoise (point pt, uniform point pperiod),
pnoise (point pt, float t, uniform point pperiod, uniform float tperiod)

pnoise returns a value similar to noise with the same arguments, however, the value returned by pnoise is periodic with period *period* (or *pperiod*, *tperiod*, etc.). That is, pnoise(v, p) == pnoise(v+p, p). The *period* parameters must be uniform and have an integer value (if it is a float expression), or lie on the integer lattice (if it is a point expression).

float **cellnoise** (float v), **cellnoise** (float u, v),
    cellnoise (point pt) **cellnoise** (point pt, float t)
color **cellnoise** (float v), **cellnoise** (float u, v),
    cellnoise (point pt), **cellnoise** (point pt, float t)
point **cellnoise** (float v), **cellnoise** (float u, v),
    cellnoise (point pt), **cellnoise** (point pt, float t)
vector **cellnoise** (float v), **cellnoise** (float u, v),
    cellnoise (point pt), **cellnoise** (point pt, float t)

cellnoise returns a value which is a pseuodrandom function of its arguments. Its domain can be 1-D (one float), 2-D (two floats), 3-D (one point), or 4-D (one point and one float). Its return value is uniformly distributed between 0 and 1, has constant value between integer lattice points, and is discontinuous at integer locations. This is useful if you are dividing space into regions ("cells") and want a different (repeatable) random number for each region. It is considerably cheaper than calling noise, and thus is preferable if you have been using noise simply to generate repeatable random sequences. The type desired is indicated by *casting* the function to the type desired.

15.2 Geometric Functions

Geometric functions provide a kernel of useful geometric operations. Most of these functions are most easily described by just giving their implementation.

float **xcomp**(*ptype* P)
float **ycomp**(*ptype* P)
float **zcomp**(*ptype* P)
void **setxcomp**(output *ptype* P; float x)
void **setycomp**(output *ptype* P; float y)
void **setzcomp**(output *ptype* P; float z)

These functions get and set individual components of points, vectors, or normals.

```
float
length( vector V )
{
    return sqrt(V.V);
}
```

Return the length of a vector.

```
vector
normalize( vector V )
{
    return V/length(V);
}
```

Return a unit vector in the direction of *V*.

```
float
distance( point P1, P2 )
{
      return length(P1-P2);
}
```

Return the distance between two points.

float **ptlined** (point Q, P1, P2)

Returns the minimum perpendicular distance between the point *Q* and the line segment that passes from the point *P0* to the point *P1* (not the infinite line which passes through *P0* and *P1*).

float **rotate** (point Q; float angle; point P1, P2)

Rotate a point *Q* by *angle* radians about the axis which passes through the points *P0* and *P1*.

```
float
area( point P )
{
      return length( Du(P)*du ^ Dv(P)*dv);
}
```

Return the differential surface area.

```
vector
faceforward( vector N, I [, Nref])
{
      return sign(-I.Ng) * N;
}
```

Flip *N* so that it faces in the direction opposite to *I*, from the point of view of the current surface element. The surface element's point of view is the geometric normal *Ng*, unless *Nref* is supplied, in which case it is used instead.

```
vector
reflect( vector I, N )
{
      return I - 2*(I.N)*N;
}
```

Return the reflection vector given an incident direction *I* and a normal vector *N*.

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

```
{
    float IdotN = I.N;
    float k = 1 - eta*eta*(1 - IdotN*IdotN);
    return k < 0 ? (0,0,0) : eta*I - (eta*IdotN + sqrt(k))*N;
}
```

Return the transmitted vector given an incident direction *I*, the normal vector *N* and the relative index of refraction *eta*. *eta* is the ratio of the index of refraction in the volume containing the incident vector to that of the volume being entered. This vector is computed using Snell's law. If the returned vector has zero length, then there is no transmitted light because of total internal reflection.

fresnel(vector I, N; float eta; output float Kr, Kt; [output vector R, T])

Return the reflection coefficient *Kr* and refraction (or transmission) coefficient *Kt* given an incident direction *I*, the surface normal *N*, and the relative index of refraction *eta*. *eta* is the ratio of the index of refraction in the volume containing the incident vector to that of the volume being entered. These coefficients are computed using the Fresnel formula. Optionally, this procedure also returns the reflected (*R*) and transmitted (*T*) vectors. The transmitted vector is computed using Snell's law.

point **transform**(string tospace; point p)
point **transform**(string fromspace, tospace; point p)
point **transform**(matrix m; point p)
point **transform**(string fromspace; matrix m; point p)

vector **vtransform**(string tospace; vector v)
vector **vtransform**(string fromspace, tospace; vector v)
vector **vtransform**(matrix m; vector v)
vector **vtransform**(string fromspace; matrix m; vector v)

normal **ntransform**(string tospace; normal n)
normal **ntransform**(string fromspace, tospace; normal n)
normal **ntransform**(matrix m; normal n)
normal **ntransform**(string fromspace; matrix m; normal n)

The transform function transforms the point *p* from the coordinate system *fromspace* to the coordinate system *tospace*. If *fromspace* is absent, it is assumed to be the "current" coordinate system. A transformation matrix may be given instead of a *tospace* name. The vtransform and ntransform functions perform the equivalent coordinate system transformations on vectors and normals, respectively.

float **depth**(point P)

Return the depth of the point *P* in camera coordinates. The depth is normalized to lie between 0 (at the near clipping plane) and 1 (at the far clipping plane).

```
point
calculatenormal( point P )
{
    return Du(P) ^ Dv(P);
}
```

Return surface normal given a point on the surface. This function is normally called after a displacement. For example:

```
P += displacement * N;
N = calculatenormal( P );
```

15.3 Color Functions

Several functions exist which operate on colors.

float **comp** (color c; float index)
void **setcomp** (output color c; float index, value)

These functions get and set individual color components. The index values are 0-based (e.g., the green channel of an RGB triple is component 1).

```
color mix( color color0, color1; float value )
{
    return (1-value)*color0 + value*color1;
}
```

Return an interpolated color value.

color **ctransform** (string tospace; color C)
color **ctransform** (string fromspace, tospace; color C)

Transform the color *C* from the color representation *fromspace* to the color representation *tospace*. If *fromspace* is absent, it is assumed to be "rgb".

15.4 Matrix Functions

float **comp** (matrix m; float row, column)
void **setcomp** (output matrix m; float row, column, value)

These functions get and set individual components of a matrix. Strict runtime bounds checking will be performed on *row* and *column* to ensure that they fall into the range 0...3.

float **determinant** (matrix m)

Returns the determinant of matrix *m*.

matrix **translate** (matrix m; vector t)
matrix **rotate** (matrix m; float angle; vector axis)
matrix **scale** (matrix m; point s)

Postconcatenates simple transformations onto the matrix *m*. These functions are similar to the RI functions **RiTranslate**, **RiRotate** and **RiScale**, except that the rotation angle in rotate() is in radians, not in degrees as with **RiRotate**.

15.5 String Functions

string **concat** (string a, b, ...)

Concatenates two or more strings into a single string.

void **printf** (string pattern, val1, val2,..., valn)

Print the values of the specified variables on the standard output stream of the renderer (much like the printf function in C. *pattern* uses "%f", "%p", "%c", "%m", and "%s" to indicate float, point, color, matrix, and string, respectively. A vector or normal may also be printed using "%p".

string **format** (string pattern, val1, val2,..., valn)

Does a formatted string creation under the control of *pattern*. This function is similar to the C function sprintf(). As with the Shading Language printf function, "%f", "%p", "%c", "%m", and "%s" to indicate float, point, color, matrix, and string, respectively.

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

Does a string pattern match on *subject*. Returns 1.0 if the *pattern* exists anywhere within *subject*, and 0.0 if the pattern does not exist within *subject*. *pattern* can be any regular expression, as described in the POSIX manual page on regex()(3X), with the following exception: the $n notation does not work, as there are no return values from this function. 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.

15.6 Shading and Lighting Functions

In this section, built-in shading and lighting functions are defined.

color
ambient()

ambient returns the total amount of ambient light incident upon the surface. An ambient light source is one in which there is no directional component, that is, a light which does not have an illuminate or a solar statement.

```
color
diffuse( normal N )
{
        color C = 0;
        illuminance( P, N, PI/2 )
                C += Cl * normalize(L).N;
        return C;
}
```

diffuse returns the diffuse component of the lighting model. *N* is a unit-length surface normal.

```
color
specular( normal N; vector V; float roughness )
{
    color C = 0;
    illuminance( P, N, PI/2 )
        C += Cl * specularbrdf (normalize(L), N, V, roughness);
    return C;
}
```

specular returns the specular component of the lighting model, using an implementation-dependent specularbrdf. *N* is the unit-length normal to the surface. *V* is a unit-length vector from a point on the surface towards the viewer.

color **specularbrdf** (vector L; normal N; vector V; float roughness)

Returns the specular attenuation of light coming from the direction *L*, reflecting toward direction *V*, with surface normal *N* and *roughness* rough. All of *L*, *V*, and *N* are assumed to be of unit length. This is the same reflection model calculation found inside the illuminance loop of the specular function. This allows users to write an illuminance loop that reproduces the functionality of the specular() function, even if the renderer has an implementation-specific formula for built-in specular reflection. Here is an example implementation of specularbrdf:

```
color specularbrdf( vector L, N, V; float roughness )
{
    vector H = normalize(L+V);
    return pow (max (0, N.H), 1/roughness);
}
```

```
color
phong( normal N; vector V; float size )
{
    color C = 0;
    vector R = reflect( -normalize(V), normalize(N) );
    illuminance( P, N, PI/2 ) {
        vector Ln = normalize(L);
        C += Cl * pow(max(0.0,R.Ln), size);
    }
    return C;
}
```

phong implements the Phong specular lighting model.

color **trace**(point P, point R)

trace returns the incident light reaching a point *P* from a given direction *R*. If a particular implementation does not support the *Ray Tracing* capability, and cannot compute the incident light arriving from an arbitrary direction, trace will return 0 (black).

15.7 Texture Mapping Functions

Texture maps are images that are mapped onto the surface of a geometric primitive. The RenderMan Interface supports three primitive types of texture access: basic texture maps (via texture), environment maps (via environment), and shadow or z-buffer maps (via shadow). Texture maps are accessed using two-dimensional coordinates and return floats or colors. Environment maps are accessed using a direction and return floats or colors. Shadow maps are accessed using points and return floats.

For two-dimensional access (texture), the texture coordinates default to the texture coordinates attached to the surface, (s,t). These default texture coordinates are equal to the surface parameters, the *current texture coordinates*, or the texture coordinates passed with the geometric primitive. Texture coordinates can also be computed in the Shading Language. This generality allows for many different types of coordinate mappings. Images stored in various map projections can be accessed by computing the map projection given a point on a sphere. This allows basic texture maps to be used as environment maps. Images can also be mapped onto surfaces using a two step process. First the surface of the geometric primitive is mapped to the surface of a parametric primitive, such as a plane or cylinder, and then the parameters of this primitive are used as the texture coordinates. This is sometimes called a *decal* projection.

For three-dimensional access (environment and shadow), the texture coordinates must always be explicitly specified.

There is no restriction on how texture map values are used in the Shading Language. For example, displacement mapping can be performed by moving a point on the surface in the direction of the normal by the amount returned by a basic texture map. Transparency mapping differs from color mapping only in which variable, either Os or Cs, the texture is assigned to. There is also, in principle, no limit on the number of texture accesses per shader or the number of texture maps per shader or per frame.

Texture maps are created in advance from image data via three types of **MakeTexture** procedures that are defined as part of the RenderMan Interface. These are described in Part I in the section on Texture Map Utilities. **RiMakeTexture** creates a texture map for access via texture. **RiMakeCubeFaceEnvironment** and **RiMakeLatLongEnvironment** create an environment map for access via environment. **RiMakeShadow** creates a shadow map for access via shadow. A texture file may contain several *channels* of information and have any horizontal or vertical resolution. This information is normally inherited from the image from which the texture is made. The s coordinate is assigned to the horizontal direction with increasing values moving right. The t coordinate is assigned to the vertical direction with increasing values moving down. These coordinates are normalized to lie in the range 0 to 1 so that changing the resolution of the texture map has no effect on the shaders that access the texture map. When a texture map is created, the wrap mode is also specified. The wrap mode controls what values are returned if the texture coordinates fall outside the unit square. Allowed wrap modes are: *periodic*, *black* and *clamp*. *periodic* causes the texture data to tile the plane, *black* causes accesses outside the unit square to return the value 0, and *clamp* causes the texture coordinates to be clamped to the closest point on the unit square and the texture value associated with that point to be returned.

The texture access functions normally pass the texture map through a low-pass filter to

prevent aliasing. If one set of texture coordinates is given to the access function, the texture will be filtered over the area of the surface element being shaded (see the Shading Rate section in Part I). Four sets of texture coordinates can also be given to the access procedure, in which case the texture is filtered over the quadrilateral determined by those four points. The quality of texture antialiasing is controlled in the same way as spatial antialiasing. Parameters control how true the answer is, the effective number of samples used before filtering, and the type and width of the filter used. For this to be done properly (since texture maps are normally prefiltered), these filtering parameters are best given to the appropriate RiMake... procedure. For flexibility, however, they can also be changed at access time. Table 15.1, Texture Access Parameters gives the standard parameters to all the texture access functions; particular implementations may have additional parameters. If a parameter is encountered by an implementation that does not support its functionality, it should be ignored.

Name	Type	Description
"blur"	varying float	Specifies an additional area to be added to the texture area filtered in both the *s* and *t* directions, expressed in units of texture coordinates. A value of 1.0 would request that the entire texture file be blurred into the result. A value of 0.001 would request that one extra texture pixel be added in the case of a one-thousand by one-thousand texture file.
"sblur"	varying float	Specifies "blur" individually in the *s* direction.
"tblur"	varying float	Specifies "blur" individually in the *t* direction.
"width"	uniform float	This value multiplies the width of the area being filtered over in both the *s* and *t* directions. A value of 0 effectively turns off texture antialiasing. The default value is 1.0.
"swidth"	uniform float	Specifies "width" individually in the *s* direction.
"twidth"	uniform float	Specifies "width" individually in the *t* direction.
"filter"	uniform string	Specifies the name of the filter to use for filtering over an area of the texture. The default is "box". Individual implementations may allow additional filters.
"fill"	uniform float	Specifies the value to be provided for channels requested that are not present in the texture file. This is most often useful for a shader that knows how to use a texture containing an alpha channel. If no alpha channel is present in the texture file, the texture system quietly provides a default value of 0.0 to requests for the alpha value resulting in a completely tranparent texture. Such a shader could provide its own "fill" value of 1.0 so that textures without an alpha channel would be opaque by default.

Table 15.1: Texture and Environment Map Access Parameters

15.7.1 Basic texture maps

Basic texture maps return either floats or colors.

float **texture** (string name[*channel*]; [*texture coordinates*,] [*parameterlist*])
color **texture** (string name[*channel*]; [*texture coordinates*,] [*parameterlist*])

where *texture coordinates* is one of the following:

```
float s, t;
float s1,t1, s2,t2, s3,t3, s4,t4;
```

Return the filtered texture value. The cast before the function determines the type returned, either a float or a color. The *name* is the name of the texture map created using **RiMakeTexture**. The *channel* selector is optional; if it is not present, the brackets are also omitted and channel 0 is assumed. *channel* selects the starting channel in the texture. The number of channels returned depends on whether the texture is interpreted as a float or a color. *texture coordinates* are also optional. If present they consist either of a single 2-D coordinate or four 2-D coordinates. If no texture coordinates are given, the current values of (s,t) are used. *parameterlist* is a list of name-value pairs that allow greater control over texture access.

Some examples of the use of this function are:

```
c = texture( "logo" [0] );
c = color texture ( "logo" );
c = color texture ( "logo", 2*s, 4*t );
```

In the first two cases, the texture coordinates are the current values of the predefined variables (s,t).

