Quick RenderMan Interface and Implementation Specification

Pixar

paste.tiff ¬

This is the 0.1 Beta version of the Quick RenderMan* Interface and Implementation Specification. In conjunction with version 3.1 of the Pixar RenderMan Interface Specification, this document represents the state of the application interface to Quick RenderMan at the time of the release. The reader is cautioned that the Quick RenderMan software and its application interface are in transition, and that any information appearing herein may become obsolete in the future without warning from Pixar or its affiliates.

This document is a technical specification, and, as such, is quite terse and requires substantial knowledge of computer graphics in general and photorealistic image synthesis in particular. The reader is also assumed to be familiar with the RenderMan Interface. For an introduction to RenderMan, the reader is referred to *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics* (Steve Upstill, 1989).

How to use this document

This document should be used in conjunction with Version 3.1 of the RenderMan Interface Specification. RenderMan features defined in the RenderMan 3.1 specification and implemented in Quick RenderMan per that specification are not discussed here; refer to the 3.1 Interface Specification for descriptions of these features. Features defined in the RenderMan 3.1 specification and implemented differently (or not implemented at all) within Quick RenderMan are discussed here, along with new features that do not appear in the 3.1 specification.

Quick RenderMan Concepts

Contexts

As an extension to RenderMan Version 3.1, Quick RenderMan introduces the notion of a *context*, which is a rendering environment possessing its own renderer, graphics state, and output medium. An application program may create multiple contexts and switch among them during the course of its execution. At any time, at most one context within an application may be *active*. Several *extracontextual* commands are available to control the creation, selection, and destruction of contexts. All other Quick RenderMan commands implicitly affect the currently active context.

Separation of Front End and Renderer

Quick RenderMan has a modular organization: multiple *renderers* are driven by one *front end*. The front end is responsible for receiving RenderMan commands from the application program and performing basic validation of the commands and their arguments. As a part of the image synthesis process, the front end will invoke a renderer to create the image. Quick RenderMan presently offers only one image renderer, referred to as the Quick Renderer. Alternatively, the application may direct the output from a context into a RIB archive in lieu of producing an image, so the RIB writer within Quick RenderMan is, in a sense, a renderer. When discussing the status of the implementation of Quick RenderMan features, this document may draw a distinction between the respective contributions of both the front end and the renderer to the operation of those features.

Quick RenderMan Command Summary

The following lists summarize the present status of each command in the Quick RenderMan implementation. The next section discusses in greater detail the differences between Quick RenderMan and the 3.1 specification. Unless otherwise indicated, all commands have entry points in Quick RenderMan and perform some validation of their arguments.

RenderMan 3.1 Commands

Quick RenderMan support of RenderMan Version 3.1 commands is summarized here, breaking it into separate categories for front-end, renderer, and RIB output.

The following abbreviations are used:

Full	Fully implemented at the indicated level
Limited	Partially implemented
Incorrect	Effects may occur, but are likely to be incorrect
Geo	Fully implemented, with the exception of certain standard shaders and all programmable shaders (a special case of "Limited")
NA	No action is taken other than argument validation
±	Not applicable

Numbers in parentheses refers to the implementation notes following the list of commands.

A "`I`" preceding the name of a command indicates the Quick RenderMan implementation of the

command is syntactically and/or semantically not backward-compatible with RenderMan Version 3.1. An "**E**" preceding the command name indicates that the syntax of the command are extended beyond the RenderMan 3.1 specification in a backward-compatible way.

	Command Name	Front-End	Quick Renderer	RIB Output
	ArchiveRecord	Full	±	Full
I	**AreaLightSource**	Full	Limited(2,18)	Full
	Atmosphere	Full	Full(18)	Full
	Attribute	Full	Full	Full
I	**AttributeBegin**	Full	Full	Full
I	**AttributeEnd**	Full	Full	Full
	Basis	Full	Full	Full
I	**Begin**	Full	Full	±
	Bound	Full	NA(3)	Full
	Clipping	Full	Full	Full
	Color	Full	Full	Full
	ColorSamples	Full	Full(15)	Full
	ConcatTransform	Full	Full	Full
	Cone	Full	Geo	Full
	CoordinateSystem	NA	NA(5)	Full
	CropWindow	Full	Full	Full
	Cylinder	Full	Geo	Full
E	**Declare**	Full	Full	Full
	Deformation	NA	NA(9,18)	NA
	DepthOfField	Full	NA(7)	Full
	Detail	Full	Limited(13,25)	Full
	DetailRange	Full	Full(13)	Full
	Disk	Full	Geo	Full
	Displacement	NA	NA(4,8,18)	Full
	Display	Full	Limited(24)	Full
I	**End**	Full	Full	±
	ErrorAbort	Full	±	±
	ErrorDefault	Full	±	±
	ErrorHandler	Full	±	±
	ErrorIgnore	Full	±	±
	ErrorPrint	Full	±	±
	Exposure	Full	Full	Full
	Exterior	NA	NA(18,23)	Full
	Format	Full	Full	Full
	FrameAspectRatio	Full	Full	Full
	FrameBegin	Full	Full	Full
	FrameEnd	Full	Full	Full
	GeneralPolygon	Full	Geo	Full
E	**GeometricApproximation**	Full	Limited(12)	Full
	Geometry	Full	Full	Full

	Hider	Full	Limited(11)	Full
	Hyperboloid	Full	Geo	Full
	Identity	Full	Full	Full
I	**Illuminate**	Full	Full	Full
	Imager	NA	NA(18)	Full
	Interior	NA	NA(18,19,23)	Full
I	**LightSource**	Full	Limited(18)	Full
	MakeBump	NA	NA(4,8)	Incorrect
	MakeCubeFaceEnvironment	NA	NA(10)	Incorrect
	MakeLatLongEnvironment	NA	NA(10)	Incorrect
	MakeShadow	NA	NA(17)	Incorrect
	MakeTexture	NA	NA(22)	Incorrect
	Matte	Full	NA(11)	Full
	MotionBegin	Full	Limited(14)	Full
	MotionEnd	Full	Limited(14)	Full
	NuPatch	Full	Geo	Full
I	**ObjectBegin**	Full	Full	Incorrect
I	**ObjectEnd**	Full	Full	Incorrect
I	**ObjectInstance**	Full	Full	Full
	Opacity	Full	NA(11)	Full
	Option	Full	Full	Full
	Orientation	Full	Full	Full
	Paraboloid	Full	Geo	Full
	Patch	Full	Geo	Full
	PatchMesh	Full	Geo	Full
	Perspective	Full	Full	Full
	PixelFilter	Full	NA(1)	Full
	PixelSamples	Full	NA(1)	Full
	PixelVariance	Full	NA(1)	Full
	PointsGeneralPolygons	Full	Geo	Full
	PointsPolygons	Full	Geo	Full
	Polygon	Full	Geo	Full
	Procedural	Full	NA	±
	Projection	Full	Geo(9,16,18)	Full
	Quantize	Full	Limited(6)	Full
	RelativeDetail	Full	Full(13)	Full
	ReverseOrientation	Full	Full	Full
	Rotate	Full	Full	Full
	Scale	Full	Full	Full
	ScreenWindow	Full	Full	Full
	ShadingInterpolation	Full	Full	Full
	ShadingRate	Full	NA(20)	Full
	Shutter	Full	NA(14)	Full
	Sides	Full	Full	Full
	Skew	Full	Full	Full
	SolidBegin	Full	NA(19)	Full

SolidEnd	Full	NA(19)	Full
Sphere	Full	Geo	Full
Surface	Full	Limited(10,18,22)	Full
TextureCoordinates	Full	NA(4,10,22)	Full
Torus	Full	Geo	Full
Transform	Full	Full	Full
TransformBegin	Full	Full	Full
TransformEnd	Full	Full	Full
TransformPoints	Full	Limited(5)	±
Translate	Full	Full	Full
TrimCurve	Full	NA(21)	Full
WorldBegin	Full	Full	Full
WorldEnd	Full	Full	Full

IMPLEMENTATION NOTES

1 Quick RenderMan performs no antialiasing.
2 Quick RenderMan does not support the optional *Area Light Source* capability, and therefore treats this command as equivalent to **LightSource**, per the 3.1 specification.
3 Quick RenderMan does not take advantage of a bound specification in culling.
4 Quick RenderMan does not support the optional *Bump Mapping* capability, and thus treats the required standard "bumpy" shader as a null displacement shader, per the 3.1 specification.

5 Quick RenderMan does not support user-defined coordinate systems.
6 Quick RenderMan does not perform dithering.
7 Quick RenderMan does not support the optional *Depth of Field* capability, and thus treats all depth of field settings as equivalent to a pin-hole, per the 3.1 specification.
8 Quick RenderMan does not support the optional *Displacement Shading* capability.
9 Quick RenderMan does not support the optional *Deformations* capability, and thus treats all deformations as equivalent to concatenating an identity transformation, per the 3.1 specification.
10 Quick RenderMan does not support the optional *Environment Mapping* capability, nor does it support the required standard "shinymetal" surface shader.
11 Quick RenderMan employs a z-buffer hider for the "hidden" hider, so it does not support transparency.
12 Quick RenderMan does not honor the "deviation" type correctly.
13 Quick RenderMan does support the optional *Level of Detail* capability; specifically, any representation with a detail level greater than one half is rendered.
14 Quick RenderMan does not support the optional *Motion Blur* capability, and therefore heeds only the first command within a motion block.
15 Quick RenderMan does not support the optional *Spectral Colors* capability, and thus converts all colors to RGB space on input.
16 Quick RenderMan does not support the optional *Special Camera Projections* capability, and thus treats all special projections as equivalent to orthographic.
17 Quick RenderMan does not support the optional *Shadow Depth Mapping* capability.
18 Quick RenderMan does not implement the RenderMan shading language or programmable shaders.
19 Quick RenderMan does not support the optional *Solid Modeling* capability, and thus treats

all solid set operations as being equivalent to `"union"`, per the 3.1 specification.

20 Quick RenderMan ignores the shading rate. It shades once per polygon for `"constant"` shading interpolation, once per vertex for `"smooth"`, and never for `"none"`.

21 Quick RenderMan does not support the optional *Trim Curves* capability, and thus ignores all trim curves, per the 3.1 specification.

22 Quick RenderMan does not support the optional *Texture Mapping* capability, nor does it support the required standard `"paintedplastic"` surface shader.

23 Quick RenderMan does not support the optional *Volume Shading* capability, and thus ignores interior and exterior volume shaders.

24 **`RiDisplay`** ignores the *mode* parameter and always outputs an `"rgba"` image with *alpha* = 1.0.

25 The raster space bounding box is not clipped by the near clipping plane.

New Commands Introduced In Quick RenderMan

Command Name	Front-End	Quick Renderer	RIB Output
`Circle`	Full	NA	Full
`Context`	Full	±	±
`CreateHandle`	Full	Full	Full
`Curve`	Full	NA	Full
`GeometricRepresentation`	Full	Full	Full
`Line`	Full	NA	Full
`MacroBegin`	Full	Full	Full
`MacroEnd`	Full	Full	Full
`MacroInstance`	Full	Full	Full
`NuCurve`	Full	NA	Full
`PointsLines`	Full	NA	Full
`ReadArchive`	Full	Full	Full
`Resource`	Full	Full	Full
`Synchronize`	Full	Full	Full

Quick RenderMan Command Descriptions

This section describes in detail the differences between commands appearing in Version 3.1 of the RenderMan Interface and the Quick RenderMan command set. Special attention should be paid to the IMPLEMENTATION STATUS, as Quick RenderMan does not completely implement some commands yet.

```
RtToken
RiAreaLightSource(handle, shader, parameterlist)
   RtToken handle;
   RtToken shader;
```

Make *shader* the *current area light shader*. **RiAreaLightSource** defines an area light source of the given shader name and type, and adds it to the *current light source list*, thus turning it on. *shader* is the name of a predefined area light shader, or a user-defined area light shader previously declared with **RiResource**. Each subsequent geometric primitive until the next area light definition or the end of the attribute block is added to the list of surfaces that define the area light source. If *shader* is "null" (the default), the geometric primitives do not emit light.

Light sources defined with **RiAreaLightSource** are attributes, and are stackable with **RiAttributeBegin**/**RiAttributeEnd**.

The area light source is implicitly instantiated (and thus potentially self-illuminating) when it is defined, and implicitly disinstantiated at the end of the attribute block in which it is defined; in addition, it can be explicitly disinstantiated or reinstantiated with **RiIlluminate**.

RIB BINDING

```
AreaLightSource handle shader parameterlist
```

For backward compatibility, RenderMan 3.1 RIB syntax is also accepted:

```
AreaLightSource shader sequencenumber parameterlist
```

EXAMPLE

```
RtToken myarealight, amb;
RtFloat intensity = 0.5;
amb = RiResource("ambientlight", RI_LIGHTSOURCE, RI_NULL);
myarealight = RiAreaLightSource("myalight", amb,
    "intensity", &intensity, RI_NULL);

AreaLightSource "pin" "ambientlight" "intensity" [0.5]
```

SEE ALSO

RiLightSource, RiResource, RiIlluminate, RiCreateHandle

IMPLEMENTATION STATUS

Quick RenderMan does not support the *Area Light Source* capability, so **RiAreaLightSource** has the same effect as **RiLightSource**.

RiAttributeBegin()
RiAttributeEnd()

Push and pop the current set of attributes and the current transformation. When invoked outside a world block, **RiAttributeBegin** pushes (and **RiAttributeEnd** pops) the current options in addition to attributes and transforms. Although the pushing and popping of options is technically not backward-compatible with RenderMan 3.1, it is likely that this will not be a problem while porting applications developed under RenderMan 3.1 to Quick

RenderMan.

RIB BINDING
AttributeBegin
AttributeEnd

EXAMPLE
```
RiAttributeBegin();
```

SEE ALSO
RiFrameBegin, RiTransformBegin, RiWorldBegin

```
RtToken
RiBegin(handle, parameterlist)
   char *handle;

RiEnd()
```

The bracketed set of commands **RiBegin-End** initialize and terminate a rendering session, or *context*. *handle* gives the new context a specific name which may be used to refer to this context in later **RiContext** calls. It is illegal for any single client to create two contexts with the same name. If *handle* is `RI_NULL`, Quick RenderMan will create a unique context handle.

When **RiBegin** is called, the currently active context (if any) is suspended, a new context is created, and all graphics state variables in the new context are set to their default values. The new context then becomes the active context. When **RiEnd** is called, the currently active context is terminated, all resources devoted to the context are released, and any cleanup operations that need to be done are performed. Following an **RiEnd** call, no context is active. The application may call **RiContext** at any time to activate a suspended context, or call **RiBegin** to create a new context.

All other RenderMan Interface procedures must be called within an **RiBegin-RiEnd** block and are specific to the currently active context; the only exceptions to this rule are **RiErrorHandler** and **RiContext**, which perform extracontextual services.

RiBegin accepts parameters that select the identity of the renderer to be run and the basic initialization of that renderer.

`"renderer"` selects among various renderer implementations that may be available. Quick RenderMan presently accepts only two values, `"draft"` and `"archive"`. The latter produces no image. If the `"renderer"` parameter is not specified, Quick RenderMan uses the `"draft"` renderer by default.

`"filepath"` is an **RtToken** value specifying the name of an output archive file into which a RIB archive should be written. This parameter is meaningful only when the `"archive"` renderer is selected. If the `"archive"` renderer

is selected and "filepath" is not specified, the RIB output is written to the default file name **ri.rib** in the current directory.