15.7.2 Environment maps

float **environment** (string *name*[*channel*]; *texture coordinates*, [*parameterlist*])
color **environment** (string *name*[*channel*]; *texture coordinates*, [*parameterlist*])

where *texture coordinates* is one of the following:

```
point R;
point R1, R2, R3, R4;
```

Return the filtered texture value from an environment map. The cast before the function determines the type returned, either a float or a color. The *name* is the name of the texture map created using RiMake...Environment. The *channel* selector is optional; if it is not present, the brackets are also omitted and channel 0 is assumed. *channel* selects the starting channel in the texture. The number of channels returned depends on whether the texture is interpreted as a float or a color. This function expects either a single texture coordinate or four texture coordinates. These are points that are used to define a direction in space. The length of this vector is unimportant. *parameterlist* is a list of name-value pairs which allow greater control over texture access.

15.7.3 Shadow depth maps

Shadow depth maps are z-buffer images as seen from a particular view point. Normally a shadow map is associated with a light source and represents a depth buffer rendered from the point of view of the light source. The texture coordinate of a shadow map is a point. The value returned is the fraction of points on the shaded surface that are farther from the light than the surface recorded in the depth map. A value of 1 indicates that the surface is completely in shadow and a value of 0 indicates that the surface is completely illuminated by the light source.

float **shadow**(string *name*[*channel*]; *texture coordinates*[, *parameterlist*])

where *texture coordinates* is one of the following:

```
point P;
point P1, P2, P3, P4;
```

Return the shadow value from a shadow depth map. The *name* is the name of the texture map created using **RiMakeShadow**. The *channel* selector is optional; if it is not present, the brackets are also omitted and channel 0 is assumed. *channel* selects the starting channel in the texture. Only one channel of a shadow map is ever accessed. *texture coordinates* are points in the coordinate system in which the depth map was created. *parameterlist* is a list of name-value pairs that allow greater control over texture access.

15.7.4 Getting Information About Texture Maps

float **textureinfo** (string texturename, dataname; output *type* variable)

Returns data that about a particular texture map, specified by the file name *texturename*. The *dataname* specifies the piece of information that will be returned in *variable*. The *dataname* is known to the renderer and its type and storage class match that of *variable*, the named data will be written into *variable* and textureinfo() will return a value of 1.0. If the data is unknown or the types do not match, *variable* will be unchanged and textureinfo() will return 0.0. Note that textureinfo data is always uniform. If *variable* is varying, the appropriate uniform-to-varying conversion will take place.

The standard data names supported by textureinfo are listed in Table 15.3. A particular implementation may support additional textureinfo queries.

15.8 Message Passing and Information Functions

float **atmosphere**(string paramname; output *type* variable)
float **displacement**(string paramname; output *type* variable)
float **lightsource**(string paramname; output *type* variable)
float **surface**(string paramname; output *type* variable)

Name	Type	Description
"blur"	varying float	Specifies an additional area to be added to the texture area filtered in both the s and t directions, expressed in units of texture coordinates. A value of 1.0 would request that the entire texture file be blurred into the result. A value of 0.001 would request that one extra texture pixel be added in the case of a one-thousand by one-thousand texture file.
"sblur"	varying float	Specifies "blur" individually in the s direction.
"tblur"	varying float	Specifies "blur" individually in the t direction.
"width"	uniform float	This value multiplies the width of the area being filtered over in both the s and t directions. A value of 0 effectively turns off texture antialiasing. The default value is 1.0.
"swidth"	uniform float	Specifies "width" individually in the s direction.
"twidth"	uniform float	Specifies "width" individually in the t direction.
"samples"	uniform float	The effective sampling rate when filtering.
"bias"	varying float	Specifies an additional bias to add to shadow depth lookups in order to prevent incorrect self-shadowing of objects.
"filter"	uniform string	Specifies the name of the filter to use for performing the percentage-closer filtering over an area of the shadow map. The default is "box". Individual implementations may allow additional filters.

Table 15.2: Shadow Map Access Parameters

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

(*) "viewingmatrix" and "projectionmatrix" are only available for shadow maps or other textures that were created by rendering an image, and assume a common world-space of all cameras.

Table 15.3: Data names known to the textureinfo function.

These functions access the value of the parameter named *paramname* of one of the shaders attached to the geometric primitive that is currently being shaded. If the appropriate shader exists, and a parameter named *paramname* exists in that shader, and the parameter is the same type as *variable*, then the value of that parameter is stored in *variable* and the function returns 1; otherwise, *variable* is unchanged and the function returns 0.

Note that if the data corresponding to *name* is uniform, but the variable *variable* is varying, the appropriate uniform-to-varying conversion will take place, however, the reverse case is considered failure.

The lightsource function is only available inside illuminance blocks and refers to the light being examined by the current iteration.

float **incident**(string paramname; output *type* variable)
float **opposite**(string paramname; output *type* variable)

These functions access the value of the volume shader parameter *paramname* that is stored in the volume shaders attached to geometric primitive surface. incident accesses values from the volume shader that describes the volume which contains the incident ray I. opposite accesses values from the volume shader that describes the volume on the other side of the surface. If the named variable exists and is of the correct type, the *value* is stored in value and the function returns 1; otherwise, *value* is unchanged and the function returns 0.

Note that if the data corresponding to *name* is uniform, but the variable *variable* is varying, the appropriate uniform-to-varying conversion will take place, however, the reverse case is considered failure.

float **attribute** (string name; output *type* variable)

Returns data that is part of the primitive's attribute state, either from individual RenderMan Interface calls that set attributes or from the **RiAttribute** call. The *name* specifies the piece of RenderMan Interface attribute state that will be returned in *variable*. If the data *name* is known to the renderer and its type and storage class match that of *variable*, the named data will be written into *variable* and attribute() will return a value of 1.0. If the data is unknown or the types do not match, *variable* will be unchanged and attribute() will return 0.0.

Note that if the data corresponding to *name* is uniform, but the variable *variable* is varying, the appropriate uniform-to-varying conversion will take place, however, the reverse case is considered failure.

The standard data names supported by attribute are listed in Table 15.4. A particular implementation may support additional attribute queries.

float **option** (string name; output *type* variable)

Returns data that is part of the renderer's global option state, either from individual RenderMan Interface calls that set options or from the **RiOption** call. The *name* specifies the piece of RenderMan Interface option state that will be returned in *variable*. If the data *name* is known to the renderer and its type and storage class match that of

Name	Type
"ShadingRate"	uniform float
"Sides"	uniform float
"Matte"	uniform float
"GeometricApproximation:motionfactor"	uniform float
"displacementbound:sphere" (*)	uniform float
"displacementbound:coordinatesystem" (*)	uniform string
"identifier:name"	uniform string

(*) Note that "displacementbound:sphere" does not return the value exactly as specified in the RIB file, but rather returns its length in the coordinate system returned by "displacementbound:coordinatesystem" (which currently always returns "camera").

Table 15.4: Data names known to the attribute function.

variable, the named data will be written into *variable* and option() will return a value of 1.0. If the data is unknown or the types do not match, *variable* will be unchanged and option() will return 0.0. Note that option data is always uniform. If *variable* is varying, the appropriate uniform-to-varying conversion will take place.

The standard data names supported by option are listed in Table 15.5. A particular implementation may support additional option queries.

Name	Type	Description
"Format"	uniform float [3]	x resolution, y resolution, pixel aspect ratio.
"DeviceResolution"	uniform float [3]	The resolution in x and y and the pixel aspect ratio. These are usually the three numbers passed to **RiFormat**, but may be different when **RiFrameAspectRatio** or **RiScreenWindow** are non-square.
"FrameAspectRatio"	uniform float	Frame aspect ratio.
"CropWindow"	uniform float [4]	Boundaries of the crop window.
"DepthOfField"	uniform float [3]	Fstop, focallength, focaldistance.
"Shutter"	uniform float [2]	Shutter open and close time.
"Clipping"	uniform float [2]	Near and far clip depths.

Table 15.5: Data names known to the option function.

float **rendererinfo** (string name; output *type* variable)

Returns data that about the renderer itself. The *name* specifies the piece of information that will be returned in *variable*. The the data *name* is known to the renderer and its type and storage class match that of *variable*, the named data will be written into *variable* and rendererinfo() will return a value of 1.0. If the data is unknown or the types do not match, *variable* will be unchanged and rendererinfo() will return 0.0. Note that renderer data is always uniform. If *variable* is varying, the appropriate uniform-to-varying conversion will take place.

The standard data names supported by rendererinfo are listed in Table 15.6. A particular implementation may support additional rendererinfo queries.

Name	Type	Description
"renderer"	uniform string	The brand name of the renderer.
"version"	uniform float [4]	Major, minor, release, and patch numbers.
"versionstring"	uniform string	The release numbers expressed as a string.

Table 15.6: Data names known to the rendererinfo function.

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

If no parameter is passed, returns the name of the shader that is currently running. If the *shadertype* is supplied, this function returns the name of the shader of the specified type that is bound to the geometric primitive which is being shaded. Acceptable values of *shadertype* are "surface", "displacement", "atmosphere", "lightsource", "interior", "exterior". If the surface being shaded does not have a shader bound to it which matches the type, this function will return the empty string ("").

Section 16

EXAMPLE SHADERS

16.1 Surface Shaders

Surface shaders inherit the surface variables of the surfaces to which they are attached. A surface shader should always set the output color Ci and optionally the output opacity Oi that is emitted in the direction -I. I is the direction of the ray incident to the surface. The length of this vector is equal to the distance between the origin of the ray and the point on the surface. Thus the actual origin of the ray is available as P-I.

16.1.1 Turbulence

The following surface shader implements a simple *turbulence* procedural texture. The shader computes the texture by adding various octaves of noise, weighting each octave by $1/f$, where f is the cutoff frequency of that octave. This texture is then used to modulate the opacity of the surface. The texture is generated in the named coordinate system "marble", which must have been established with by the use of an **RiCoordinateSystem** ("marble") call before the instantiation of the turbulence shader. Notice that after the opacity has been computed, it is multiplied into the color, so that the colors and opacities output by the shader are *premultiplied* for use by pixel compositors.

```
surface
turbulence ( float Kd=.8, Ka=.2 )
{
    float a, scale, sum ;
    float IdotN;
    point M; /* convert to texture coordinate system */
    M = transform( "marble", P );
    scale = 1;
    sum = 0;
    a = sqrt(area(M));
    while( a < scale ) {
        sum += scale * float noise(M/scale);
        scale *= 0.5;
    }
```

```
        Oi = sum;
        Ci = Cs * Oi * (Ka + Kd * I.N * I.N / (I.I * N.N) );
    }
```

16.1.2 Ray tracer

The following is a procedure to implement a Turner Whitted-style ray tracer.

```
surface
whitted(
        float eta = 1.5;        /* index of refraction */
        float Kr =.8;           /* reflective coefficient */
        float Kt =.2;           /* transmissive coefficient */
        float Ks =.2;           /* specular coefficient */
        float Kss = 2;          /* specular exponent */
        float eta = 1.5; )      /* index of refraction */
{
    normal Nn = faceforward(normalize(N), I);

    /* ambient term */
    Ci = Kd * ambient();

    /* diffuse and specular terms */
    illuminance( P, Nn, PI/2 ) {
        /* diffuse */
        Ci += Kd * Cl * (L . Nn);
        /* specular */
        vector H = normalize(normalize(L)+I);
        Ci += Ks * Cl * pow(max(0,0, Nn.H), Kss);
    }

    /* reflection */
    Ci += Ks * trace( reflect( I, Nn ) );

    /* transmittance */
    vector T = refract( I, Nn, (N.I)<0 ? eta : 1/eta );
    if ( length(T) != 0.0 )
        Ci += Kt * trace( T );
}
```

16.2 Light Sources

There are several types of light source shaders, distinguished by their directional properties. The directional properties of light sources depend on whether the sources execute a solar or an illuminate statement. Light source shaders without explicit illuminate or solar

statements are assumed to be non-directional, or *ambient*. The total amount of ambient light incident on a surface is normally returned to a surface shader through ambient. A solar statement indicates that the light source is a directional light source, while an illuminate statement indicates that the light source is a local light source. Local light sources have a position. The position can be a property of the shader or can be inherited from a surface. If the light source is attached to a geometric primitive the light source is an area light source.

Light sources set Cl inside a solar or illuminate block unless they are defining an ambient light. Inside these blocks the direction L points towards the surface. This variable is available so that light source output intensities can be directional. If the light source has a position, the length of L is the distance from the light source to the surface being shaded.

For example, consider the following light source:

```
light
phong(
        float intensity = 1.0;
        color color = 1;
        float size = 2.0;
        point from = point "shader" (0,0,0);
        point to = point "shader" (0,0,1); )
{
    uniform vector R = normalize(to-from);

    solar( R, PI/2 )
        Cl = intensity * color * pow( R.L/length(L), size );
}
```

The Phong shading model can be interpreted to be a procedural directional light source. The light source has a direction *R* and a *size* parameter that controls its fall-off. The solar statement specifies that the light source casts light in the forward facing hemisphere.

An environment background light source would be specified as follows:

```
light
reflection( string texturename = ""; float intensity = 1.0 )
{
    solar()
        Cl = intensity * color environment( texturename, -L );
}
```

The solar statement implies the light is cast from infinity in all directions. The color of the light is given by the environment map.

16.3 Volume Shader

Volume shaders change Ci and Oi due to volumetric scattering, self-luminosity, and attenuation. A volume shader is called once per ray so it should explicitly integrate along the

path of the ray.

The input Ci and Oi are the colors and opacities at the point P. The volume shader should set the color and opacity that result at the point P-I.

16.4 Displacement Shaders

Displacement shaders move the position P of a surface. After a point of the surface is moved, the normals should be recalculated with calculatenormal unless the new normals can be computed as part of the displacement.

The following shader places a sinusoidal bump on a surface.

```
displacement
ripple( float amplitude = 1.0, float wavelength = 0.25 )
{
    P += N * amplitude * sin( 2*PI*(s / wavelength) );
    N = calculatenormal(P);
}
```

16.5 Imager Shaders

Imager shaders change the value of Ci and Oi. The exposure and quantization process specified in the section on Displays in Part I could be specified as the following imager:

```
imager
exposure( float gain=1.0, gamma=1.0, one = 255, min = 0, max = 255 )
{
    Ci = pow( gain * Ci, 1/gamma );
    Ci = clamp( round( one * Ci ), min, max );
    Oi = clamp( round( one * Oi ), min, max );
}
```

Appendix A

STANDARD RENDERMAN INTERFACE SHADERS

In this section the required RenderMan Interface shaders are defined.

A.1 Null Shader

This shader does nothing and is intended to be a placeholder if no action is to be performed. There is a null shader for every class of shader.

A.2 Surface Shaders

A.2.1 Constant surface

```
surface
constant()
{
    Oi = Os;
    Ci = Os * Cs;
}
```

A.2.2 Matte surface

```
surface
matte(
        float Ka = 1;
        float Kd = 1;)
{
    normal Nf = faceforward(normalize(N), I);
    Oi = Os;
    Ci = Os * Cs * (Ka*ambient() + Kd*diffuse(Nf));
}
```

A.2.3 Metal surface

```
surface
metal(
        float Ka = 1;
        float Ks = 1;
        float roughness =.1;)
{
    normal Nf = faceforward(normalize(N), I);
    vector V = -normalize(I);
    Oi = Os;
    Ci = Os * Cs * (Ka*ambient()+Ks*specular(Nf, V, roughness));
}
```

A.2.4 Shiny metal surface

```
surface
shinymetal(
        float Ka = 1;
        float Ks = 1;
        float Kr = 1;
        float roughness = .1;
        string texturename = ""; )
{
    normal Nf = faceforward(normalize(N), I);
    vector V = -normalize(I);
    vector D = reflect(I, normalize(Nf));
    D = vtransform("current", "world", D);
    Oi = Os;
    Ci = Os * Cs * (Ka*ambient() + Ks*specular(Nf, V, roughness)
            + Kr*color environment(texturename, D));
}
```

If the Environment Mapping capability is not supported by a particular renderer implementation, the shinymetal surface shader operates identically to the metal shader.