"format" is an **RtToken** specifying the type of output archive; it is used in conjunction with "filepath". The valid values are "asciifile", which specifies ASCII RIB output, and "binaryfile", which specifes binary RIB output. If "format" is not specified, ASCII RIB is output by default.

RiBegin returns a tokenized version of the context handle, which can be used to switch among existing contexts, or RI_NULL if the context cannot be created for any reason. **RiBegin** can be called at any time to create a new context, since it is completely independent of the graphics state of any currently active context.

EXAMPLE

```
RtToken c1, c2;
char *filename = "myfile.rib", *archive = "archive";

c1 = RiBegin("c1", "renderer", &archive,
             "filepath", &filename, RI_NULL);
/* context "c1" is now active */
c2 = RiBegin("c2", RI_NULL);
/* context "c2" is now active */
RiEnd();   /* terminates context "c2" */
/* no context is active */
RiContext(c1, RI_NULL);
/* context "c1" is now active */
RiEnd();   /* terminates context "c1" */
/* no context is active */
```

SEE ALSO

RiContext

RiCircle(radius, thetamax, parameterlist)

```
RtFloat radius, thetamax;
```

Requests a circular arc defined by the following equations:

```
theta = u * thetamax
x = radius * cos(theta)
y = radius * sin(theta)
z = 0.0
```

parameterlist is a list of token-value pairs where each token is one of the standard geometric primitive variables applicable to lines, or a variable that has been defined with **RiDeclare**. Position variables should not be given. If a geometric primitive variable is varying, it contains two values, one for each arc endpoint. If a variable is uniform, it contains a single element. The angle *theta* is measured in degrees.

RIB BINDING

```
Circle radius thetamax parameterlist
```

EXAMPLE

```
RiCircle(2.0, 180.0, RI_NULL);

Circle 1 290 "Cs" [1 0 0  0 0 1]
```

SEE ALSO

RiDisk

RtToken
RiContext(context, reserved)
RtToken context, reserved;

Suspend the currently active context (if any) and make *context* become the active context. *context* is a context handle returned by a previous call to **RiBegin**. *reserved* must be RI_NULL; this argument is reserved for expansion of capabilities in a future version of the interface. Returns the handle of the suspended context. **RiContext** can be called at any time to switch contexts, since it is completely independent of the graphics state of any currently active context. If there is no active context, only the extracontextual RenderMan Interface procedure calls may be made.

EXAMPLE

```
RtToken tmp;
tmp = RiContext(c1, RI_NULL);
(void) RiContext(tmp, RI_NULL);
```

SEE ALSO

RiBegin

RtToken
RiCreateHandle(handle, type)
char *handle;
RtToken type;

Predeclare a handle of a resource or data structure. Informs the RenderMan Interface that the user intends to define a data structure or an external resource identified by *handle*, and wishes its scope to be the current attribute block, rather than the interior attribute block within which it will be later defined.

Six Quick RenderMan Interface routines create handles: **RiAreaLightSource**, **RiCoordinateSystem**, **RiLightSource**, **RiMacroBegin**, **RiObjectBegin**, and **RiResource**. **RiAreaLightSource** and **RiLightSource** create identical types of handles, and **RiResource** makes four different types of handles. *type* identifies the type of handle which is being created by **RiCreateHandle**, and may be any one of the following:

`"archive"`	RIB archive resource
`"coordinatesystem"`	application-defined coordinate system
`"image"`	hardware frame buffer or image data file resource
`"lightsource"`	light source or area light source
`"macro"`	RenderMan command macro
`"object"`	composite object
`"shader"`	programmable shader resource (**`.slo`** file)
`"texture"`	texture map resource

RiCreateHandle returns *handle*, or a tokenized version of *handle* if *handle* has not already been declared, or `RI_NULL` on errors. It is an error to use a data structure handle before it has been defined with *type*.

RiCreateHandle replaces the scoping mechanisms available in version 3.1 of the RenderMan Interface with a more powerful (but not backward-compatible) scoping mechanism.

RIB BINDING

```
CreateHandle handle type
```

EXAMPLE

```
RtToken mylight = RiCreateHandle("mylight", RI_LIGHTSOURCE);
RiAttributeBegin();
RiLightSource(mylight, "ambientlight", RI_NULL);
...
RiAttributeEnd();
RiIlluminate(mylight, RI_TRUE);

CreateHandle "mylight" "lightsource"
```

SEE ALSO

RiAreaLightSource, RiLightSource, RiMacroBegin, RiObjectBegin, RiResource, RiCoordinateSystem

---

RiCurve(type, nvertices, wrap, parameterlist)

```
RtToken type;
RtInt nvertices;
RtToken wrap;
```

type can be either `"linear"` or `"cubic"`. *nvertices* is the number of control points in a piecewise-type curve. *parameterlist* is a list of token-array pairs where each token is one of the standard geometric primitive variables applicable to lines, or a variable which has been defined with **RiDeclare**. The parameter list must include at least position (`"P"`, `"Pw"`, or `"Pz"`) information. `"cubic"` curves use the *ustep* step and *ubasis* basis matrix from **RiBasis**.

If a curve wraps (*wrap* is `"periodic"`), it closes upon itself at the ends and the first control points will be automatically repeated. As many as three control points may be repeated, depending on the u-basis step of the curve. Linear curves have a step of 1.

The actual number of curve segments produced by this request depends on the type of the curve and the wrap mode specified. For linear curves, the number of segments (*nsegments*) is:

`nvertices`	if wrap = `"periodic"`
`nvertices-1`	if wrap = `"nonperiodic"`

while for cubic curves, *nsegments* is given by:

`nvertices/ustep`	if wrap = `"periodic"`
`((nvertices-4)/ustep)+1`	if wrap = `"nonperiodic"`

If a variable other than position is varying, it contains *nsegments*`+1` values if *wrap* is `"nonperiodic"` or *nsegments* if *wrap* is `"periodic"`, one for each segment endpoint. If a variable is uniform, it contains *nsegments* elements.

Height fields can be specified with the `"Pz"` parameter by giving just a *z* coordinate of each vertex. The *x* coordinates are set equal to the parametric surface parameter *u*, running from **0** at the first vertex to **1** at the last. The *y* coordinates are set to **0**. Height fields cannot be wrapped.

RIB BINDING

```
Curve type wrap parameterlist
```

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

EXAMPLE

```
RtPoint verts[] = { 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.25, 0.25, 4.0 };
RiCurve(RI_LINEAR, 3, RI_NONPERIODIC,
        "P", (RtPointer)verts, RI_NULL);

Curve "linear" "nonperiodic"
      "P" [0.0 0.0 0.0 0.0 1.0 0.0 0.25 0.25 4.0]
```

SEE ALSO

RiLine, RiNuCurve, RiPatchMesh

```
RtToken
RiDeclare(name, declaration)
   char *name;
   char *declaration;
```

Declare the name and type of a parameterlist variable. The declaration indicates the size and

semantics of values associated with the variable. This information is used by the renderer in processing the variable argument list semantics of the RenderMan Interface. Returns `RI_NULL` on errors.

The syntax of *name* is:

```
[[namespace:]table:]var
```

If *name* begins with an optional table name, the declaration of *var* is installed into the specified table; otherwise it is installed in the global table. There are twelve table namespaces, corresponding to the six types of programmable shaders and the six types of implementation-definable interface extensions: `light, surface, volume, imager, displacement, transformation, attribute, option, geometricapproximation, geometry, hider,` and `resource`. The use of the namespace is optional only if it is not required to identify the table uniquely.

The syntax of declaration is:

```
[class] type ['['n']']
```

where *class* is either `uniform` (the default) or `varying` (required only for primitive variables) and type is either `float, point, color, integer,` or `string` (corresponding to parameterlist values of arrays of type **`RtFloat, RtPoint, RtColor, RtInt`** and **`RtString`**, respectively). The optional bracket notation indicates an array of *n* items of *type*, where *n* is a positive integer. If no array is specified, one item is assumed.