A.2.5 Plastic surface

```
surface
plastic(
        float Ka = 1;
        float Kd =.5;
        float Ks =.5;
        float roughness =.1;
        color specularcolor = 1;)
{
```

```
    normal Nf = faceforward(normalize(N), I);
    vector V = -normalize(I);
    Oi = Os;
    Ci = Os * (Cs * (Ka*ambient() + Kd*diffuse(Nf)) +
            specularcolor * Ks*specular(Nf, V, roughness) );
}
```

A.2.6 Painted plastic surface

```
surface
paintedplastic(
        float Ka = 1;
        float Kd = .5;
        float Ks = .5;
        float roughness = .1;
        color specularcolor = 1;
        string texturename = "";)
{
    normal Nf = faceforward(normalize(N), I);
    vector V = -normalize(I);
    Oi = Os;
    Ci = Os * (Cs * color texture(texturename) *
        (Ka * ambient() + Kd * diffuse(Nf)) +
        specularcolor * Ks * specular(Nf, V, roughness));
}
```

If the *Texture Mapping* capability is not supported by a particular renderer implementation, the paintedplastic surface shader operates identically to the plastic shader.

A.3 Light Source Shaders

A.3.1 Ambient light source

```
light
ambientlight(
        float intensity = 1;
        color lightcolor = 1;)
{
    Cl = intensity * lightcolor;
}
```

A.3.2 Distant light source

```
light
```

```
distantlight(
        float intensity = 1;
        color lightcolor = 1;
        point from = point ”shader” (0,0,0);
        point to = point ”shader” (0,0,1);
{
    solar(to-from, 0.0)
        Cl = intensity * lightcolor;
}
```

A.3.3 Point light source

```
light
pointlight(
        float intensity = 1;
        color lightcolor = 1;
        point from = point ”shader” (0,0,0); )
{
    illuminate(from)
        Cl = intensity * lightcolor / L.L;
}
```

A.3.4 Spotlight source

```
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;)
{
    float atten, cosangle;
    uniform vector A = (to - from)/length(to-from);
    illuminate(from, A, coneangle) {
        cosangle = L . A / length(L);
        atten = pow(cosangle, beamdistribution) / L.L;
        atten *= smoothstep(cos(coneangle),
                cos(coneangle - conedeltaangle), cosangle);
        Cl = atten * intensity * lightcolor;
    }
}
```

A.4 Volume Shaders

A.4.1 Depth cue shader

```
volume
depthcue(
        float mindistance = 0, maxdistance = 1;
        color background = 0;)
{
    float d;
    d = clamp((depth(P) - mindistance) /
            (maxdistance - mindistance), 0.0, 1.0);
    Ci = mix(Ci, background, d);
    Oi = mix(Oi, color(1, 1, 1), d);
}
```

A.4.2 Fog shader

```
volume
fog(float distance = 1; color background = 0;)
{
    float d;
    d = 1 - exp(-length(I) / distance);
    Ci = mix(Ci, background, d);
    Oi = mix(Oi, color(1, 1, 1), d);
}
```

A.5 Displacement Shaders

A.5.1 Bumpy shader

```
displacement
bumpy(
        float Km = 1;
        string texturename = "";)
{
    float amp = Km * float texture(texturename, s, t);
    P += amp * normalize(N);
    N = calculatenormal(P);
}
```

A.6 Imager Shaders

A.6.1 Background shader

```
imager
background( color background = 1; )
{
    Ci += (1-alpha) * background;
    Oi = 1;
    alpha = 1;
}
```

Appendix B

RENDERMAN SHADING LANGUAGE SYNTAX SUMMARY

This summary of the Shading Language syntax is intended more for aiding comprehension than as an exact description of the language.

B.1 Declarations

Shading Language source files consist of definitions:

definitions:
    shader_definition
    function_definition

shader_definition:
    shader_type identifier ([formals]) { *statements* }

function_definition:
    [type] identifier ([formals]) { *statements* }

shader_type:
    light
    surface
    volume
    displacement
    imager

formals:
    formal_variable_definitions
    formals ; formal_variable_definitions

formal_variable_definitions:
    [outputspec] typespec def_expressions

variables:

variable_definitions ;
variables variable_definitions ;

variable_definitions:
[externspec] typespec def_expressions

typespec:
[detail] type

def_expressions:
def_expression
def_expressions , def_expression

def_expression:
identifier [def_init]

def_init:
= expression

type:
float
string
color
point
vector
normal
matrix
void

detail:
varying
uniform

outputspec:
output

externspec:
extern

B.2 Statements

statements:
statements statement

statement:
variable_definitions ;
assignexpression ;
procedurecall ;
return *expression ;*
loop_modstmt ;
if *relation statement*
if *relation statement* else *statement*
loop_control statement

loop_control:
while *relation*
for *(expression ; relation ; expression)*
solar *([expressionlist])*
illuminate *([expressionlist])*
illuminance *([expressionlist])*

loop_modstmt:
loop_mod [integer]

loop_mod:
break
continue

B.3 Expressions

The basic expressions are:

expressionlist:
expression [, expressionlist]

expression:
primary
expression binop expression
- expression
relation ? *expression* : *expression*
typecast expression

primary:
number
stringconstant
texture
identifier
identifier [arrayindex]
procedurecall
assignexpression

triple
sixteentuple

arrayindex:
[*expression*]

triple:
(*expression* , *expression* , *expression*)

sixteentuple:
(*expression* , *expression* , *expression* , *expression* ,
expression , *expression* , *expression* , *expression* ,
expression , *expression* , *expression* , *expression* ,
expression , *expression* , *expression* , *expression*)

typecast:
float
string
color *[spacetype]*
point *[spacetype]*
vector *[spacetype]*
normal *[spacetype]*
matrix *[spacetype]*

spacetype:
stringconstant

relation:
(*relation*)
expression relop expression
relation logop relation
! *relation*

assignexpression:
identifier asgnop expression
identifier [arrayindex] asgnop expression

procedurecall:
identifier (*[proc_arguments]*)

proc_arguments:
expression
proc_arguments , *expression*

texture:
texture_type (*texture_filename [channel] [texture_arguments]*)

texture_type:

texture
environment
shadow

texture_filename:
expression

channel:
[*expression*]

texture_arguments:
, *expression*
texture_arguments , *expression*

The primary-expression operators

( )

have highest priority and group left-to-right. The unary operators

- !

have priority below the primary operators but higher than any binary or relational operator and group right-to-left. Binary, relational, and logical operators all group left-to-right and have priority decreasing as indicated:

binop:
.
/ *
^
+ -
relop:
> >= < <=
== !=
logop:
&&
||

The conditional operator groups right-to-left

? :

Assignment operators all have the same priority and all group right-to-left.

asgnop:
= += -= *= /=

Logical expressions have the value 1 for true, 0 for false. As in C, a non-zero logical expression is deemed to be true. In general, logical expressions are only defined for scalar types. The exception is == and != which are defined for every type.

B.4 Preprocessor

Shading Language compilers will respect ANSI C preprocessor directives, including:

#define *identifier token-string*
#define *identifier* (*identifier , ... , identifier*) *token_string*
#undef *identifier*
#include "*filename*"
#include <*filename*>
#if *constant-expression*
#ifdef *identifier*
#ifndef *identifier*
#else
#endif
#line *constant identifier*
#pragma *token-string*

The behaviors of these preprocessor directives should be identical to that expected from an ANSI C compiler, and it is reasonable to assume that a Shading Language compiler may use whatever compliant ANSI C preprocessor is available on a system. It should, therefore, not be assumed that the preprocessor truly understands the semantics of the Shading Language. For example, simple constant floating-point or boolean expressions that can be evaluated at compile time may be used for *constant-expression*, but not expression involving vectors or other types specific to the Shading Language.

Appendix C

LANGUAGE BINDING DETAILS

C.1 ANSI C Binding

The following is the version of ri.h required for the ANSI-standard C binding of the RenderMan Interface.

```
/*
 *        RenderMan Interface Standard Include File
 *            (for ANSI Standard C)
 */


        /* Definitions of Abstract Types used in RI */
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[4];
typedef RtFloat RtMatrix[4][4];
typedef RtFloat RtBasis[4][4];
typedef RtFloat RtBound[6];
typedef char   *RtString;

typedef void   *RtPointer;
#define RtVoid  void

typedef RtFloat (*RtFilterFunc)(RtFloat, RtFloat, RtFloat, RtFloat);
typedef RtVoid  (*RtErrorHandler)(RtInt, Rtnt, char *);

typedef RtVoid (*RtProcSubdivFunc)(RtPointer, RtFloat);
typedef RtVoid (*RtProcFreeFunc)(RtPointer);
typedef RtVoid (*RtArchiveCallback)(RtToken, char *, ...);

typedef RtPointer RtObjectHandle;
typedef RtPointer RtLightHandle;
```

```
typedef RtPointer RtContextHandle;


        /* Extern Declarations for Predefined RI Data Structures */
#define RI_FALSE     0
#define RI_TRUE      (! RI_FALSE)
#define RI_INFINITY  1.0e38
#define RI_EPSILON   1.0e-10
#define RI_NULL      ((RtToken)0)

extern RtToken  RI_FRAMEBUFFER, RI_FILE;
extern RtToken  RI_RGB, RI_RGBA, RI_RGBZ, RI_RGBAZ, RI_A, RI_Z, RI_AZ;
extern RtToken  RI_PERSPECTIVE, RI_ORTHOGRAPHIC;
extern RtToken  RI_HIDDEN, RI_PAINT;
extern RtToken  RI_CONSTANT, RI_SMOOTH;
extern RtToken  RI_FLATNESS, RI_FOV;
extern RtToken  RI_AMBIENTLIGHT, RI_POINTLIGHT, RI_DISTANTLIGHT,
                RI_SPOTLIGHT;
extern RtToken  RI_INTENSITY, RI_LIGHTCOLOR, RI_FROM, RI_TO, RI_CONEANGLE,
                RI_CONEDELTAANGLE, RI_BEAMDISTRIBUTION;
extern RtToken  RI_MATTE, RI_METAL, RI_SHINYMETAL,
                RI_PLASTIC, RI_PAINTEDPLASTIC;
extern RtToken  RI_KA, RI_KD, RI_KS, RI_ROUGHNESS, RI_KR,
                RI_TEXTURENAME, RI_SPECULARCOLOR;
extern RtToken  RI_DEPTHCUE, RI_FOG, RI_BUMPY;
extern RtToken  RI_MINDISTANCE, RI_MAXDISTANCE, RI_BACKGROUND,
                RI_DISTANCE, RI_AMPLITUDE;
extern RtToken  RI_RASTER, RI_SCREEN, RI_CAMERA, RI_WORLD, RI_OBJECT;
extern RtToken  RI_INSIDE, RI_OUTSIDE, RI_LH, RI_RH;
extern RtToken  RI_P, RI_PZ, RI_PW, RI_N, RI_NP,
                RI_CS, RI_OS, RI_S, RI_T, RI_ST;
extern RtToken  RI_BILINEAR, RI_BICUBIC;
extern RtToken  RI_LINEAR, RI_CUBIC;
extern RtToken  RI_PRIMITIVE, RI_INTERSECTION, RI_UNION, RI_DIFFERENCE;
extern RtToken  RI_PERIODIC, RI_NONPERIODIC, RI_CLAMP, RI_BLACK;
extern RtToken  RI_IGNORE, RI_PRINT, RI_ABORT, RI_HANDLER;
extern RtToken  RI_COMMENT, RI_STRUCTURE, RI_VERBATIM;
extern RtToken  RI_IDENTIFIER, RI_NAME, RI_SHADINGGROUP;
extern RtToken  RI_WIDTH, RI_CONSTANTWIDTH;

extern RtBasis  RiBezierBasis, RiBSplineBasis, RiCatmullRomBasis,
                RiHermiteBasis, RiPowerBasis;

#define RI_BEZIERSTEP     ((RtInt)3)
#define RI_BSPLINESTEP    ((RtInt)1)
#define RI_CATMULLROMSTEP ((RtInt)1)
#define RI_HERMITESTEP    ((RtInt)2)
#define RI_POWERSTEP      ((RtInt)4)

extern RtInt    RiLastError;


        /* Declarations of All the RenderMan Interface Subroutines */
extern RtFloat  RiGaussianFilter(RtFloat x, RtFloat y,
                        RtFloat xwidth, RtFloat ywidth);
extern RtFloat  RiBoxFilter(RtFloat x, RtFloat y,
                        RtFloat xwidth, RtFloat ywidth);
extern RtFloat  RiTriangleFilter(RtFloat x, RtFloat y,
```

```
                        RtFloat xwidth, RtFloat ywidth);
extern RtFloat  RiCatmullRomFilter(RtFloat x, RtFloat y,
                        RtFloat xwidth, RtFloat ywidth);
extern RtFloat  RiSincFilter(RtFloat x, RtFloat y,
                        RtFloat xwidth, RtFloat ywidth);

extern RtVoid RiErrorIgnore(RtInt code, RtInt severity, char *msg);
extern RtVoid RiErrorPrint(RtInt code, RtInt severity, char *msg);
extern RtVoid RiErrorAbort(RtInt code, RtInt severity, char *msg);

extern RtVoid RiProcDelayedReadArchive(RtPointer data, RtFloat detail);
extern RtVoid RiProcRunProgram(RtPointer data, RtFloat detail);
extern RtVoid RiProcDynamicLoad(RtPointer data, RtFloat detail);

extern RtContextHandle RiGetContext(void);
extern RtVoid RiContext(RtContextHandle);

extern RtToken
        RiDeclare(char *name, char *declaration);

extern RtVoid
        RiBegin(RtToken name),
        RiEnd(void),
        RiFrameBegin(RtInt frame),
        RiFrameEnd(void),
        RiWorldBegin(void),
        RiWorldEnd(void);

extern RtVoid
        RiFormat(RtInt xres, RtInt yres, RtFloat aspect),
        RiFrameAspectRatio(RtFloat aspect),
        RiScreenWindow(RtFloat left, RtFloat right, RtFloat bot, RtFloat top),
        RiCropWindow(RtFloat xmin, RtFloat xmax, RtFloat ymin, RtFloat ymax),
        RiProjection(RtToken name, ...),
        RiProjectionV(RtToken name, RtInt n,RtToken tokens[],RtPointer parms[]),
        RiClipping(RtFloat hither, RtFloat yon),
        RiClippingPlane(RtFloat x, RtFloat y, RtFloat z,
                        RtFloat nx, RtFloat ny, RtFloat nz),
        RiShutter(RtFloat min, RtFloat max);

extern RtVoid
        RiPixelVariance(RtFloat variation),
        RiPixelSamples(RtFloat xsamples, RtFloat ysamples),
        RiPixelFilter(RtFilterFunc filterfunc, RtFloat xwidth, RtFloat ywidth),
        RiExposure(RtFloat gain, RtFloat gamma),
        RiImager(RtToken name, ...),
        RiImagerV(RtToken name, RtInt n, RtToken tokens[], RtPointer parms[]),
        RiQuantize(RtToken type, RtInt one, RtInt min, RtInt max, RtFloat ampl),
        RiDisplay(char *name, RtToken type, RtToken mode, ...),
        RiDisplayV(char *name, RtToken type, RtToken mode,
                RtInt n, RtToken tokens[], RtPointer parms[]);

extern RtVoid
        RiHider(RtToken type, ...),
        RiHiderV(RtToken type, RtInt n, RtToken tokens[], RtPointer parms[]),
        RiColorSamples(RtInt n, RtFloat nRGB[], RtFloat RGBn[]),
        RiRelativeDetail(RtFloat relativedetail),
        RiOption(RtToken name, ...),
```

```
        RiOptionV(RtToken name, RtInt n, RtToken tokens[], RtPointer parms[]);

extern RtVoid
        RiAttributeBegin(void),
        RiAttributeEnd(void),
        RiColor(RtColor color),
        RiOpacity(RtColor color),
        RiTextureCoordinates(RtFloat s1, RtFloat t1, RtFloat s2, RtFloat t2,
                RtFloat s3, RtFloat t3, RtFloat s4, RtFloat t4);

extern RtLightHandle
        RiLightSource(RtToken name, ...),
        RiLightSourceV(RtToken name,RtInt n,RtToken tokens[],RtPointer parms[]),
        RiAreaLightSource(RtToken name, ...),
        RiAreaLightSourceV(RtToken name,
                RtInt n, RtToken tokens[], RtPointer parms[]);
extern RtVoid
        RiIlluminate(RtLightHandle light, RtBoolean onoff),
        RiSurface(RtToken name, ...),
        RiSurfaceV(RtToken name, RtInt n, RtToken tokens[], RtPointer parms[]),
        RiAtmosphere(RtToken name, ...),
        RiAtmosphereV(RtToken name,RtInt n,RtToken tokens[],RtPointer parms[]),
        RiInterior(RtToken name, ...),
        RiInteriorV(RtToken name,RtInt n,RtToken tokens[],RtPointer parms[]),
        RiExterior(RtToken name, ...),
        RiExteriorV(RtToken name,RtInt n,RtToken tokens[],RtPointer parms[]),
        RiShadingRate(RtFloat size),
        RiShadingInterpolation(RtToken type),
        RiMatte(RtBoolean onoff);

extern RtVoid
        RiBound(RtBound bound),
        RiDetail(RtBound bound),
        RiDetailRange(RtFloat minvis, RtFloat lowtran, RtFloat uptran, RtFloat
                maxvis),
        RiGeometricApproximation(RtToken type, RtFloat value),
        RiOrientation(RtToken orientation),
        RiReverseOrientation(void),
        RiSides(RtInt sides);

extern RtVoid
        RiIdentity(void),
        RiTransform(RtMatrix transform),
        RiConcatTransform(RtMatrix transform),
        RiPerspective(RtFloat fov),
        RiTranslate(RtFloat dx, RtFloat dy, RtFloat dz),
        RiRotate(RtFloat angle, RtFloat dx, RtFloat dy, RtFloat dz),
        RiScale(RtFloat sx, RtFloat sy, RtFloat sz),
        RiSkew(RtFloat angle, RtFloat dx1, RtFloat dy1, RtFloat dz1,
                RtFloat dx2, RtFloat dy2, RtFloat dz2),
        RiDeformation(RtToken name, ...),
        RiDeformationV(RtToken name,RtInt n,RtToken tokens[],RtPointer parms[]),
        RiDisplacement(RtToken name, ...),
        RiDisplacementV(RtToken name,RtInt n,RtToken tokens[],RtPointer
                parms[]),
        RiCoordinateSystem(RtToken space),
        RiCoordSysTransform(RtToken space);
```

```
extern RtPoint *
        RiTransformPoints(RtToken fromspace, RtToken tospace, RtInt n,
                RtPoint points[]);
extern RtVoid
        RiTransformBegin(void),
        RiTransformEnd(void);

extern RtVoid
        RiAttribute(RtToken name, ...),
        RiAttributeV(RtToken name, RtInt n, RtToken tokens[], RtPointer parms[]);

extern RtVoid
        RiPolygon(RtInt nverts, ...),
        RiPolygonV(RtInt nverts, RtInt n, RtToken tokens[], RtPointer parms[]),
        RiGeneralPolygon(RtInt nloops, RtInt nverts[], ...),
        RiGeneralPolygonV(RtInt nloops, RtInt nverts[],
                RtInt n, RtToken tokens[], RtPointer parms[]),
        RiPointsPolygons(RtInt npolys, RtInt nverts[], RtInt verts[], ...),
        RiPointsPolygonsV(RtInt npolys, RtInt nverts[], RtInt verts[],
                RtInt n, RtToken tokens[], RtPointer parms[]),
        RiPointsGeneralPolygons(RtInt npolys, RtInt nloops[], RtInt nverts[],
                RtInt verts[], ...),
        RiPointsGeneralPolygonsV(RtInt npolys, RtInt nloops[], RtInt nverts[],
                RtInt verts[], RtInt n, RtToken tokens[], RtPointer parms[]),
        RiBasis(RtBasis ubasis, RtInt ustep, RtBasis vbasis, RtInt vstep),
        RiPatch(RtToken type, ...),
        RiPatchV(RtToken type, RtInt n, RtToken tokens[], RtPointer parms[]),
        RiPatchMesh(RtToken type, RtInt nu, RtToken uwrap,
                RtInt nv, RtToken vwrap, ...),
        RiPatchMeshV(RtToken type, RtInt nu, RtToken uwrap,
                RtInt nv, RtToken vwrap,
                RtInt n, RtToken tokens[], RtPointer parms[]),
        RiNuPatch(RtInt nu, RtInt uorder, RtFloat uknot[], RtFloat umin,
                RtFloat umax, RtInt nv, RtInt vorder, RtFloat vknot[],
                RtFloat vmin, RtFloat vmax, ...),
        RiNuPatchV(RtInt nu, RtInt uorder, RtFloat uknot[], RtFloat umin,
                RtFloat umax, RtInt nv, RtInt vorder, RtFloat vknot[],
                RtFloat vmin, RtFloat vmax,
                RtInt n, RtToken tokens[], RtPointer parms[]),
        RiTrimCurve(RtInt nloops, RtInt ncurves[], RtInt order[],
                RtFloat knot[], RtFloat min[], RtFloat max[], RtInt n[],
                RtFloat u[], RtFloat v[], RtFloat w[]);

extern RtVoid
        RiSphere(RtFloat radius, RtFloat zmin, RtFloat zmax, RtFloat tmax, ...),
        RiSphereV(RtFloat radius, RtFloat zmin, RtFloat zmax, RtFloat tmax,
                RtInt n, RtToken tokens[], RtPointer parms[]),
        RiCone(RtFloat height, RtFloat radius, RtFloat tmax, ...),
        RiConeV(RtFloat height, RtFloat radius, RtFloat tmax,
                RtInt n, RtToken tokens[], RtPointer parms[]),
        RiCylinder(RtFloat radius,RtFloat zmin,RtFloat zmax,RtFloat tmax, ...),
        RiCylinderV(RtFloat radius, RtFloat zmin, RtFloat zmax, RtFloat tmax,
                RtInt n, RtToken tokens[], RtPointer parms[]),
        RiHyperboloid(RtPoint point1, RtPoint point2, RtFloat tmax, ...),
        RiHyperboloidV(RtPoint point1, RtPoint point2, RtFloat tmax,
                RtInt n, RtToken tokens[], RtPointer parms[]),
        RiParaboloid(RtFloat rmax,RtFloat zmin,RtFloat zmax,RtFloat tmax, ...),
        RiParaboloidV(RtFloat rmax, RtFloat zmin, RtFloat zmax, RtFloat tmax,
```

```
                RtInt n, RtToken tokens[], RtPointer parms[]),
        RiDisk(RtFloat height, RtFloat radius, RtFloat tmax, ...),
        RiDiskV(RtFloat height, RtFloat radius, RtFloat tmax,
                RtInt n, RtToken tokens[], RtPointer parms[]),
        RiTorus(RtFloat majrad, RtFloat minrad, RtFloat phimin,
                RtFloat phimax, RtFloat tmax, ...),
        RiTorusV(RtFloat majrad,RtFloat minrad,
                RtFloat phimin, RtFloat phimax, RtFloat tmax,
                RtInt n, RtToken tokens[], RtPointer parms[]);


extern RtVoid RiBlobby(RtInt nleaf, RtInt ncode, RtInt code[],
                       RtInt nflt, RtFloat flt[],
                       RtInt nstr, RtToken str[], ...);
extern RtVoid RiBlobbyV(RtInt nleaf, RtInt ncode, RtInt  code[],
                        RtInt nflt, RtFloat flt[],
                        RtInt nstr, RtToken str[],
                        RtInt n , RtToken tokens[], RtPointer parms[]);
extern RtVoid
        RiCurves(RtToken type, RtInt ncurves,
                 RtInt nvertices[], RtToken wrap, ...),
        RiCurvesV(RtToken type, RtInt ncurves, RtInt nvertices[], RtToken wrap,
                  RtInt n, RtToken tokens[], RtPointer parms[]),
        RiPoints(RtInt nverts,...),
        RiPointsV(RtInt nverts, RtInt n, RtToken tokens[], RtPointer parms[]),
        RiSubdivisionMesh(RtToken mask, RtInt nf, RtInt nverts[],
                          RtInt verts[],
                          RtInt ntags, RtToken tags[], RtInt numargs[],
                          RtInt intargs[], RtFloat floatargs[], ...),
        RiSubdivisionMeshV(RtToken mask, RtInt nf, RtInt nverts[],
                           RtInt verts[], RtInt ntags, RtToken tags[],
                           RtInt nargs[], RtInt intargs[],
                           RtFloat floatargs[], RtInt n,
                           RtToken tokens[], RtPointer *parms);

extern RtVoid
        RiProcedural(RtPointer data, RtBound bound,
                RtVoid (*subdivfunc)(RtPointer, RtFloat),
                RtVoid (*freefunc)(RtPointer)
        RiGeometry(RtToken type, ...),
        RiGeometryV(RtToken type, RtInt n, RtToken tokens[], RtPointer parms[]);

extern RtVoid
        RiSolidBegin(RtToken operation),
        RiSolidEnd(void) ;

extern RtObjectHandle
        RiObjectBegin(void);
extern RtVoid
        RiObjectEnd(void),
        RiObjectInstance(RtObjectHandle handle),
        RiMotionBegin(RtInt n, ...),
        RiMotionBeginV(RtInt n, RtFloat times[]),
        RiMotionEnd(void) ;

extern RtVoid
        RiMakeTexture(char *pic, char *tex, RtToken swrap, RtToken twrap,
                RtFilterFunc filterfunc, RtFloat swidth, RtFloat twidth, ...),
```

```
        RiMakeTextureV(char *pic, char *tex, RtToken swrap, RtToken twrap,
                 RtFilterFunc filterfunc, RtFloat swidth, RtFloat twidth,
                 RtInt n, RtToken tokens[], RtPointer parms[]),
        RiMakeBump(char *pic, char *tex, RtToken swrap, RtToken twrap,
                RtFilterFunc filterfunc, RtFloat swidth, RtFloat twidth, ...),
        RiMakeBumpV(char *pic, char *tex, RtToken swrap, RtToken twrap,
                RtFilterFunc filterfunc, RtFloat swidth, RtFloat twidth,
                RtInt n, RtToken tokens[], RtPointer parms[]),
        RiMakeLatLongEnvironment(char *pic, char *tex,
                RtFilterFunc filterfunc,
                RtFloat swidth, RtFloat twidth, ...),
        RiMakeLatLongEnvironmentV(char *pic, char *tex,
                RtFilterFunc filterfunc,
                RtFloat swidth, RtFloat twidth,
                RtInt n, RtToken tokens[], RtPointer parms[]),
        RiMakeCubeFaceEnvironment(char *px, char *nx, char *py, char *ny,
                char *pz, char *nz, char *tex, RtFloat fov,
                RtFilterFunc filterfunc, RtFloat swidth, RtFloat ywidth, ...),
        RiMakeCubeFaceEnvironmentV(char *px, char *nx, char *py, char *ny,
                char *pz, char *nz, char *tex, RtFloat fov,
                RtFilterFunc filterfunc, RtFloat swidth, RtFloat ywidth,
                RtInt n, RtToken tokens[], RtPointer parms[]),
        RiMakeShadow(char *pic, char *tex, ...),
        RiMakeShadowV(char *pic, char *tex,
                RtInt n, RtToken tokens[], RtPointer parms[]);

extern RtVoid
        RiArchiveRecord(RtToken type, char *format, ...),
        RiReadArchive(RtToken name, RtArchiveCallback callback, ...),
        RiReadArchiveV(RtToken name, RtArchiveCallback callback, char*, ...),
                RtInt n, RtToken tokens[], RtPointer parms[]);

extern RtVoid
        RiErrorHandler(RtErrorHandler handler);


/*
       Error Codes
         1 - 10       System and File Errors
        11 - 20       Program Limitations
        21 - 40       State Errors
        41 - 60       Parameter and Protocol Errors
        61 - 80       Execution Errors
*/
#define RIE_NOERROR     ((RtInt)0)
#define RIE_NOMEM       ((RtInt)1)     /* Out of memory */
#define RIE_SYSTEM      ((RtInt)2)     /* Miscellaneous system error */
#define RIE_NOFILE      ((RtInt)3)     /* File nonexistent */
#define RIE_BADFILE     ((RtInt)4)     /* Bad file format */
#define RIE_VERSION     ((RtInt)5)     /* File version mismatch */
#define RIE_DISKFULL    ((RtInt)6)     /* Target disk is full */
#define RIE_INCAPABLE   ((RtInt)11)    /* Optional RI feature */
#define RIE_UNIMPLEMENT ((RtInt)12)    /* Unimplemented feature */
#define RIE_LIMIT       ((RtInt)13)    /* Arbitrary program limit */
#define RIE_BUG         ((RtInt)14)    /* Probably a bug in renderer */
#define RIE_NOTSTARTED  ((RtInt)23)    /* RiBegin not called */
#define RIE_NESTING     ((RtInt)24)    /* Bad begin-end nesting */
#define RIE_NOTOPTIONS  ((RtInt)25)    /* Invalid state for options */
```

```
#define RIE_NOTATTRIBS  ((RtInt)26)    /* Invalid state for attribs */
#define RIE_NOTPRIMS    ((RtInt)27)    /* Invalid state for primitives */
#define RIE_ILLSTATE    ((RtInt)28)    /* Other invalid state */
#define RIE_BADMOTION   ((RtInt)29)    /* Badly formed motion block */
#define RIE_BADSOLID    ((RtInt)30)    /* Badly formed solid block */
#define RIE_BADTOKEN    ((RtInt)41)    /* Invalid token for request */
#define RIE_RANGE       ((RtInt)42)    /* Parameter out of range */
#define RIE_CONSISTENCY ((RtInt)43)    /* Parameters inconsistent */
#define RIE_BADHANDLE   ((RtInt)44)    /* Bad object/light handle */
#define RIE_NOSHADER    ((RtInt)45)    /* Can't load requested shader */
#define RIE_MISSINGDATA ((RtInt)46)    /* Required parameters not provided */
#define RIE_SYNTAX      ((RtInt)47)    /* Declare type syntax error */
#define RIE_MATH        ((RtInt)61)    /* Zerodivide, noninvert matrix, etc. */

        /* Error severity levels */
#define RIE_INFO        ((RtInt)0)    /* Rendering stats and other info */
#define RIE_WARNING     ((RtInt)1)    /* Something seems wrong, maybe okay */
#define RIE_ERROR       ((RtInt)2)    /* Problem. Results may be wrong */
#define RIE_SEVERE      ((RtInt)3)    /* So bad you should probably abort */
```

C.2 RIB Binding

The RenderMan Interface Bytestream Protocol, abbreviated RIB, is a byte-oriented protocol for specifying requests to the RenderMan Interface (RI) library. RIB permits clients of the RenderMan Interface to communicate requests to a remote rendering service, or to save requests in a file for later submission to a renderer. To satisfy the many different needs of clients, the protocol is designed to provide both

- an understandable (potentially) interactive interface to a rendering server, and
- a compact encoded format that minimizes transmission time (and space when stored in a file)

RIB also strives to minimize the amount of communication from a server to a client. This is particularly important in the situation where no communication is possible; e.g., when recording RIB in a file.