A new declaration replaces any existing declaration of *name* in the specified table. A declaration is automatically removed at the end of the current attribute block, unless *name* has been predeclared as described in **`RiCreateHandle`**. The declaration table associated with a shader and all of its declarations are removed when the shader itself is deallocated.

Quick RenderMan **`RiDeclare`** is backward-compatible with RenderMan Version 3.1.

RIB BINDING

```
Declare name declaration
```

Additionally, *type* may be `vertex`, which indicates position data, such as bicubic patch control points. This is provided for renderer-internal data types, and user-specified data visible in the shading language cannot have this type.

EXAMPLE

```
RiDeclare("temperature", "varying float");
RiDeclare("surface:weird:p", "uniform point");

Declare "option:shadow:bias" "float[2]"
```

SEE ALSO

`RiCreateHandle, RiResource`

IMPLEMENTATION STATUS

Programmable shaders are not implemented in Quick RenderMan.

RiDisplay

Quick RenderMan extends **RiDisplay** as follows: The *name* argument may be either a literal name (per Version 3.1 of the RenderMan Interface specification) or a resource handle of type "image", obtained from **RiResource**. Under Quick RenderMan, platform-specific window-ID information can be specified to **RiResource** and the resultant resource passed to **RiDisplay**.

If *name* is not a resource, Quick RenderMan performs an implicit **RiResource** command to create a resource by that name. The created resource will go out of scope at the end of the current attribute block.

EXAMPLE

```
RtToken res;
RtInt origin[2], windownum;

{ set up origin and windownum }
res = RiResource("mywindow", RI_IMAGE,
                 RI_WINDOWID, &windownum, RI_NULL);
RiDisplay(res, RI_FRAMEBUFFER, RI_RGBAZ,
          RI_ORIGIN, (RtPointer)origin, RI_NULL);
```

SEE ALSO

RiResource

IMPLEMENTATION STATUS

RiDisplay ignores the *mode* parameter and always outputs an "rgba" image with *alpha* = 1.0.

RiErrorHandler(handler)

```
RtFunc handler;
```

Associate with the current context an error handling procedure to be invoked by the renderer when an error is detected. Error handling procedures have the following form:

```
RtVoid handler(type, severity, message, routine, context)
      RtInt type, severity;
      char *message;
      RtToken routine, context;
```

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

context identifies the individual context within which the error occurred. *routine* identifies the RenderMan function in which the error occurred.

The following standard error handlers are supplied:

RiErrorAbort An error of severity `RIE_WARNING` or greater will cause a diagnostic message to be generated and the rendering system will immediately abort the rendering of the current image. On less serious errors, the rendering system will generate the diagnostic message but attempt to continue rendering. When an image is aborted due to an error, the current world block is immediately terminated and no further pixels are generated. The graphics state stack is left as though **RiWorldEnd** had been executed. If the error occurs outside of a world block, there is no change to the graphics state stack. Note the similarity between this behavior and that of the `"abort"` function of **RiSynchronize**.

RiErrorDefault Identical to **RiErrorAbort**, except that any error of severity `RIE_SEVERE` or greater causes the rendering system to abort rendering of the current image.

RiErrorIgnore All errors are silently ignored, and the rendering system attempts to continue rendering.

RiErrorPrint A diagnostic message is generated and written to *stderr* for each warning and error. The rendering system ignores the erroneous information and attempts to continue rendering.

RIB BINDING

```
ErrorHandler "abort"
ErrorHandler "default"
ErrorHandler "ignore"
ErrorHandler "print"
```

SEE ALSO

RiSynchronize

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. However, the size of these surface elements also affects the speed of rendering, since it controls the number of surface elements into which an object is subdivided. Renderer implementations may be able to increase rendering performance by using coarse approximations, or by representing objects with collections of lower-dimensional geometric primitives, such as lines

or points.

RiGeometricApproximation(type, parameterlist)
RtToken type;

Specify a geometric approximation criterion to be applied to future primitives. The only predefined geometric approximation types are "deviation" and "tesselation". "deviation" accepts the **RtFloat** parameter "raster", specifying the maximum permissible distance from the approximated surface to the true surface in raster-space pixels. "tesselation" accepts the parameter "parametric", an array of two **RtFloat**s specifying the minimum number of approximating surface elements per unit in each of the two parametric dimensions.

RIB BINDING

GeometricApproximation type parameterlist

For backward compatibility, RenderMan 3.1 RIB syntax is also accepted:

GeometricApproximation "flatness" value

EXAMPLE

```
RtFloat v = 2.5;
RiGeometricApproximation(RI_DEVIATION, RI_RASTER, &v, RI_NULL);

GeometricApproximation "deviation" "raster" [2.5]
GeometricApproximation "tesselation" "parametric" [8 8]
```

GRAPHIC EXAMPLE

The effects of different parametric tesselation densities on the **RiSphere** command rendered in wire-frame:

844725_paste.tiff ¬

Tesselation	[4 4]	[8 8]	[16 16]

SEE ALSO

RiShadingRate, RiGeometricRepresentation

IMPLEMENTATION STATUS

The "tesselation" type is fully implemented. When called with the "deviation" type, **RiGeometricApproximation** ignores its parameter list and uses the default tesselation instead.

RiGeometricRepresentation(type)
RtToken type;

Select the current geometric representation for geometric primitives. This function specifies the type of rendering primitives that will be used to represent the subsequent geometric

primitives. *type* is one of the following:

`"points"`	representation of an object as a set of colored, drawn dots
`"lines"`	representation of an object as a set of colored, drawn line segments
`"primitive"`	representation of an object as the unapproximated true shaded primitive

Certain renderers have algorithmic limitations on which representations can be rendered, or on which representations can be renderered quickly. Such a renderer will always represent geometric primitives with surface elements that it can render as quickly as possible, and no less complex than *type*, if possible (e.g., a vector-drawing renderer will always draw spheres as lines, but possibly as high quality circular curves if `"primitive"` is selected, whereas a ray tracer may decide it is always faster to render a sphere exactly, even if the `"lines"` representation is selected). In no case will the renderer be expected to represent geometric primitives as objects more complex than the primitives themselves. The default value for *type* is `"primitive"`.

RiGeometricRepresentation does not affect rendering and display options.

RIB BINDING

GeometricRepresentation `type`

EXAMPLE

```
RiGeometricRepresentation("lines");
```

SEE ALSO

RiShadingRate, RiGeometricApproximation

RiGeometry

IMPLEMENTATION STATUS

Quick RenderMan supports the `"teapot"` primitive under the **RiGeometry** command.

192935_paste.tiff ¬

RiHider

Quick RenderMan recognizes the following values for *type*:

`"hidden"`	Performs complete, accurate hidden surface elimination. Every surface obscures every other farther-away surface.
`"sketch"`	Identical to `"hidden"`, with the following exception: Before processing a tile, its vertices are tested against the other surfaces already in the image. If all vertices are obscured by nearer surfaces, the tile is

discarded and none of it appears in the image; otherwise, the "hidden" algorithm is performed for every pixel of the tile. "sketch" can execute faster than "hidden" but can in some instances fail to display tiles that should appear.

"paint" Geometric primitives are written to the image in the order in which they are defined, with no attempt to suppress the appearance of hidden surfaces. New surfaces obscure overlapping old surfaces.

"null" Performs no pixel computations and produces no output. Very efficient.

"pick" Calls an application-provided subroutine with information identifying geometric primitives being rendered and their respective minimum *z* values. This hider produces no graphic output.

RiIlluminate(lightsource, on)
RtToken lightsource;
RtBoolean on;

If *on* is RI_TRUE and the light source referred to by *lightsource* is not in the *current light source list*, add it to the list. If *on* is RI_FALSE and the light source referred to by *lightsource* is in the *current light source list*, remove it from the list. *lightsource* is the name of a light source previously defined with **RiLightSource** or **RiAreaLightSource**. Note that popping the attributes restores the on values of all lights to their previous values.