RIB is a byte stream protocol. That is, RIB interpreters work by scanning the input stream one byte at a time. This implies interpreters should make no assumptions about data alignment. The protocol is best thought of as a command language where tokens in the input stream can be transmitted either as 7-bit ASCII strings or, optionally, as compressed binary data. The ASCII interface provides a convenient interface for users to interactively communicate with a rendering server and for developers to debug systems that generate RIB. The binary encoding significantly compresses the data stream associated with an RI description with an associated savings in communication overhead and/or file storage cost.

C.2.1 Syntax rules

RenderMan Interface Protocol requests are constructed from sequences of *tokens*. Tokens are formed by the input scanner by grouping characters according to the RIB syntax rules (described below). Other than requirements associated with delimiting tokens, RIB employs a free format syntax.

Character set

The standard character set is the printable subset of the ASCII character set, plus the characters space, tab, and newline (return or line-feed). Non-printing characters are accepted, but are discouraged as they impair portability.

The characters space, tab, and newline are referred to as *white space* characters and are treated equivalently (except when they appear in comments or strings). White space is used to delimit syntactic constructs such as identifiers or numbers. Any number of consecutive white space characters are treated as a single white space character.

The characters '"', '#', '[', and ']' are *special*: they delimit syntactic entities. All other characters are termed *regular characters* and may be used in constructing syntactic entities such as identifiers and numbers.

Comments

Any occurrence of the '#' character, except when in a string, indicates a *comment*. The comment consists of all characters between the '#' and the next newline character. Comments are treated as white space when they are encountered by the input scanner.

Numbers

Numbers include signed integers and reals. An integer consists of an optional sign ('+', '-') followed by one or more decimal digits. The number is interpreted as a signed decimal integer.

A real consists of an optional sign and one or more decimal digits, with an embedded period (decimal point), a trailing exponent, or both. The exponent, if present, consists of 'E' or 'e' followed by an optional sign and one or more decimal digits. The number is interpreted as a real number and converted to an internal floating point value.

Strings

A string is an arbitrary sequence of characters delimited by double quote marks ('"'). Within a string the only special characters are '"' and the '\' (back-slash) character. The '\' character is used as an 'escape' to include the '"' character, non-printing characters, and the '\' character itself. The character immediately following the '\' determines the precise interpretation, as follows:

\n	linefeed (newline)
\r	carriage return
\t	horizontal tab
\b	backspace
\f	form feed
\\	backslash
\"	double quote
ddd	character code *ddd* (octal)
newline	no character – both are ignored

If the character following the '\' is not one of the above, the '\' is ignored.

The *ddd* form may be used to include any 8-bit character constant in a string. One, two, or three octal digits may be specified (with high-order overflow ignored).

The *newline* form is used to break a string into a number of lines but not have the newlines be part of the string.

Names

Any token that consists entirely of regular characters and that cannot be interpreted as a number is treated as a *name*. All characters except specials and white space can appear in names.

Arrays

The characters '[' and ']' are self-delimiting tokens that specify the construction of an array of numbers or strings. An *array* cannot contain both numbers and strings. If an array contains at least one floating point value, all integer values in the array are converted to floating point. Arrays of numbers are used, for example, to specify matrices and points. Arrays of strings are used in specifying options.

Binary encoding

For efficiency, compressed binary encodings of many types of data are also supported. These encodings may be freely intermixed with the normal ASCII strings. The two encodings are differentiated by the top bit of the eight-bit bytes in the input stream. If the top bit is zero, then the byte is interpreted as a 7-bit ASCII character. Otherwise, if the top bit is one, the byte is interpreted as a compressed token according to the rules given below. This differentiation is not applied within string constants or the parameter bytes which follow the initial byte of a compressed token. Table C.1 shows the encoding for compressed tokens with all byte *values* displayed in octal.

Values	Span	Interpreted as...
0–0177	128	ASCII characters
0200–0217	16	encoded integers and fixed-point numbers
0220–0237	16	encoded strings of no more than 15 characters
0240–0243	4	encoded strings longer than 15 characters
0244	1	encoded single precision IEEE floating point value
0245	1	encoded double precision IEEE floating point value
0246	1	encoded RI request
0247–0307	32	*nothing* (reserved)
0310–0313	4	encoded single precision array (length follows)
0314	1	define encoded request
0315–0316	2	define encoded string token
0317–0320	2	interpolate defined string
0321–0377	46	*nothing* (reserved)

Table C.1: Binary Encoding

Four separate data types are supported: signed integers, signed fixed-point numbers, strings, and floating-point numbers. Integers and fixed-point numbers are encoded using a single format while strings are encoded with two different formats according to the length of the string. Both single- and double-precision IEEE format floating-point numbers are supported. Strings that are used repeatedly can be *defined* and then subsequently *referenced* with a compact form that is usually more space efficient.

Arrays of floating-point values are directly supported for efficiency (they can also be specified using the array definition symbols). Single-precision matrices (arrays of 16 floating-point values) can be specified in a total of 66 bytes, while other arrays may require slightly more.

In the following sections the syntax for each encoding is presented as a sequence of bytes separated by '|' symbols. Numeric values should interpreted as octal values (base 8) if they have a leading '0' digit, otherwise as decimal values. Items shown in angle brackets '< >' represent varying items, such as a numeric value or string that is being encoded.

Integers and fixed-point numbers. Integer and fixed-point values can be transmitted in 2-5 bytes. The encoded token has the form:

$$0200 + (d \cdot 4) + w \quad | \quad \textit{<value>}$$

where the next $w + 1$ bytes form a signed integer taken from the most significant byte to the least significant byte, and the bottom d bytes are after the decimal point.

Strings. Strings shorter than 16 bytes, say w bytes, can be transmitted with a prefixing token:

$$0220 + w \quad | \quad \textit{<string>}$$

Other strings must use a prefixing token followed by a variable length string length, and then followed by the string itself:

$$0240 + \ell \quad | \quad \textit{<length>} \quad | \quad \textit{<string>}$$

where $\ell + 1$ is the number of bytes needed to specify the length of the string, $0 \leq \ell \leq 3$. The string length is an unsigned value and is transmitted from most significant byte to least significant byte. Unlike unencoded strings, there are no escape or special characters in an encoded string.

Defining strings. For strings that are to be transmitted repeatedly, a string *token* can be defined with:

$$0315 + w \quad | \quad \textit{<token>} \quad | \quad \textit{<string>}$$

where $w + 1$ is the number of bytes needed to specify the token, (1 or 2), and the string being defined is transmitted in an encoded or unencoded form. The token is an unsigned value and is transmitted from most significant byte to least significant byte. For efficiency, the range of tokens defined should be as *compact* as possible.

Referencing defined strings. To interpolate a string that has previously been defined (as described above), the following is used:

$$0317 + w \quad | \quad \textit{<token>}$$

where the *token* refers to the string to be interpolated.

Floating-point values. Floating-point values are transmitted in single-precision or double-precision IEEE format from most significant byte to least significant byte. Single-precision floating-point values occupy four bytes in the input stream and double-precision values occupy eight bytes.

Floating point arrays. Aggregates of single-precision floating-point values can be transmitted with a prefixing token byte. Variable sized arrays are transmitted as a token byte followed by a variable length array size, and then followed by the array itself:

$$0310 + \ell \quad | \quad \textit{<length>} \quad | \quad \textit{<array of floats>}$$

The array length is an unsigned value, $\ell + 1$ bytes long, and is transmitted from most

significant byte to least significant byte.

Defining RI requests. Before an encoded request can be used, it must first be bound to its ASCII equivalent with:

0314 | *<code>* | *<string>*

where the *code* is one byte and *string* is the ASCII form of the request.

Referencing defined RI requests. A previously defined RI request is referenced with two bytes; a prefixing token, 0246, followed by a request code.

0246 | *<code>*

This means that no more than 256 RI requests can be directly encoded in the binary protocol.

Example. Consider the following sequence of RIB commands:

```
version 3.03
ErrorHandler "print"
Display "test.25.pic" "file" "rgba"
Format 512 307 1
Clipping 0.1 10000
WorldBegin
Declare "direction" "point"
LightSource "windowlight" 0 "direction" [1 0 -0.1]
Color [1 1 1]
Orientation "lh"
Sides 1
AttributeBegin
MotionBegin [0 1]
Translate 1.91851 0.213234 1.55
Sphere 2 -0.3 1.95 175
MotionEnd
AttributeEnd
```

This could translate into the encoded sequence of bytes shown in Figure C.1.

Version Number

The RIB stream supports a special version request that is used internally to ensure that the syntax of the stream is compatible with the parser being used.

version num — Specifies the protocol version number of the RIB stream. The stream specified in this document is version 3.03. A RIB parser may refuse to parse streams with incompatible version numbers.

C.2.2 Error handling

There are two types of errors that may be encountered while processing a RIB input stream: *syntactic errors* and *semantic errors*. Syntactic errors occur when the stream of tokens fails to form a syntactically legal statement or request. For example, a syntactic error occurs when a required parameter is missing, or when a string is left unterminated. Semantic errors occur when a syntactically legal statement contains incorrect data; e.g., when a parameter that must be non-negative is specified to be -1.

RIB defines a number of syntactic errors and a limited number of semantic errors. In theory RIB should be responsible only for syntactic errors. However, due to the weak typing of programming languages such as C, semantic errors that can not be easily recognized within the RenderMan Interface software are checked at the RIB level. For example, RIB checks arrays that are to be converted to matrices to be sure they have 16 values.

Table C.2 shows the set of errors recognized by RIB. Detailed descriptions of the errors are given below.

All errors encountered by a RIB interpreter require some associated action to be performed. In the case of syntax errors, if input processing is to be continued, the input scanner must resynchronize itself with the input stream. This synchronization is done by reading and discarding tokens from the input stream until a valid RIB request token is encountered. That is, any tokens between the point of the syntax error and the next request token are discarded. The protocol has been designed so that no more than one request (along with any associated parameters) must be discarded when recovering from an error.

Errors are handled in one of three ways:

- They are ignored and the rendering process will proceed to its completion no matter what input stream is provided.
- They cause diagnostic messages to be generated on the renderer's standard error stream, but they are otherwise ignored (the default).
- The first error causes a diagnostic message to be generated and the renderer terminates immediately without creating an image.

If the RIB interpreter is acting as a network server, in direct communication with a client application, the interpreter may send parsing error signals back to the client. These signals take the form of the following RIB requests, though they are not valid in the client-to-server stream. None of these error requests have arguments. Note that some errors may not be recognized immediately by a RIB interpreter upon parsing a request. This may be due to buffering or queuing built into the interface between the intepreter and the renderer. In a client-server environment this may have implications for the client application.

arraytoobig	The interpreter was unable to allocate sufficient memory to store an array specified in the input stream. This error is dependent on the interpreter's implementation. Good implementations of a RIB interpreter support arrays as large as memory will permit.

Name	Description
arraytoobig	insufficient memory to construct array
badargument	incorrect parameter value
badarray	invalid array specification
badbasis	undefined basis matrix name
badcolor	invalid color specification
badhandle	invalid light or object handle
badparamlist	parameter list type mismatch
badripcode	invalid encoded RIB request code
badstringtoken	undefined encoded string token
badtoken	invalid binary token
badversion	protocol version number mismatch
limitcheck	overflowing an internal limit
outofmemory	generic instance of insufficient memory
protocolbotch	malformed binary encoding
stringtoobig	insufficient memory to read string
syntaxerror	general syntactic error
unregistered	undefined RIB request

Table C.2: RIB Errors

badargument — The RIB interpreter encountered an invalid parameter value in parsing a request.

EXAMPLE

```
Polygon "N" [...] # no "P" specified
PointsGeneralPolygons [2 2] [4 3 4]... # bad nloops
```

badarray — The number of items in an array is inappropriate for the specified parameter, or an array has both string and number elements. EXAMPLE

```
Basis [0 1 2 3] Cone [1.5] Bound [0 1 0 "oops" ]
```

badbasis — The basis matrix name specified in a basis request is not known by the RIB interpreter.

EXAMPLE

```
Basis "my-favorite-basis"
```

badcolor — An invalid color was supplied as a parameter to a request. That is, an array was specified with an incorrect number of elements.

EXAMPLE

```
Opacity [.5 1] # with 3-channel colors
```

badhandle — An invalid light or object handle was supplied as a parameter to an **Illuminate**, **ObjectInstance**, **LightSource**, **AreaLightSource**, or **ObjectBegin** request. For **Illuminate**, the light handle must be an integer value specified in a previous **LightSource** or **AreaLightSource** request. For **ObjectInstance**, the object handle must be an integer value specified in a previous **ObjectBegin** request. For **LightSource**, **AreaLightSource**, and **ObjectBegin** this error is raised if the number specified for a light handle is significantly larger than any previous handle; for example, specifying 3000 when the largest previous handle was 10 (this is used as a "sanity check" to guard against corrupted input data).

EXAMPLE

```
LightSource "finite" 1
Illuminate 99999
```

badparamlist — In a token-value pair of a parameter list, the type of a value did not agree with the declared type of the token.

EXAMPLE

```
Declare "gridsize" "uniform float[2]"
Option "limits" "gridsize" "not a number"
```

badripcode — A binary encoded token that specified a RIB request used an undefined request code. Request codes must be defined, prior to their use, with the binary encoding protocol; see the section on Binary Encoding.

badstringtoken — A binary encoded string token referenced a string that had not previously been defined. The binary encoding scheme is described in Binary encoding.

badtoken — A byte with the most significant bit set was not recognized as a valid binary encoding. The binary encoding scheme is described in Binary encoding.

EXAMPLE

```
\300
```

badversion — The RIB protocol version number specified in a version request was greater than the protocol version number of the interpreter. SEE ALSO version

limitcheck — An internal limit was encountered during normal operation of the interpreter. Implementers of RIB interpreters are expected to avoid imposing arbitrary limits. Some implementations may, however, need to limit the maximum size of strings, arrays, etc. due to memory constraints.

outofmemory — The interpreter ran out of memory in the normal course of operation. Interpreters are expected to utilize whatever memory is available in their operating environment. If only a limited amount of memory is present on the machine they are operating on, they may restrict their use. If memory is arbitrarily limited, however, running out of space should result in a limitcheck error, not outofmemory.

protocolbotch — A protocol error was encountered while parsing binary encoded data in the input stream. In particular, when defining a string or request code, an expected string was not encountered. The binary encoding scheme is described in Binary encoding.

stringtoobig — The interpreter ran out of space while parsing a string. This error is a specific instance of the outofmemory error. SEE ALSO outofmemory, limitcheck

syntaxerror — he interpreter recognized a syntax error of indeterminate nature. Syntax errors can occur from unterminated strings or invalid numbers.

EXAMPLE

```
"this is an unterminated string
01a3 # invalid integer
```

unregistered — The interpreter encountered a name token that was not a valid request. This is usually due to misspelling a request name, or not enclosing a string in quote marks (").

EXAMPLE

```
Basis power
```

```
  v    e    r    s    i    o    n  212    # version
003  007  256    E    r    r    o    r    # 3.03 Error
  H    a    n    d    l    e    r  225    # Handler "
  p    r    i    n    t    D    i    s    # print" Dis
  p    l    a    y  315   \0  233    t    # play <defstr 0 "t
  e    s    t    .    2    5    .    p    # est.25.p
  i    c  317   \0  315  001  224    f    # ic"> <str 0> <defstr 1 "f
  i    l    e  317  001  315  002  224    # ile"> <str 1> <defstr 2 "
  r    g    b    a  317  002    F    o    # rgba"> <str 2> Fo
  r    m    a    t  201  002   \0  201    # rmat 512
001    3  200  001    C    l    i    p    # 307 1 Clip
  p    i    n    g  211  031  231  201    # ping 0.1
  ’  020  314  272  232    W    o    r    # 10000 <defreq 0272 "Wor
  l    d    B    e    g    i    n  246    # ldBegin"> <req
272  314  207  227    D    e    c    l    # 0272> <defreq 0207 "Decl
  a    r    e  246  207  315  003  231    # are"> <req 0207> <defstr 3 "
  d    i    r    e    c    t    i    o    # directio
  n  317  003  315  004  225    p    o    # n"> <str 3> <defstr 4 "po
  i    n    t  317  004  314  224  233    # int"> <str 4> <defreq 0224 "
  L    i    g    h    t    S    o    u    # LightSou
  r    c    e  246  224  315  005  233    # rce"> <req 0224> <defstr 5 "
  w    i    n    d    o    w    l    i    # windowli
  g    h    t  317  005  200  001  317    # ght"> <str 5> 1 <str
003  310  003    ?  200   \0   \0   \0    # 3> [1
 \0   \0   \0  275  314  314  315  314    # 0 -0.1] <defreq
203  225    C    o    l    o    r  246    # 0203 "Color"> <req
203  310  003    ?  200   \0   \0    ?    # 0203> [1
200   \0   \0    ?  200   \0   \0  314    # 1 1] <defreq
237  233    O    r    i    e    n    t    # 0237 "Orient
  a    t    i    o    n  246  237  315    # ation"> <req 0237> <defstr
006  222    l    h  317  006  314  254    # 6 "lh"> <str 6> <defreq 0254
225    S    i    d    e    s  246  254    # "Sides"> <req 0254>
200  001  314  177  236    A    t    t    # 1 <defreq 0177 "Att
  r    i    b    u    t    e    B    e    # ributeBe
  g    i    n  246  177  314  227  233    # gin"> <req 0177> <defreq 0227 "
  M    o    t    i    o    n    B    e    # MotionBe
  g    i    n  246  227  310  002   \0    # gin"> <req 0227> [
 \0   \0   \0    ?  200   \0   \0  314    # 0 1] <defreq
270  231    T    r    a    n    s    l    # 0270 "Transl
  a    t    e  246  270  212  001  353    # ate"> <req 0270> 1.91851
  #  211    6  226  212  001  214  314    # 0.213234 1.55
314  260  226    S    p    h    e    r    # <defreq 0260 "Spher
  e  246  260  200  002  211  263    4    # e"> <req 0260> 2 -0.3
212  001  363    3  201   \0  257  314    # 1.95 1.75 <defreq
230  231    M    o    t    i    o    n    # 0230 "Motion
  E    n    d  246  230  314  200  234    # End"> <req 0230> <defreq 0200 "
  A    t    t    r    i    b    u    t    # Attribut
  e    E    n    d  246  200              # eEnd"> <req 0200>
```

Figure C.1: Example encoded RIB byte stream

Appendix D

RENDERMAN INTERFACE BYTESTREAM CONVENTIONS

Version 1.1

File structuring conventions for RIB files are presented to facilitate the use of RIB as a file format for rendering interchange. A format for single User Entities is presented to allow importing external models into existing RIB streams. Finally, we describe a rendering services file format that will enable Render Managers to provide services to a specific renderer.

D.1 RIB File Structuring Conventions

The RenderMan Interface Bytestream (RIB) is a complete specification of the required interface between modelers and renderers. In a distributed modeling and rendering environment RIB serves well as a rendering file format. As RIB files are passed from one site to another, utilities for shader management, scene editing, and rendering job dispatching (referred to hereafter as Render Managers) can benefit from additional information not strictly required by rendering programs. Additional information relating to User Entities, resource requirements and accounting can be embedded in the RIB file by a modeler through the "proper" use of RIB in conjunction with some simple file structuring conventions.

This section lays out a set of RIB file format conventions which are patterned loosely after the model put forth in Adobe's "Document Structuring Conventions."

D.1.1 Conforming files

The conventions outlined in this section are optional in the sense that they are not interpreted by a renderer and thus will not have any effect on the image produced. Although a Render Manager may require conformance to these conventions, it may choose to utilize or ignore any subset of the structural information present in the RIB file. A RIB file is said to be conforming if it observes the Pixar RIB File Structuring Conventions, and the conforming file can be expected to adhere to specific structuring constraints in that case.

Using RIB File structuring conventions

These conventions are designed to facilitate communication between modeling/animation systems and network rendering management systems. In a distributed environment many decisions relating to the final appearance of rendered frames may need to be deferred until the selection of a particular renderer can be made. A render management system should provide the ability to tailor the scene to the resources and capabilities of the available rendering and output systems. Unfortunately, a modeling/animation system cannot, in general, assume that any particular render management services are available. The following strategies should thus be adopted with the goal of making a RIB file reasonably self-contained and renderer-independent:

- Any nonstandard shaders, optional RenderMan features (motion blur, CSG, level of detail) or textures should be flagged as special resource requirements.
- Renderer-specific options or attributes should be specified according to the special comment conventions described below.
- Display-dependent RenderMan options should not be included except to indicate to Render Managers that such options are mandatory.

D.1.2 RIB File structure conventions

Following is a structured list of components for a conforming RIB file that diagrams the "proper" use of RIB. Some of the components are optional and will depend greatly on the resource requirements of a given scene.

Scope

Indentation indicates the scope of the following command.

Preamble and global variable declarations (RIB requests: version, declare)
Static options and default attributes (image and display options, camera options)
Static camera transformations (camera location and orientation)
Frame block (if more than one frame)
    Frame-specific variable declarations
    Variable options and default attributes
    Variable camera transforms
    World block
        (scene description)
        User Entity (enclosed within **AttributeBegin**/**AttributeEnd**)
        User Entity (enclosed within **AttributeBegin**/**AttributeEnd**)
        User Entity
more frame blocks

This structure results from the vigorous application of the following **Scoping Conventions**:

- No attribute inheritance should be assumed unless implicit in the definition of the User Entity (i.e., within a hierarchy).
- No attribute should be exported except to establish either global or local defaults.

The RenderMan Specification provides block structuring to organize the components of a RIB file. Although the use of blocks is only required for frame and world constructs by the Specification, the liberal use of attribute and transform blocks is encouraged. A modeler enables a Render Manager to freely manipulate, rearrange, or delete scene elements (frames, cameras, lights, User Entities) by carefully bounding these elements in the RIB file according to scope. A Render Manager might, for example, strip all of the frames out of a RIB file and distribute them around a network of rendering servers. This, of course, is only possible if the RIB file has been structured in such a way as to bound those things pertaining to a given frame within its frame block and those things pertaining to all frames outside and before all frame blocks.

User Entities

A User Entity couples a collection of geometric primitives and/or User Entities with shading and geometric attributes. As such it introduces a level of scope that is more local than that implied by the RenderMan world block. Typically, the term User Entity refers to a geometric element within a scene whose attributes or position a user may wish to modify or tweak. Because there is some computational expense associated with attribute block structuring, there is an intrinsic trade-off between control over individual User Entities and rendering time/memory requirements. At one extreme, the entire scene is made up of one User Entity within one attribute block. At the other extreme, each polygon is a User Entity and the renderer is forced to spend most of its time managing the graphics state. Modeling programs and their users may wish to carefully weigh this trade-off.

The Scoping Conventions above prescribe the following User Entity Conventions:

- All User Entities must be delimited by an attribute block.
- All User Entities must have an identifier attribute that uniquely characterizes that Entity to the user. Two special identifier attributes are provided to distinguish between Entities organized by a geometric relationship (the name identifier) and Entities organized according to material makeup (the shadinggroup identifier).
- A User Entity must be completely described within its attribute block.

Nonportable options and attributes

The following list of RIB requests are restricted as they either limit the device independence of the file or they control rendering quality or speed parameters. Render managers should provide this kind of control to users at render time. The inclusion of these restricted requests by a modeler should indicate to a Render Manager that they are, in some sense, mandatory. When including nonportable options or attributes in the RIB file, they should be located contiguously (according to scope) in a RIB file.

Attribute	**Format**	**PixelFilter**	**ShadingRate**
ColorSamples	**FrameAspectRatio**	**PixelSamples**	
Cropwindow	**Imager**	**PixelVariance**	
Exposure	**Option**	**Quantize**	

D.1.3 Conventions for structural hints

The '##' character sequence is used to designate structural hints. Any characters found after these special characters and before the next newline character are construed as special hints intended for Render Managers. Such hints should conform to the conventions outlined herein and should provide structural, resource, or administrative information which cannot easily be incorporated into or derived from the standard RIB stream. The same scoping considerations which apply to RIB should also be applied toward special comments.

Header information

Header information must be located immediately beginning any conforming RIB file. These hints should provide scene-global administrative and resource information. Header entries should precede any RIB requests and must be contiguous. If a header entry appears twice in a file, the first occurrence should be considered to be the true value.

##RenderMan RIB-Structure 1.1 [*keyword*]
: This entry should be the first line in a conforming RIB file. Its inclusion indicates full conformance to these specifications. The addition of the special keyword, *Entity*, specifies that the file conforms to the User Entity conventions described in the Rib Entity Files section.

##Scene *name*
: This entry allows a scene name to be associated with the RIB file.

##Creator *name*
: Indicates the file creator (usually the name of the modeling or animation software).

##CreationDate *time*
: Indicates the time that the file was created. It is expressed as a string of characters and may be in any format.

##For *name*
: Indicates the user name or user identifier (network address) of the individual for whom the frames are intended.

##Frames *number*
: Indicates the number of frames present in the file.

##Shaders *shader1*, *shader2*, *...*
: Indicates the names of nonstandard shaders required. When placed in the header of a RIB file, any nonstandard shaders that appear in the entire file should be listed. When

placed within a frame block, any nonstandard shaders that appear in that frame must be listed.

##**Textures** *texture1*, *texture2*, *...*
Lists any preexisting textures required in the file. When placed in the header of a RIB file, any preexisting textures that appear anywhere in the file should be listed. When placed within a frame block, any preexisting shaders that appear in that frame must be listed.

##**CapabilitiesNeeded** *feature1*, *feature2*, *...*
Indicates any RenderMan Interface optional capabilites required in the file (when located in the header) or required in the frame (when located at the top of a frame block). The optional capabilities are:

Area Light Sources	Motion Blur	Special Camera Projections
Bump Mapping	Programmable Shading	Spectral Colors
Deformations	Radiosity	Texture Mapping
Displacements	Ray Tracing	Trim Curves
Environment Mapping	Shadow Depth Mapping	Volume Shading
Level Of Detail	Solid Modeling	

See Part I, Section 1, Introduction, for a description of these capabilities.

Frame information

Frame-local information must be located directly after a FrameBegin RIB request and be contiguous. These comments should provide frame-local information that contains administrative and resource hints.

##**CameraOrientation** *eyex eyey eyez atx aty atz* [*upx upy upz*]
Indicates the location and orientation of the camera for the current frame in World Space coordinates. The up vector is optional and the default value is [0 1 0].

##**Shaders** *shader1*, *shader2*,...
Lists the nonstandard shaders required in the current frame.

##**Textures** *texture1*, *texture2*,...
Lists the nonstandard textures required in the current frame.

##**CapabilitiesNeeded** *feature1*, *feature2*,...
Lists the special capabilities required in the current frame from among those listed under Header Information.

Body Information

Body information may be located anywhere in the RIB file.

##Include *filename*

This entry allows the specification of a file name for inclusion in the RIB stream. Note that the Include keyword itself does not cause the inclusion of the specified file. As with all structural hints, the Include keyword serves only as a special hint for render management systems. As such, the Include keyword should only be used if render management facilities are known to exist.

D.1.4 RIB File structuring example

```
##RenderMan RIB-Structure 1.1
##Scene Bouncing Ball
##Creator /usr/ucb/vi
##CreationDate 12:30pm 8/24/89
##For RenderMan Jones
##Frames 2
##Shaders PIXARmarble, PIXARwood, MyUserShader
##CapabilitiesNeeded ShadingLanguage Displacements
version 3.03
Declare "d" "uniform point"
Declare "squish" "uniform float"
Option "limits" "bucketsize" [6 6]  #renderer specific
Option "limits" "gridsize" [18]  #renderer specific
Format 1024 768 1  #mandatory resolution
Projection "perspective"
Clipping 10 1000.0
FrameBegin 1
##Shaders MyUserShader, PIXARmarble, PIXARwood
##CameraOrientation 10.0 10.0 10.0 0.0 0.0 0.0
Transform  [.707107  -.408248  -.57735 0
            0  .816497  -.57735  0
            -.707107  -.408248  -.57735  0
            0  0  17.3205  1 ]
WorldBegin
AttributeBegin
Attribute "identifier" "name" "myball"
Displacement "MyUserShader" "squish" 5
AttributeBegin
Attribute "identifier" "shadinggroup" ["tophalf"]
Surface "PIXARmarble"
Sphere .5 0 .5 360
AttributeEnd
AttributeBegin
Attribute "identifier" "shadinggroup" ["bothalf"]
Surface "plastic"
Sphere .5 -.5 0. 360
AttributeEnd
AttributeEnd
AttributeBegin
Attribute "identifier" "name" ["floor"]
Surface "PIXARwood" "roughness" [.3] "d" [1]
# geometry for floor
Polygon "P" [-100. 0. -100.  -100. 0. 100.  100. 0. 100.  10.0 0. -100.]
AttributeEnd
WorldEnd
FrameEnd
```

```
FrameBegin 2
##Shaders PIXARwood, PIXARmarble
##CameraOrientation 10.0 20.0 10.0 0.0 0.0 0.0
Transform [.707107  -.57735  -.408248  0
           0   .57735
           -.815447 0
           -.707107  -.57735  -.408248  0
           0  0  24.4949 1 ]
WorldBegin
AttributeBegin
Attribute "identifier" "name" ["myball"]
AttributeBegin
Attribute "identifier" "shadinggroup" ["tophalf"]
Surface "PIXARmarble"
ShadingRate .1
Sphere .5 0 .5 360
AttributeEnd
AttributeBegin
Attribute "identifier" "shadinggroup" ["bothalf"]
Surface "plastic"
Sphere .5 -.5 0 360
AttributeEnd
AttributeEnd
AttributeBegin
Attribute "identifier" "name" ["floor"]
Surface "PIXARwood" "roughness" [.3] "d" [1]
# geometry for floor
AttributeEnd
WorldEnd
FrameEnd
```

D.2 RIB Entity Files

A RIB Entity File contains a *single* User Entity. RIB Entity Files are incomplete since they do not contain enough information to describe a frame to a renderer. RIB Entity Files depend on Render Management services for integration into "legal," or complete, RIB Files. These files provide the mechanism for 3-D "clip-art" by allowing Render Managers to insert objects into preexisting scenes.

RIB Entity Files must conform to the User Entity Conventions described in the User Entites section. To summarize, a User Entity must be delimited by an attribute block, must have a name attribute, and must be completely contained within a single attribute block. Three additional requirements must also be met:

- The header hint: ##RenderMan RIB-Structure 1.1 Entity must be included as the first line of the file.
- The Entity must be built in an object coordinate system which is centered about the origin.
- The Entity must have a RIB bound request to provide a single bounding box of all geometric primitives in the User Entity.

D.2.1 RIB Entity File example

```
##RenderMan RIB-Structure 1.1 Entity
AttributeBegin  #begin unit cube
Attribute "identifier" "name" "unitcube"
Bound -.5 .5 -.5 .5 -.5 .5
TransformBegin
# far face
Polygon "P" [.5 .5 .5  -.5 .5 .5  -.5 -.5 .5  .5 -.5 .5]
Rotate 90  0 1 0
# right face
Polygon "P" [.5 .5 .5  -.5 .5 .5  -.5 -.5 .5  .5 -.5 .5]
# near face
Rotate 90  0 1 0
Polygon "P" [.5 .5 .5  -.5 .5 .5  -.5 -.5 .5  .5 -.5 .5]
# left face
Rotate 90  0 1 0
Polygon "P" [.5 .5 .5  -.5 .5 .5  -.5 -.5 .5  .5 -.5 .5]
TransformEnd
TransformBegin
# bottom face
Rotate 90  1 0 0
Polygon "P" [.5 .5 .5  -.5 .5 .5  -.5 -.5 .5  .5 -.5 .5]
TransformEnd
TransformBegin
# top face
Rotate -90  1 0 0
Polygon "P" [.5 .5 .5  -.5 .5 .5  -.5 -.5 .5  .5 -.5 .5]
TransformEnd
AttributeEnd  #end unit cube
```

D.3 RenderMan Renderer Resource Files

Renderer Resource Files are intended to provide information to Render Managers and modelers about the specific features, attributes, options, and resources of a particular renderer. In an environment where multiple renderers are available, a Render Manager can provide the user with the ability to tailor RIB file to best suit the desired renderer.