RIB BINDING

Illuminate lightsource on

For backward RIB compatibility, *lightsource* may be an integer instead of a token.

EXAMPLE

```
RtToken myamb;
RtFloat intensity = 0.2;
myamb = RiLightSource("myamb", "ambientlight",
        "intensity", &intensity, RI_NULL);
RiIlluminate(myamb, RI_FALSE);

LightSource "myamb" "ambientlight" "intensity" [0.2]
Illuminate "myamb" 0
```

SEE ALSO

RiLightSource, RiAreaLightSource

RtToken
RiLightSource(handle, shader, parameterlist)
RtToken handle;
RtToken shader;

RiLightSource defines a non-area light source of the given name and type, and adds it to the *current light source list*, thus turning it on. *shader* is the name of a predefined light shader, or a user-defined light shader previously declared with **RiResource**.

Light sources defined with **RiLightSource** are attributes, and are stackable with **RiAttributeBegin**/**RiAttributeEnd**.

The light source is implicitly instantiated (turned on) when it is defined, and implicitly disinstantiated (turned off) at the end of the attribute block in which it is defined; in addition, it can be explicitly disinstantiated or reinstantiated with **RiIlluminate**.

RIB BINDING

```
LightSource handle shader parameterlist
```

For backward compatibility, RenderMan 3.1 RIB syntax is also accepted:

```
LightSource shader sequencenumber parameterlist
```

EXAMPLE

```
RtFloat angle = 40.0;
RtToken mylight;
mylight = RiLightSource("mylight", "spotlight", "coneangle",
                        (RtPointer)&angle, RI_NULL);

LightSource "redAmbientLight" "ambientlight" "lightcolor"
            [.5 0 0] "intensity" [.6]
```

SEE ALSO

RiIlluminate, RiResource, RiCreateHandle, RiAreaLightSource

IMPLEMENTATION STATUS

Only the standard light shaders are supported. Quick RenderMan presently supports no programmable light shaders.

```
RiLine(nvertices, parameterlist)
   RtInt nvertices;
```

nvertices is the number of vertices in a connected piecewise-linear curve. If *nvertices* is one, a single point is drawn. *parameterlist* is a list of token-array pairs where each token is one of the standard geometric primitive variables applicable to lines, or a variable which has been defined with **RiDeclare**. The parameter list must include at least position ("P" or "Pw") information. If a primitive variable is varying, the array contains *nvertices* elements of the type corresponding to the token. If the variable is uniform, the array contains a single element. The number of floats associated with each type is given in the table, *Standard Geometric Primitive Variables*.

RIB BINDING

```
Line parameterlist
```

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

EXAMPLE

```
RtPoint points[3] = {
    0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.25, 0.25, 4.0
};
RiLine(3, RI_P, (RtPointer)points, RI_NULL);

Line "P" [0 0 0 0 1 0 0.25 0.25 4] "Cs" [1 1 1 1 0 0 0 0 1]
```

SEE ALSO

RiCurve, RiPolygon

Macros

Quick RenderMan provides a mechanism for defining and instantiating hierarchical named macros, whose use is encouraged both to clarify the structure of complex scenes for the benefit of other RIB-processing programs and people, to speed up the RenderMan Interface and compact the RIB representation of repetitive scenes, and to provide caching hints to the renderer. Any sequence of RenderMan Interface function calls, with few exceptions, may be retained as a named macro by enclosing it within an **RiMacroBegin/RiMacroEnd** block. A macro so defined can then be instantiated with **RiMacroInstance** or redefined with a subsequent **RiMacroBegin/End** block. Hierarchical macros can be defined by instantiating submacros within macro definitions. Except for data-type information, the current graphics state has no effect on a macro definition, nor does the macro definition have any effect on the current graphics state or cause any rendering to occur; in contrast, macro instances outside of all macro definition blocks do inherit and alter the current graphics state, and do (if they contain geometry) cause rendering to occur. Specifically, the dimensionality of any colors and the sizes of any patch-mesh parameter arrays specified in the macro definition are determined by the current color dimensionality and spline basis step sizes, and the class, type, dimension, and size of any user-defined parameters are determined by the current declarations of those parameters.

In some implementations, macros may be interpreted by the server rather than by the client library. Thus, metacontextual function calls (**RiBegin**, **RiEnd**, **RiContext**, **RiSynchronize**, and **RiErrorHandler**) may not appear within a macro definition. Furthermore, when a macro is instantiated, its contents are executed without actually being called by the application, so if the macro included a call to a function returning a value, the return value would have no recipient. Thus, function calls that return irrecoverable values (for example, **RiTransformPoints**) may not appear in macro definitions. With these two exceptions, any RenderMan Interface commands can be enclosed in a macro definition. In particular, the definition need not contain any geometry, and side effects, though discouraged, are allowed. However, since macros provide natural caching handles, restricting the contents of macro definitions can greatly facilitate caching of more than just the sequence of RenderMan Interface commands for some renderers.

Note that, when changing color spaces with **RiColorSamples**, the current color dimensionality during the macro definition must agree with the current color dimensionality during its instantiations if the macro contains any color specifications; similarly, when changing spline bases (with **RiBasis**), the current spline step sizes during the macro definition must agree with the current spline step sizes during its instantiations if the macro contains any patch meshes or curve chains. And in general, when using user-defined parameters, the class, type, and dimension of such parameters used during a macro definition must agree with the declarations current during macro instantiation.

```
MacroBegin "cube"
  TransformBegin
  Scale .8 .8 .8
  Polygon "P" [-0.5 -0.5 0.5 -0.5 -0.5 -0.5 0.5 -0.5 -0.5 0.5 -0.5 0.5]
  Polygon "P" [-0.5 -0.5 0.5 -0.5 0.5 0.5 -0.5 0.5 -0.5 -0.5 -0.5 -0.5]
  Polygon "P" [0.5 0.5 -0.5 -0.5 0.5 -0.5 -0.5 0.5 0.5 0.5 0.5 0.5]
  Polygon "P" [0.5 0.5 -0.5 0.5 0.5 0.5 0.5 -0.5 0.5 0.5 -0.5 -0.5]
  Polygon "P" [0.5 -0.5 0.5 0.5 0.5 0.5 -0.5 0.5 0.5 -0.5 -0.5 0.5]
  Polygon "P" [-0.5 0.5 -0.5 0.5 0.5 -0.5 0.5 -0.5 -0.5 -0.5 -0.5 -0.5]
  TransformEnd
MacroEnd
MacroBegin "row"
  TransformBegin
    Translate 0 0 -1
    MacroInstance "cube"
  TransformEnd
  MacroInstance "cube"
  TransformBegin
    Translate 0 0 1
    MacroInstance "cube"
  TransformEnd
MacroEnd
MacroBegin "face"
  TransformBegin
    Color [1 0 0]
    Translate 0 -1 0
    MacroInstance "row"
  TransformEnd
  Color [1 1 0]
  MacroInstance "row"
  TransformBegin
    Translate 0 1 0
    Color [1 0 1]
    MacroInstance "row"
  TransformEnd
MacroEnd
MacroBegin "lotsofcubes"
  TransformBegin
    Translate -1 0 0
    MacroInstance "face"
  TransformEnd
  MacroInstance "face"
  TransformBegin
    Translate 1 0 0
    Rotate 20 1 0 0
    MacroInstance "face"
```

```
  TransformEnd
MacroEnd

WorldBegin
MacroInstance "lotsofcubes"
WorldEnd
```

Macro Example: A 3x3 cube constructed out of nested macro instantiations

224974_paste.tiff ¬

RtToken
RiMacroBegin(handle, parameterlist)
   char *handle;

RiMacroEnd()

RiMacroBegin starts the definition of a macro, storing the following sequence of RenderMan Interface calls up to the next **RiMacroEnd** call under the specified name. *macro* is the name given to the macro itself. A macro implicitly persists until the end of the outermost attribute block in which it is defined with **RiMacroBegin** or declared with **RiCreateHandle**(*macro,* "macro"). The macro can be redefined with a subsequent **RiMacroBegin** call as a stackable attribute; if it is predeclared with **RiCreateHandle**, the outermost definition persists until that declaration goes out of scope. A macro is not instantiated when it is defined, but can be subsequently instantiated with **RiMacroInstance**. Presently, no parameters are valid for **RiMacroBegin**, so *parameterlist* must be empty.

RiMacroEnd ends the definition of the current macro.

RIB BINDING
```
MacroBegin macro parameterList

MacroEnd
```

EXAMPLE
```
MacroBegin "unitSphere"
Sphere 1 -1 1 360
MacroEnd
```

SEE ALSO
RiMacroInstance, RiCreateHandle

IMPLEMENTATION STATUS
When writing a RIB file, commands appearing between a **MacroBegin** command and the following **MacroEnd** command are not output to the file. This is a known problem.

Any pairing RenderMan primitives (e.g., **RiAttributeBegin** and **RiAttributeEnd**)

appearing within a macro must be balanced. For example, this is legal:

```
myhandle = RiMacroBegin("mymacro", RI_NULL);
RiAttributeBegin();
...
RiAttributeEnd();
RiMacroEnd();
```

while this is not:

```
myhandle = RiMacroBegin("mymacro", RI_NULL);
RiAttributeEnd();  /* not balanced with RiAttributeBegin */
RiMacroEnd();
```

RiMacroInstance(macro, parameterlist)
RtToken macro;

Create an instance of the specified, previously defined macro. The macro inherits the current graphics state at the time of instantiation. *macro* is the **RtToken** returned by **RiMacroBegin**. Presently, no parameters are valid for **RiMacroInstance**, so *parameterlist* must be empty.

RIB BINDING
MacroInstance macro parameterlist

EXAMPLE
MacroInstance "unitSphere"

SEE ALSO
RiMacroBegin, RiMacroEnd

Motion

Quick RenderMan does not support the *Motion Blur* capability as defined in the RenderMan 3.1 specification. Thus, only the first element in the list of **RiMotionBegin** is evaluated, as prescribed by the specification.

RiNuCurve(nvertices, order, knot, min, max, parameterlist)
RtInt nvertices, order;
RtFloat knot[], min, max;

Create a rational or polynomial non-uniform B-spline curve. *parameterlist* is a list of token-array pairs where each token is one of the standard geometric primitive variables applicable to lines, or a variable that has been defined with **RiDeclare**. The parameter list must

include at least position ("P" or "Pw") information. The curve specified is rational if the positions of the control points are 4-vectors (*x,y,z,w*), and polynomial if the positions are 3-vectors (*x,y,z*).

nvertices is the number of control points in the curve. The *order* must be positive and is equal to the degree of the polynomial basis plus 1. 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 vector associated with each control point must also be specified. Each value in this array 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 curve is defined in the range *min* to *max*. This is different from other curve primitives where the parameter value is always assumed to lie between **0** and **1**. *min* must be less than *max*, and must also be greater than or equal to `knot[order-1]`. *max* must be less than or equal to `knot[nvertices]`.

RIB BINDING

NuCurve order knot min max parameterlist

The number of vertices in the non-uniform curve is determined implicitly by the order and the number of elements in the knot array, and this number must match the number of elements in the required position array.

EXAMPLE

```
RtFloat knots[] = { 0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 4 };
RtPoint verts[] = {
    1, .5, 0, 1, 1, 1, 0, 1, 1, 2, 0, 2, 0, 1, 0, 1, 0, .5, 0, 1,
    0, 0, 0, 1, 1, 0, 0, 2, 1, 0, 0, 1, 1, .5, 0, 1
};
RiNuCurve(9, 3, knots, 0, 4, RI_PW, (RtPointer)verts, RI_NULL);

NuCurve 3 [0 0 0 1 1 2 2 3 3 4 4 4] 0 4
        "Pw" [1 .5 0 1 1 1 0 1 1 2 0 2 0 1 0 1 0 .5 0 1
              0 0 0 1 1 0 0 2 1 0 0 1 1 .5 0 1]
```

SEE ALSO

RiCurve, RiNuPatch, RiTrimCurve

Retained Geometry

A single geometric primitive or a hierarchy of geometric primitives may be retained by enclosing them in an **RiObjectBegin-ObjectEnd** object block. The RenderMan Interface identifies each object using the user-supplied **RtToken** as a handle to this retained data structure. This handle can subsequently be used to reference the object when creating instances with **RiObjectInstance**.

An object may contain an arbitrary sequence of geometric primitives, geometric attributes and transformations, including attribute and transformation stack manipulations. The single restriction is that the attribute and transformation stacks must properly balanced and closed,

that is, end at the same stacking level as they started. Functions that are invalid inside the **RiWorldBegin-WorldEnd** world block, such as options and texture making subroutines, are invalid inside the object block. Object block creation may be nested. A more general method of retaining and reusing RenderMan Interface procedure calls is provided by **RiMacroBegin**.

An instanced object will inherit the graphics state that exists at the time it is instanced, not at the time at which is created. Any attributes set in the object will modify the graphics state for subsequent primitives inside, but not after the object instanced, (that is, objects do not have side effects). **RiObjectBegin** can be called outside of the world block, and this is the only time that geometric primitives can be called outside of a world block.

```
RtToken
RiObjectBegin(handle)
   RtToken handle;

RiObjectEnd()
```

RiObjectBegin starts the definition of an object. Returns *handle*, or a tokenized version of *handle* if *handle* has not already been declared, or `RI_NULL` if the object could not be created for any reason.

Objects are not rendered when they are defined within the **RiObjectBegin-RiObjectEnd** block; only an internal definition is created. After definition, an object may then be instanced any number of times until it is deleted. Objects are automatically deleted at the end of the current attribute block, unless *handle* has been predeclaredwith **RiCreateHandle**.

Note that **RiObjectBegin** in Quick RenderMan returns an **RtToken**, whereas in RenderMan Version 3.1 the return type is **RtObjectHandle**.

RIB BINDING

```
ObjectBegin handle
ObjectEnd
```

EXAMPLE

```
RtToken sph;
sph = ObjectBegin("myobject");
RiSphere(1.0, -1.0, 1.0, 360.0, RI_NULL);
ObjectEnd();

ObjectBegin "myobject"
Color 1 1 0
Sphere 1 -1 1 360
ObjectEnd
```

SEE ALSO

RiObjectInstance, RiCreateHandle, RiMacroBegin

IMPLEMENTATION STATUS

Quick RenderMan presently implements the **RiObject** commands by calling the corresponding **RiMacro** commands internally. This implies that the "macro" handle type (see **RiCreateHandle**) applies instead of the "object" handle type, and that objects are subject to all the restrictions and freedoms of macros.

When writing a RIB file, commands appearing between an **ObjectBegin** command and the following **ObjectEnd** command are not output to the file.

RiObjectInstance(handle)
RtToken handle;

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

RIB BINDING

ObjectInstance handle

SEE ALSO

RiObjectBegin

IMPLEMENTATION STATUS

When writing a RIB file, Quick RenderMan writes **MacroInstance** commands in lieu of **ObjectInstance** commands.