Renderer Resource Files should be shipped with any RenderMan renderer and should be updated on-site by the local system administrator to reflect the resources available to a renderer. Only those sections containing site-specific information can be customized in this way. The simple ASCII format of Renderer Resource Files makes them easy to read, modify and parse.

D.3.1 Format of Renderer Resource Files

A Renderer Resource File is broken up into a series of sections delimited by special keywords. Within each section, all related information is provided using a section-specific predefined format. A special include keyword is provided to simplify the task of customizing Resource Files. The keywords are as follows:

##RenderMan Resource-1.0
Must be included as the first line in any Renderer Resource File.]

##Renderer *name*
Requires the name of the renderer along with any revision or date information.]

##Include *file*
Allows the inclusion of a specified file. This keyword should only be used in sections which are modifiable.

##RenderManCapabilitiesAvailable
This keyword identifies the section enumerating the Capabilities provided by the renderer. The list of capabilities are found in the Header information section and in Section 1. Each capability implemented by the renderer must appear as one line in this section. No entries in this section should be modified.

##RendererSpecificAttributes
This keyword identifies the section which enumerates the Renderer Specific Attributes. These attributes are invoked with the RIB call Attribute. Each attribute implemented by the renderer must appear as one line in this section with legal RIB syntax. The class of all parameter identifiers must be declared previously with a Declare RIB request. If arguments are required for a given attribute, the entry should specify the default value for that attribute. No entries in this section should be modified.

##RendererSpecificOptions
This keyword identifies the section which enumerates Renderer Specific Options. These attributes are invoked with the RIB call **Option**. Each option implemented by the renderer must appear as one line in this section with legal RIB syntax. The class of all parameter identifiers must be declared previously with a **Declare** RIB request. No entries in this section should be modified.

##ShaderResources
This keyword identifies the section which enumerates Shaders available to the renderer. Both built-in and programmed shaders should be listed here. A RenderMan Shading Language declaration for each shader must be provided to enumerate the specific instantiated variables. A declaration may cross line boundaries. This section can be customized to a specific site.

##TextureResources
This keyword identifies the section which enumerates the Textures available to the renderer. The name of each texture may be followed on the same line by an optional string which provides a short description of the Texture. If included, the string should be preceded by the '#' character. This section can be customized to a specific site.

D.3.2 Renderer Resource File example

```
##RenderMan Resource-1.0
##Renderer TrayRacer 1.0
##RenderManCapabilitiesAvailable
Solid Modeling
```

```
Motion Blur
Programmable Shading
Displacements
Bump Mapping
Texture Mapping
Ray Tracing
Environment Mapping
##RendererSpecificAttributes
Declare "refractionindex" "uniform float"
Declare "displacement" "uniform float"
Attribute "volume" "refractionindex" [1.0]
Attribute "bound" "displacement" 3.5
##RendererSpecificOptions
Declare "bucketsize" "uniform integer[2]"
Declare "texturememory" "uniform integer"
Declare "shader" "string"
Declare "texture" "string"
Option "limits" "bucketsize" [12 12]
Option "limits" "texturememory" 1024
Option "searchpath" "shader" "/usr/local/prman/shaders"
Option "searchpath" "texture" "/usr/local/prman/textures"
##ShaderResources
surface wood(
float ringscale = 10;
color lightwood = color(.3, .12, 0.0);
darkwood = color(.05, .01, .005);
float Ka =.2,
Kd =.4,
Ks  =.6,
roughness =.1)
displacement dented(float Km = 1.0)
light slideprojector (
float fieldofview=PI/32;
point from = {8,-4,10), to = {0,0,0), up = point "eye" (0,1,0);
string slidename = "" )
##Include othershaderfile
##TextureResources
brick
bluebrick
grass #kentucky bluegrass-1 square meter
moss  #spanish moss
logo
##Include othertexturefile
```

Appendix E

STANDARD BUILT-IN FILTERS

In this section the required RenderMan Interface filters are defined. Keep in mind that the filter implementations may assume that they will never be passed (x, y) values that are outside the $([-xwidth/2, xwidth/2], [-ywidth/2, ywidth/2])$ range.

E.1 Box Filter

```
RtFloat
RiBoxFilter (RtFloat x, RtFloat y, RtFloat xwidth, RtFloat ywidth)
{
    return 1.0;
}
```

E.2 Triangle Filter

```
RtFloat
RiTriangleFilter (RtFloat x, RtFloat y, RtFloat xwidth, RtFloat ywidth)
{
    return   ( (1.0 – fabs(x)) / (xwidth*0.5) )
           * ( (1.0 – fabs(y)) / (ywidth*0.5) ) ;
}
```

E.3 CatmullRom Filter

```
RtFloat
RiCatmullRomFilter (RtFloat x, RtFloat y, RtFloat xwidth, RtFloat ywidth)
{
    RtFloat r2 = (x*x + y*y);
    RtFloat r = sqrt(r2);
    return (r >= 2.0) ? 0.0 :
           (r < 1.0) ? (3.0*r*r2 – 5.0*r2 + 2.0) : (–r*r2 + 5.0*r2 – 8.0*r + 4.0);
```

```
}
```

E.4 Gaussian Filter

```
RtFloat
RiGaussianFilter (RtFloat x, RtFloat y, RtFloat xwidth, RtFloat ywidth)
{
    x *= 2.0 / xwidth;
    y *= 2.0 / ywidth;
    return exp(–2.0 * (x*x + y*y));
}
```

E.5 Sinc Filter

```
RtFloat
RiSincFilter (RtFloat x, RtFloat y, RtFloat xwidth, RtFloat ywidth)
{
    RtFloat s, t;
    if (x > –0.001 && x < 0.001)
        s = 1.0;
    else
        s = sin(x)/x;
    if (y > –0.001 && y < 0.001)
        t = 1.0;
    else
        t = sin(y)/y;
    return s*t;
}
```

Appendix F

STANDARD BASIS MATRICES

In this section the required RenderMan Interface basis matrices (used for bicubic patches).

$$\text{RiBezierBasis} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\text{RiBSplineBasis} = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}$$

$$\text{RiCatmullRomBasis} = \frac{1}{2}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$$

$$\text{RiHermiteBasis} = \begin{bmatrix} 2 & 1 & -2 & 1 \\ -3 & -2 & 3 & -1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\text{RiPowerBasis} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Appendix G

RENDERMAN INTERFACE QUICK REFERENCE

G.1 Interface Routines

Graphics State	
Function	**Description**
RiAreaLightSource (name, parameterlist)	creates an area light and makes it the current area light source. Each subsequent geometric primitive is added to the list of surfaces that define the area light.
RiAtmosphere (name, parameterlist)	sets the current atmosphere shader.
RiAttribute (name, parameterlist)	sets the parameters of the attribute name, using the values specified in the token-value list parameterlist.
RiAttributeBegin () **RiAttributeEnd** ()	pushes and pops the current set of attributes.
RiBegin () **RiEnd** ()	initializes and terminates a rendering session.
RiBound (bound)	sets the current bound to bound.
RiClipping (near, far)	sets the position of the near and far clipping planes along the direction of view.
RiClippingPlane (p0, p1, p2, n0, n1, n2)	Clip all geometry on the positive side of the plane described by a point and normal.
RiColor (color)	sets the current color to color.
RiColorSamples (n, nRGB, RGBn)	controls the number of color components or samples to be used in specifying colors.
RiConcatTransform (transform)	concatenates the transformation transform onto the current transformation.
RiCoordinateSystem (space)	marks the coordinate system defined by the current transformation with the name space and saves it.

Graphics State (continued)	
Function	**Description**
RiCoordSysTransform (space)	sets the current transformation to be the one previously named with **RiCoordinateSystem**.
RiCropWindow (xmin, xmax, ymin, ymax)	renders only a subrectangle of the image.
RiDepthOfField (fstop, focallength, focaldistance)	focaldistance sets the distance along the direction of view at which objects will be in focus.
RiDetail (bound)	sets the current detail to the area of the bounding box bound in the raster coordinate system.
RiDetailRange (minvisible, lowertransition, uppertransition, maxvisible)	sets the current detail range.
RiDisplacement (name, parameterlist)	sets the current displacement shader to the named shader.
RiDisplay (name, type, mode, parameterlist)	chooses a display by name and sets the type of output being generated.
RiExposure (gain, gamma)	controls the sensitivity and non-linearity of the exposure process.
RiExterior (name, parameterlist)	sets the current exterior volume shader.
RiFormat (xresolution, yresolution, pixelaspectratio)	sets the horizontal (xresolution) and vertical (yresolution) resolution (in pixels) of the image to be rendered.
RiFrameAspectRatio (frameaspectratio)	frameaspectratio is the ratio of the width to the height of the desired image.
RiFrameBegin (frame) **RiFrameEnd** ()	marks the beginning and end of a single frame of an animated sequence.
RiGeometricApproximation (type, value)	The predefined geometric approximation is "flatness".
RiHider (type, parameterlist)	The standard types are "hidden", "paint", and "null".
RiIdentity ()	sets the current transformation to the identity.
RiIlluminate (light, onoff)	If onoff is RI_TRUE and the light source referred to by the RtLightHandle is not currently in the current light source list, add it to the list.
RiImager (name, parameterlist)	selects an imager function programmed in the Shading Language.
RiInterior (name, parameterlist)	sets the current interior volume shader.
RiLightSource (name, parameterlist)	creates a non-area light, turns it on, and adds it to the current light source list.
RiMatte (onoff)	indicates whether subsequent primitives are matte objects.

Graphics State (continued)	
Function	**Description**
RiOpacity (color)	sets the current opacity to color.
RiOption (name, parameterlist)	sets additional implementation-specific options.
RiOrientation (orientation)	sets the current orientation to be either left-handed or right-handed.
RiPerspective (fov)	concatenates a perspective transformation onto the current transformation.
RiPixelFilter (filterfunc, xwidth, ywidth)	performs antialiasing by filtering the geometry (or supersampling) and then sampling at pixel locations.
RiPixelSamples (xsamples, ysamples)	sets the effective sampling rate in the horizontal and vertical directions
RiPixelVariance (variation)	sets the amount computed image values are allowed to deviate from the true image values.
RiProjection (name, parameterlist)	sets the type of projection and marks the current coordinate system before projection as the camera coordinate system.
RiQuantize (type, one, min, max, ditheramplitude)	sets the quantization parameters for colors or depth.
RiRelativeDetail (relativedetail)	The relative level of detail scales the results of all level of detail calculations.
RiReverseOrientation ()	causes the current orientation to be toggled.
RiRotate (angle, dx, dy, dz)	concatenates a rotation of angle degrees about the given axis onto the current transformation.
RiScale (sx, sy, sz)	concatenates a scaling onto the current transformation.
RiScreenWindow (left, right, bottom, top)	defines a rectangle in the image plane that gets mapped to the raster coordinate system and that corresponds to the display area selected.
RiShadingInterpolation ()	controls how values are interpolated between shading samples (usually across a polygon).
RiShadingRate (size)	sets the current shading rate to size.
RiShutter (min, max)	sets the times at which the shutter opens and closes.
RiSides (sides)	If sides is 2, subsequent surfaces are considered two-sided and both the inside and the outside of the surface will be visible.
RiSkew (angle, dx1, dy1, dz1, dx2, dy2, dz2)	concatenates a skew onto the current transformation.

Graphics State (continued)	
Function	**Description**
RiSurface (name, parameterlist)	sets the current surface shader. name is the name of a surface shader.
RiTextureCoordinates (s1, t1, s2, t2, s3, t3, s4, t4)	sets the current set of texture coordinates to the values passed as arguments.
RiTransform (transform)	sets the current transformation to the transformation *transform*.
RiTransformBegin () **RiTransformEnd** ()	saves and restores the current transformation.
RiTransformPoints (fromspace, tospace, n, points)	transforms the array of points from the coordinate system fromspace to the coordinate system tospace.
RiTranslate ()	concatenates a translation onto the current transformation.
RiWorldBegin () **RiWorldEnd** ()	Starts and ends the description of the scene geometry for a specific image.

Geometric Primitives	
Function	**Description**
RiBasis (ubasis, ustep, vbasis, vstep)	sets the current u-basis to *ubasis* and the current v-basis to *vbasis*.
RiBlobby (nleaf, ncode, code, nfloats, floats, nstrings, strings, ...)	requests an implicit surface.
RiCone (height, radius, thetamax, parameterlist)	requests a cone.
RiCurves (type, ncurves, nvertices, wrap, parameterlist)	requests a collection of lines, curves, or ribbons.
RiCylinder (radius, zmin, zmax, thetamax, parameterlist)	requests a cylinder.
RiDisk (height, radius, thetamax, parameterlist)	requests a disk.
RiGeneralPolygon (nloops, nverts, parameterlist)	defines a general planar concave polygon with holes.
RiGeometry (type, parameterlist)	provides a standard way of defining an implementation-specific geometric primitive.
RiHyperboloid (point1, point2, thetamax, parameterlist)	requests a hyperboloid.
RiNuPatch (nu, uorder, uknot, umin, umax, nv, vorder, vknot, vmin, vmax, parameterlist)	creates a single tensor product rational or polynomial non-uniform B-spline surface patch.

Geometric Primitives (continued)	
Function	**Description**
RiObjectBegin () **RiObjectEnd** ()	begins and ends the definition of an object.
RiObjectInstance (handle)	creates an instance of a previously defined object.
RiParaboloid (rmax, zmin, zmax, thetamax, parameterlist)	requests a paraboloid.
RiPatch (type, parameterlist)	define a single bilinear or bicubic patch.
RiPatchMesh (type, nu, uwrap, nv, vwrap, parameterlist)	specifies in a compact way a quadrilateral mesh of patches.
RiPoints (npoints, parameterlist)	requests a collection of point-like particles.
RiPointsPolygons (npolys, nverts, parameterlist)	defines *npolys* planar convex polygons that share vertices.
RiPointsGeneralPolygons (npolys, nloops, nverts, verts, parameterlist)	defines *npolys* planar concave polygons, with holes, that share vertices.
RiPolygon (parameterlist)	*nverts* is the number of vertices in a single closed planar convex polygon. *parameterlist* is a list of token-array pairs where each token is one of the standard geometric primitive variables or a variable which has been defined with **RiDeclare**.
RiProcedural (parameterlist)	defines a procedural primitive.
RiSolidBegin (operation) **RiSolidEnd** ()	starts and ends the definition of a CSG solid primitive.
RiSphere (radius, zmin, zmax, thetamax, parameterlist)	requests a sphere.
RiSubdivisionMesh (scheme, nfaces, nvertices, vertices, ntags, tags, nargs, intargs, floatargs, parameterlist)	requests a subdivision surface mesh.
RiTorus (majorradius, minorradius, phimin, phimax, thetamax, parameterlist)	requests a torus.
RiTrimCurve (order, knot, min, max, n, u, v, w)	sets the current trim curve.

Motion	
Function	**Description**
RiMotionBegin (n, t0, t1, ..., tnminus1) **RiMotionEnd** ()	starts and ends the definition of a moving primitive.