RiPointsLines(nlines, nvertices, vertices, parameterlist)
RtInt nlines, nvertices[], vertices[];

Define *nlines* lines that share vertices. The array *nvertices* contains the number of vertices in each line and has length *nlines*. The array *vertices* contains, for each line vertex, an index into the varying primitive variable arrays. The varying arrays are 0-based. *vertices* has a length equal to the sum of all of the values in the *nvertices* array. Individual vertices in *parameterlist* are thus accessed indirectly through the indices in the array *vertices*. *parameterlist* is a list of token-value pairs where each token is one of the standard geometric variables applicable to lines, or a variable which has been declared with **RiDeclare**. The parameter list must include at least position ("P" or "Pw") information. If a primitive variable is varying, the array contains *n* elements of the type corresponding to the token, where *n* is equal to the maximum value in the array *nvertices* plus one. If the variable is uniform, the array contains *nlines* elements of the associated type. The number of floats associated with each type is given in the table, *Standard Geometric Primitive Variables*.

RIB BINDING

PointsLines nvertices vertices parameterlist

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

EXAMPLE

```
RtInt nvertices[5] = { 4, 3, 3, 3, 3 };
RtInt vertices[16] = {
      1, 2, 3, 4, 0, 1, 2, 0, 2, 3, 0, 3, 4, 0, 4, 1
};
RtPoint verts[5] = {
      0, 0, 0, -1, -1, 1, -1, 1, 1, 1, -1, 1, 1, 1, 1
};
RiPointsLines(5, nvertices, vertices,
              RI_P, (RtPointer)verts, RI_NULL);

PointsLines [4 3 3 3 3] [1 2 3 4 0 1 2 0 2 3 0 3 4 0 4 1]
            "P" [0 0 0 -1 -1 1 -1 1 1 1 -1 1 1 1 1]
            "Cs" [1 1 1 0 0 0 1 0 0 0 0 1 1 0 1]
```

SEE ALSO

RiPointsPolygons

RiReadArchive(resource, callback, parameterlist)

```
RtToken resource;
RtFunc callback;
```

Interpolate the contents of a RIB archive resource. Quick RenderMan will parse and execute each command in the archive as if it had been called by the application program directly. *resource* is the name of a resource (as returned by **RiResource**) or the host-specific pathname of a file containing RIB. *callback* is a user-supplied C function with the following definition:

callback(type, format, data)
RtToken type;
char *format, *data;

Quick RenderMan calls this function each time it encounters a user data record (beginning with '#') within the archive. If *callback* is zero, Quick RenderMan will not call back to the application. *type* will be either "comment" or "structure" as described in **RiArchiveRecord**. *data* is a NUL-terminated ASCII string that contains the text of the user data record; the application should not modify this data. *format* is always the literal string "%s". Notice that the calling sequence for *callback* is compatible with **RiArchiveRecord**. Under RenderMan Version 3.1, any user data records encountered within a RIB archive were automatically fed to **RiArchiveRecord** by the RIB parser. This is no longer the case in Quick RenderMan, but the application is free to call **RiArchiveRecord** itself.

The only legal parameter is the **RtInt** value "defer", which governs the action taken when Quick RenderMan is writing a RIB archive. If "defer" is specified as a non-zero value, Quick RenderMan writes a **ReadArchive** command into the output RIB stream and does not attempt to read the contents of the archive; otherwise, Quick RenderMan opens the RIB

archive resource and processes its contents, writing them into the RIB output stream. `"defer"` has no effect if Quick RenderMan is not writing a RIB archive.

It is legal for a RIB resource to contain a **`ReadArchive`** command itself; however, Quick RenderMan makes no effort to detect circular inclusion.

RIB BINDING

`ReadArchive` `resource parameterlist`

EXAMPLE

Including a file resource containing RIB commands:

```
char *myrib = "test.rib";
RtToken myname;
myname = RiResource("myres", RI_ARCHIVE,
                    RI_FILEPATH, &myrib, RI_NULL);
RiReadArchive(myname, mycallback, RI_NULL);
```

Including a memory resource (NUL-terminated ASCII text) containing RIB commands:

```
char *myrib = "Cylinder .5 .2 1 360";
RtToken myname;
myname = RiResource("myres", RI_ARCHIVE,
                    RI_ADDRESS, &myrib, RI_NULL);
RiReadArchive(myname, mycallback, RI_NULL);
```

RIB example:

```
ReadArchive "myrib.rib"
```

SEE ALSO

`RiResource, RiBegin, RiArchiveRecord`

External Resources

Certain RenderMan entities, such as images and RIB files, may be defined externally to RenderMan, and are known as *resources*. Resources can be either memory-resident or file-resident. Resources are attributes, and can thus be redeclared with subsequent calls to **`RiResource`** and stacked with **`RiAttributeBegin`**/**`RiAttributeEnd`**.

```
RtToken
RiResource(handle, type, parameterlist)
   char *handle;
   RtToken type;
```

Images, shaders, maps, and RIB entities, known collectively as *resources*, are declared as to type. Images, declared as the type `"image"`, can be both input and output. The current

output image of the renderer is set by **RiDisplay**, and defined by everything until **RiWorldEnd**, where the output is an image. ASCII RIB data for input to Quick RenderMan can be declared as type "archive". *type* may be any one of the following:

"archive"	RIB archive
"image"	hardware frame buffer or image data file
"shader"	programmable shader (**.slo** file)
"texture"	texture map

Note that each resource type has a corresponding handle type (see **RiCreateHandle**).

For memory-resident resources, the parameter "address" is an **RtPointer** specifying the address of an array in the application's address space; memory-resident framebuffer resources also recognize the parameter "windowid", an **RtInt** specifying an open window in a window-system-specific fashion. For file-resident resources, the parameter "filepointer" is a *stdio* FILE€* pointing to an open file in an operating-system-specific fashion, while "filepath" specifies the name of the file in a file-system-specific fashion.

If the resource was created successfully, **RiResource** returns an **RtToken**, which can be passed to other Quick RenderMan commands that accept a resource identifier, for example, **RiSurface** and **RiDisplay**. **RiResource** returns RI_NULL on failure.

RIB BINDING

Resource handle type parameterlist

EXAMPLE

This example demonstrates declaring and using a memory resource containing RIB commands.

```
char *myrib = "Cylinder .5 .2 1 360";
RtToken myname;
myname = RiResource("myres", RI_ARCHIVE,
                    RI_ADDRESS, &myrib, RI_NULL);
RiReadArchive(myname, mycallback, RI_NULL);
```

SEE ALSO

RiDisplay, RiReadArchive

RiSurface

IMPLEMENTATION STATUS

Quick RenderMan presently implements only the "constant", "matte", "metal", "shinymetal", "paintedplastic", and "plastic" standard surface shaders. Since Quick RenderMan does not yet support texture maps, "shinymetal" is equivalent to "metal", and "paintedplastic" is equivalent to "plastic". In addition, Quick RenderMan provides two implementation-specific surface shaders, "defaultsurface" and "show_nxnynz". Programmable surface shaders are not implemented.

Synchronization

RenderMan implementations may employ remote rendering servers or renderers that are accessed through interprocess communication on computers that support multiprocessing. RenderMan provides a mechanism for coordinating the synchronization of renderers that run outside of the execution thread of the user application. The *current synchronization mode* determines how soon the RenderMan implementation returns control to the application when asked to transmit a RenderMan Interface subroutine call to such an external renderer.

RiSynchronize(mode)
RtToken mode;

Synchronize the renderer with the application program according to *mode*, which is one of:

"synchronous"	Set the current synchronization mode of the current context to wait for the processing required by each RenderMan Interface call to complete before returning from that call.
"unbuffered"	Set the current synchronization mode of the current context to transmit each RenderMan Interface call to the renderer without delay, and return after the call is transmitted but possibly before the processing required by the call is completed.
"asynchronous"	Set the current synchronization mode of the current context to buffer each RenderMan Interface call and return from the call immediately, transmit the buffer to the renderer as is appropriate and efficient.
"flush"	Transmit all buffered RenderMan Interface calls to the renderer and return from **RiSynchronize** immediately, but do not otherwise modify the synchronization mode.
"wait"	Transmit all buffered RenderMan Interface calls to the renderer, and return from **RiSynchronize** only when all processing required from these calls is complete, but do not otherwise modify the synchronization mode.
"abort"	Abort rendering of the current image, but do not otherwise modify the synchronization mode. The current world block (and any interior blocks) is immediately terminated and no further graphic output is generated. The graphics state stack is left as though **RiWorldEnd** had been executed. If called outside a world block, there is no effect upon the graphics state.