Texture Map Utilities	
Function	**Description**
RiMakeCubeFaceEnvironment (px, nx, py, ny, pz, nz, texturename, fov, filterfunc, swidth, twidth, parameterlist)	converts six images in a standard picture file representing six viewing directions into an environment map whose name is *texturename*.
RiMakeLatLongEnvironment (picturename, texturename, filterfunc, swidth, twidth, parameterlist)	converts an image in a standard picture file representing a latitude-longitude map whose name is *picturename* into an environment map whose name is *texturename*.
RiMakeShadow (picturename, texturename, parameterlist)	creates a depth image file named *picturename* into a shadow map whose name is *texturename*.
RiMakeTexture (picturename, texturename, swrap, twrap, parameterlist)	converts an image in a standard picture file whose name is *picturename* into a texture file whose name is *texturefile*.

External Resources	
Function	**Description**
RiErrorHandler (handler)	sets the user error handling procedure.
RiArchiveRecord (type, format, ...)	writes a user data record into a RIB archive file.
RiReadArchive (name, callback, ...)	reads RIB from an archive file.

Math Functions	
Function	**Description**
abs(x)	returns the absolute value of its argument.
acos(a)	returns the arc cosine in the range 0 to π.
asin(a)	returns the arc sine in the range $-\pi/2$ to $\pi/2$.
atan(yoverx), atan(y,x)	with one argument returns the arc tangent in the range $-\pi/2$ to $\pi/2$ (1 argument) or $-\pi$ to π (2 arguments).
ceil(x)	returns the largest integer (expressed as a float) not greater than *x*.
cellnoise(v), cellnoise(u,v), cellnoise(pt), cellnoise(pt,t)	returns a value which is a pseudorandom function of its arguments and is constant within each cell defined by integer lattice points, but discontinuous at integer values. Its value is always between 0 and 1. The domain of this function can be 1-D (one float), 2-D (two floats), 3-D (one point), or 4-D (one point and one float).
clamp(a,min,max)	returns min if $a < min$, max if $a > max$; otherwise a.
cos(a)	standard trigonometric function of radian arguments.
degrees(rad)	converts from radians to degrees.
Du(p), Dv(p), Deriv(num,den)	computes the derivatives of the arguments. The type returned depends on the type of the first argument. Du and Dv compute the derivatives in the u and v directions, respectively. Deriv computes the derivative of the first argument with respect to the second argument.
exp(x)	returns pow(e,x).
filterstep(edge,s1,...)	returns a filtered version of step.
floor(x)	returns the smallest integer (expressed as a float) not smaller than x.
log(x), log(x,base)	returns the natural logarithm of x (x=log(exp(x))) (1 arg.) or the logarithm in the specified base (x=log(pow(base,x),base)) (2 args).
max(a,b,...)	returns the argument with maximum value.
min(a,b,...)	returns the argument with minumum value.
mix(color0, color1, value)	returns a linearly interpolated color value.
mod(a,b)	returns 0 <= mod < b such that mod(a, b) = a - nb for an integer n.

Math Functions (continued)	
Function	**Description**
noise(v), noise(u,v), noise(pt), noise(pt,t)	returns a value which is a random function of its arguments. Its value is always between 0 and 1. The domain of this noise function can be 1-D (one float), 2-D (two floats), 3-D (one point), or 4-D (one point and one float).
pnoise(v,vp), pnoise(u,v,up,vp), pnoise(pt,ptp), pnoise(pt,t,ptp,tp)	returns a value like noise, but with the given periods over its domain.
pow(x,y)	returns x^y.
radians(deg)	converts from degrees to radians.
random()	returns a float, color, or point whose components are a random number between 0 and 1.
round(x)	returns the integer closest to x.
sign(x)	returns -1 with a negative argument, 1 with a positive argument, and 0 if its argument is zero.
sin(a)	standard trigonometric function of radian arguments.
smoothstep(min, max, value)	returns 0 if value $<$ min, 1 if value $>$ max, and performs a smooth Hermite interpolation between 0 and 1 in the interval min to max.
spline([basis,] value, f1, f2, ... fn	fits a spline to the control points given. At least four control points must always be given.
spline([basis,] value, array)	fits a spline to the control points given in the array.
sqrt(x)	returns pow(x,.5).
step(min,value)	returns 0 if value $<$ min; otherwise 1.
tan(a)	standard trigonometric function of radian arguments.

Geometric Functions	
Function	**Description**
area(P)	returns the differential surface area.
calculatenormal(P)	returns surface normal given a point on the surface.
depth(P)	returns the depth of the point P in camera coordinates. The depth is normalized to lie between 0 (at the near clipping plane) and 1 (at the far clipping plane).
distance(p1, p2)	returns the distance between two points.
faceforward(N, I)	flips N so that it faces in the direction opposite to I.
fresnel(I, N, eta, Kr, Kt [, R, T])	returns the reflection coefficient Kr and refraction (or transmission) coefficient Kt given an incident direction I, the surface normal N, and the relative index of refraction eta. Optionally, this procedure also returns the reflected (R) and transmitted (T) vectors.
length(V)	returns the length of a vector.
normalize(V)	returns a unit vector in the direction of V.
ptlined(Q, P1, P2)	returns the distance from Q to the line segment joining P1 and P2.
transform(fromspace, tospace, P)	transforms the point P from the coordinate system fromspace to the coordinate system tospace. If fromspace is absent, it is assumed to be the "current" coordinate system.
vtransform(fromspace, tospace, V)	transforms the vector V from the coordinate system fromspace to the coordinate system tospace. If fromspace is absent, it is assumed to be the "current" coordinate system.
ntransform(fromspace, tospace, N)	transforms the normal N from the coordinate system fromspace to the coordinate system tospace. If fromspace is absent, it is assumed to be the "current" coordinate system.
reflect(I, N)	returns the reflection vector given an incident direction I and a normal vector N.
refract(I, N, eta)	returns the transmitted vector given an incident direction I, the normal vector N and the relative index of refraction eta.
setxcomp(P, x), setycomp(P, y), setzcomp(P, z)	sets the x, y, or z component.
xcomp(P), ycomp(P), zcomp(P)	gets the x, y, or z component.

Color Functions	
Function	**Description**
comp(c, index)	gets individual color component.
setcomp(c, index, value)	sets individual color component.

String Functions	
Function	**Description**
printf(format, val1, val2, ..., valn)	Prints the values of the specified variables on the standard output stream of the renderer. *format* uses "%f", "%p", "%c", and "%s" to indicate float, point, color and string, respectively.
format(pattern, val1, val2, ..., valn)	Returns a formatted string (using the same rules as printf).
concat(str1, ..., strn)	Returns a concatenated string.
match(pattern, subject)	String pattern matching.

Shading and Lighting Functions	
Function	**Description**
ambient()	returns the total amount of ambient light incident upon the surface.
diffuse(N)	returns the diffuse component of the lighting model.
phong(N, V, size)	implements the Phong specular lighting model.
specular(N, V, roughness)	returns the specular component of the lighting model. *N* is the normal to the surface. *V* is a vector from a point on the surface towards the viewer.
specularbrdf(L, N, V, roughness)	returns the specular reflection contribution of a particular light direction.
trace(P, R)	returns the incident light falling on a point *P* in a given direction *R*.

Texture Mapping Functions	
Function	**Description**
environment(name[channel], texture coordinates [, parameterlist])	accesses an environment map.
shadow(name[channel], texture coordinates [,parameterlist])	accesses a shadow depth map.
texture(name[channel] [,texture coordinates] [, parameterlist])	accesses a basic texture map.
textureinfo(texturename, paramname, variable)	gets information about a texture map.

Message Passing Functions	
Function	**Description**
atmosphere(name, variable)	looks up the value of a variable that is stored in the atmosphere shader attached to the geometric primitive surface.
displacement(name, variable)	looks up the value of a variable that is stored in the displacement shader attached to the geometric primitive surface.
incident(name, variable)	looks up the value of a variable that is stored in the volume shaders attached to geometric primitive surface, on the same side as the incident ray.
lightsource(name, variable)	looks up the value of a variable that is stored in a light shader attached to the geometric primitive surface.
opposite(name, variable)	looks up the value of a variable that is stored in the volume shaders attached to geometric primitive surface, on the opposite side as the incident ray.
surface(name, variable)	looks up the value of a variable that is stored in the surface shader attached to the geometric primitive surface.

Renderer State Functions	
Function	**Description**
attribute(name, variable)	looks up the value of a renderer attribute.
option(name, variable)	looks up the value of a renderer option.
rendererinfo(name, variable)	looks up information about the renderer implementation itself.

Appendix H

LIST OF RENDERMAN INTERFACE PROCEDURES

Appendix I

DIFFERENCES BETWEEN VERSION 3.2 AND 3.1

Several new API calls have been added to the RenderMan Interface:

- New geometric primitives: **RiSubdivisionMesh**, **RiPoints**, **RiCurves**, **RiBlobby**.
- New built-in procedural routines suitable to use as **RiProcedural** primitives: **RiProcDelayedReadArchive**, **RiProcRunProgram**, **RiProcDynamicLoad**.
- Reading an existing RIB archive into the renderer at an arbitrary point in the command stream. **RiReadArchive**.
- New routines affecting the graphics state: **RiCoordSysTransform**, **RiClippingPlane**.

There is no longer an approved K&R C binding of the RenderMan Interface. Instead, only an ANSI C binding is described. All interfaces and examples are now shown in ANSI C in this document.

The following data types have been added to the RenderMan Interface: vector, normal, matrix, hpoint, and to ri.h (as **RtVector**, **RtNormal**, and **RtHpoint**). The ri.h header file also now defines an RtInt to be an int, not a long.

The storage classes vertex and constant have been added to the RenderMan Interface. A vertex primitive variable must supply the same number of data elements as are supplied for the position, "P", and interpolation is performed in the same manner as position. A constant primitive variable supplies a single data value for an entire geometric primitive (even an aggregate primitive, like a **PointsPolygons**). Descriptions of all the geometric primitives have been updated to explain the expected number of data elements for each storage class.

"NDC" space is now a standard space known to an implementation.

Clarified that output depth values are camera space z values, not screen space values.

Clarified that the *type* parameter of **RiDisplay** is not limited to be "file" or "framebuffer", but may be any format or device type name supported by a particular implementation. The "file" and "framebuffer" names select the implementation's default file format or framebuffer device.

The RenderMan Interface 3.1 Specification was silent on the number of uniform and varying values supplied for primitive variables on **RiNuPatch** primitives. The number of varying

and uniform variables on an **RiNuPatch** has now officially been deemed to be computed as if the **NuPatch** is a nonperiodic uniform B-spline mesh, rather than a single B-spline patch. Details are given in Section 5.2 where the **RiNuPatch** primitive is described.

The 3.1 spec implied that Shading Language variable initialization expressions could only be uniform. That restriction is lifted.

The Shading Language variables du and dv are potentially varying, and are not restricted to be uniform as the 3.1 specification suggests.

A number of new optional arguments to the Shading Language texturing functions have been added: "blur", "sblur", "tblur", "filter", "fill".

Many new built-in functions, or new variants of existing functions, have been added to the Shading Language, including: inversesqrt, random, transform, vtransform, ntransform, determinant, concat, format, match, translate, rotate, scale, ctransform, ptlined, filterstep, noise, cellnoise, pnoise, specularbrdf, attribute, option, textureinfo, rendererinfo, min, max, clamp, mix, spline, atmosphere, displacement, lightsource, surface, Dtime.

The illuminance statement has been expanded to accept category specifiers.

Shading Language has added new variables: dtime and dPdtime.

Rather than strictly requiring user-defined and nonstandard data to be typed using **RiDeclare**, such tokens can have their types specified "in-line" by prepending the type declaration to the token name.

Renderers may output images composed of arbitrary data computed by shaders, in addition to "rgb" and so on. Such extra data may also be sent to different output files.

We have clarified that interior and exterior volume shaders are not specific to CSG solids, but rather are shaders that alter the colors of rays spawned by trace() calls in the shaders of the primitive.

RiArchiveRecord now takes "verbatim" as the record type.

We have clarified that in imager shaders, P is the raster space position of the pixel center, and not the point on any piece of scene geometry.

We have changed the wording of the section that once described the required versus optional capabilities of RenderMan-compliant renderers. A few features previously described as optional, such as programmable shading, texture mapping, and trim curves, have been moved unambiguously to the list of requirements. There is still a list of "advanced features" that are no longer referred to as "optional," but it is understood that algorithmic limitations may prevent some implementations from fully supporting those features.

Object Instances are a bit more flexible in 3.2 than they were in 3.1. In particular, relative transformations and Motion blocks are allowed within Object definitions.

Some changes have been made to the required standard shaders: they have been updated to make use of the new data types, the bumpy shader now does displacement instead of using the depricated bump() function, and there is now a standard background imager shader.

A few features previously described are now depricated due to either having been ambiguously articulated, useless, unimplementable, or having been replaced by more recent and obviously superior features. These are therefore removed from the RenderMan Interface:

- Deformations and transformation shaders have been eliminated. We now recognize that this functionality is not only ambiguous, but should be considered a modeling feature, not a rendering feature.
- Bump maps (including the **RiMakeBump** and the Shading Language `bump()` function) have been removed. This functionality is completely subsumed by displacement mapping.
- Support for "Painter's algorithm" hidden surface removal is no longer required.

Statement About Pixar's Copyright and Trademark Rights for the RenderMan 3-D Scene Description Interface

The RenderMan 3-D Scene Description Interface, created by Pixar, is used for describing three-dimensional scenes in a manner suitable for photorealistic image synthesis. The RenderMan Interface specifies procedure calls that are listed collectively in the Procedures List appendix (Appendix F) to the RenderMan Interface document. The RenderMan Interface Bytestream ("RIB") is a byte-oriented protocol for specifying requests to the RenderMan Interface, and specifies a set of encoded requests according to the methods described in the RenderMan Interface document.

Pixar owns the copyrights in the RenderMan Interface and RIB including the Procedures List, Binary Encoding table and the RenderMan written specifications and manuals. These may not be copied without Pixar's permission. Pixar also owns the trademark "RenderMan".

Pixar will enforce its copyrights and trademark rights. However, Pixar does not intend to exclude anyone from:

(a) creating modeling programs that make RenderMan procedure calls or RIB requests.

(b) creating rendering systems that execute the RenderMan procedure calls or RIB requests provided a separate written agreement is entered into with Pixar.

Permitted Use of the RenderMan Interface by Modelers

Pixar gives permission for you to copy the procedure calls included the Procedures List and the encoded requests in the Binary Encoding table for writing modeling programs that use the RenderMan Interface. Any program that incorporates any of the RenderMan procedure calls or RIB requests must include the proper copyright notice on each program copy. The copyright notice should appear in a manner and location to give reasonable notice of Pixar's copyright, as follows:

The right to copy the RenderMan procedures from the Procedures List does not include the right to copy the RenderMan documentation or manuals, or the programming code in any Pixar products, in whole or in part, in any manner except as described above.

Written License for Use of the RenderMan Interface by Renderers

A no-charge license is available from Pixar for anyone who wishes to write a renderer that uses the Pixar RenderMan procedure calls or RIB requests. This license must be in writing.

Limited Use of the Trademark "RenderMan"

The trademark "RenderMan" should refer only to the scene description interface created by Pixar. Anyone that creates a routine or computer program that includes any of the

procedure calls from the RenderMan Procedures List statement or RIB requests from the Binary Encoding table may refer to the computer program as "using" or "adhering to" or "compatible with" the RenderMan Interface, if that statement is accurate. Any such reference must be accompanied by the following legend:

RenderMan® is a registered trademark of Pixar

No-one may refer to or call a product or program which did not originate with Pixar a "RenderMan program" or "RenderMan modeler" or "RenderMan renderer."

Nothing in this statement shall be construed as granting a license or permission of any type to any party for the use of the trademark RenderMan, or any thing confusingly similar to it, in connection with any products or services whatsoever, including, but not limited to, computer hardware, software or manuals. Use of the trademark "RenderMan" should follow the trademark use procedure guidelines of Pixar. Any use of the RenderMan Interface, RIB and related materials other than as described in this statement is an unauthorized use and violates Pixar's proprietary rights, and Pixar will enforce its rights to prevent such use.