Note that the first three modes described above select a mode of operation that endures until the next call to **RiSynchronize** or until the context terminates. The last three modes specify a one-time action that does not affect the synchronization of subsequent RenderMan commands.

RIB BINDING

```
Synchronize mode
```

EXAMPLE

```
RiSynchronize("asynchronous");
RiWorldBegin();
...
RiWorldEnd();
RiSynchronize("wait");

Synchronize "flush"
```

RiTransformPoints

IMPLEMENTATION STATUS

Only the predefined spaces may be specified as *tospace* and *fromspace*. The predefined spaces are `"camera"`, `"world"`, `"object"`, and `"raster"`.

Quick RenderMan Special Features

The following Quick RenderMan features are not part of the Interface Specfication.

Clip Object RIB Archives

The **RiReadArchive** command can be instructed to regard a RIB file as a *clip object*, and process a particular subset of the commands in the file. When a RIB file is read in clip object mode, Quick RenderMan processes only the **Declare** commands and all commands appearing between (but not including) the first **WorldBegin** and **WorldEnd** commands. To select clip object mode, the application must set the value of a special "clipobject" option variable to 1. Subsequent **RiReadArchive** commands will be executed in clip object mode. For example:

```
static RtInt clipon = 1, clipoff = 0;

/* select clip object mode and read a RIB file */
RiOption(RI_ARCHIVE, "clipobject", &clipon, RI_NULL);
RiReadArchive("myclipobject", mycallback, RI_NULL);
/* return to normal (not clip object) mode */
RiOption(RI_ARCHIVE, "clipobject", &clipoff, RI_NULL);
```

Writing Version 3.1-Compatible RIB Archives

When an application creates a context to write a RIB file, Quick RenderMan will normally write RIB conforming to this specification. An application desiring to write RIB that is compatible with RenderMan Version 3.1 can specify this immediately after creating the context, as shown in the example below.

```
static RtInt v31 = 301;
char *outputfile = "myfilename", *archive = RI_ARCHIVE;

RiBegin(RI_NULL, RI_RENDERER, &archive,
        RI_FILEPATH, &outputfile, RI_NULL);
RiOption(RI_ARCHIVE, "outputversion", &v31, RI_NULL);
```

Expanding Macros During RIB Output

When an application writes to a RIB file, Quick RenderMan will normally pass all **RiMacroBegin**, **RiMacroEnd**, and **RiMacroInstance** commands, as well as the macro definition itself directly into the RIB archive. An application can alternatively request that all **RiMacroInstance** commands be replaced by their respective macro definitions immediately after creating the context, using the code below.

```
static RtInt one = 1;
RiOption(RI_ARCHIVE, "expandmacros", &one, RI_NULL);
```

Expanding Macros During RIB Output

When an application writes to a RIB file, Quick RenderMan will normally pass all **RiMacroBegin**, **RiMacroEnd**, and **RiMacroInstance** commands, as well as the macro definition itself directly into the RIB archive. An application can alternatively request that all **RiMacroInstance** commands be replaced by their respective macro definitions immediately after creating the context, using the code below.

```
static RtInt one = 1;
RiOption(RI_ARCHIVE, "expandmacros", &one, RI_NULL);
```

Picking

Quick RenderMan provides a "pick" hider as a graphic option. This hider produces no graphic output, but instead calls an application-provided routine with information identifying geometric primitives being rendered and their respective minimum *z* values. As each primitive is rendered, it inherits the *pickable* and *pick tag* attributes from the graphics state. Both of these attributes are set with **RiAttribute**, using the "picking" name The pickable attribute is a boolean value expressed as an **RtInt**; if it is non-zero, the application program will be called when the

primitive is rendered. The pickable attribute is initialized to **1** (true) when the context is created. The application calls **QRMSetPickCallback()** to associate a *callback routine* with a Quick RenderMan context:

```
RtVoid QRMSetPickCallback(RtVoid (*func)(RtInt, RtInt *, RtFloat));
```

When QuickRenderMan renders a pickable geometric primitive, it calls the callback routine. A pick tag is a list of **RtInt** values. The application can set the last element of the list with the **RiAttribute** command. The list grows by one value the first time the application sets a pick tag within an attribute block. Subsequent **RiAttribute** calls within an attribute block replace the last value in the list. The first argument to the callback routine is the length of the tag list. The second argument is a pointer to the zero*th* element of the tag list. The third argument is the minimum *z* value of the primitive.

An example of using the pick hider follows:

```
RtVoid mycallback(RtInt tagCount, RtInt *tagList, RtFloat z)
{
    ...
}

static RtInt tag1 = 1, tag2 = 2, tag3 = 3, tag4 = 4;
static RtInt isPickable = 1, notPickable = 0;

QRMSetPickCallback(mycallback);
RiHider(RI_PICK, RI_NULL);
... set up camera and other options ...
RiWorldBegin();
RiAttribute(RI_PICKING, RI_PICKTAG, &tag1, RI_NULL);
/* the tag list now contains { 1 } */
RiAttribute(RI_PICKING, RI_PICKTAG, &tag2, RI_NULL);
/* the tag list now contains { 2 } */
/* the following RiSphere command will cause Quick RenderMan to
 * invoke the application's routine mycallback() */
RiSphere(1.0, -1.0, 1.0, 360.0, RI_NULL);
RiAttributeBegin();
RiAttribute(RI_PICKING, RI_PICKTAG, &tag3, RI_NULL);
/* the tag list now contains { 2, 3 } */
RiAttribute(RI_PICKING, RI_PICKABLE, &notPickable, RI_NULL);
/* the following RiSphere command will not cause a callback
 * because "pickable" is presently False */
RiSphere(1.0, -1.0, 1.0, 360.0, RI_NULL);
RiAttribute(RI_PICKING, RI_PICKTAG, &tag4, RI_NULL);
/* the tag list now contains { 2, 4 } */
RiAttributeEnd();
/* the tag list now contains { 2 } */
/* the "pickable" attribute has returned to True */
RiWorldEnd();
```

RIB Reader Intercept

When Quick RenderMan reads a RIB archive resource, it parses each RIB command and then calls a command-specific *handler* subroutine to process the RIB command. By default, the handler routine for each RIB command is its respective **Ri** subroutine. Quick RenderMan provides two C language entry points that permit the application to substitute its own RIB handlers for the defaults on a per-context basis. **QRMGetRIBHandlers()** returns a structure with the current handlers for the active context. **QRMSetRIBHandlers()** accepts a structure containing handler pointers and associates them with the active context.

```
RtVoid QRMGetRIBHandlers(RtRIBHandlers *handlersPointer);
RtVoid QRMSetRIBHandlers(RtRIBHandlers *handlersPointer);
```

An example of intercepting the **Rotate** command follows. The header file *ribhdlr.h* contains the definition of the **RtRIBHandlers** structure.

```
#include <ri.h>
#include <ribhdlr.h>

RtVoid myRotate(RtFloat angle, RtFloat dx, RtFloat dy, RtFloat dz)
{
        if (angle > 30.0)
                RiRotate(angle + 20.0, dx, dy, dz);
}

main(int argc, char **argv)
{
        RtRIBHandlers handlers;

        .
        RiBegin(...
        .
        QRMGetRIBHandlers(&handlers);  /* get current handlers */
        handlers.Rotate = myRotate;
        /* note that all other handlers remain unchanged */
        QRMSetRIBHandlers(&handlers);
        .
        RiReadArchive(...
}
```

Note that although the application-specified handler is free to call one or more **Ri** routines, is is not obligated to do anything.