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.

Modes

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 type of rendering context is determined by *name*:

Rendering If *name* is the name of a renderer, to select among various implementations that may be available, or RI_RENDER for the default renderer.

Detached rendering If *name* obeys a special syntax, an external process will be launched and further Ri calls will be serialized in the form of RIB and piped to the standard input of the external process:

```
RiBegin("launch:executable? <options>")
```

In this mode, all Ri calls other than RiEnd are non-blocking. The external process operates asynchronously with respect to the client application. RiEnd signals the external process to exit and all interprocess communication with the external process is shut down. RiEnd does not return until this is complete.

The command line options to the executable can be anything, but three special wildcard options are needed to enable: the Ric API, display services in the client process, and error message processing in the client process. The command needed to launch prman with all of these services is:

```
RiBegin("launch:prman? -ctrl $ctrlin $ctrlout -dspy $dspyin $dspyout -xcpt $xcptin")
```

The Ri Control API provides functions a client application will find useful when interacting with a detached prman process. These functions will not work if the launch string does not contain "-ctrl $ctrlin $ctrlout".

For off-line rendering, display drivers are loaded as dynamic libraries into the renderer's process. When writing an interactive application, it is often desireable to load a user-written display driver into the client application process and configure prman to ship pixels to the driver through a fast, low-latency interprocess communication channel. The "-dspy$dspyin $dspyout" option directs the rendering context to configure the detached prman to send pixels back to the client application. To load a display driver directly, an application should use DspyRegisterDriverTable described in Direct-linked Display Driver Registration

Finally, an application may want error messages processed by the error handler installed in the application's process. The "-xcpt $xcptin" option directs the rendering context to send error messages from the detached process to the client process for delivery.

RIB generation Otherwise, *name* is assumed to be the name of a file to create.

Supplying RI_NULL for *name* 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. 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++).

Note that only RIB contexts may be created by calling RiBegin(), with the filename for that RIB being the name supplied to RiBegin(). An immediate rendering context cannot be created, save for the one already created for you by the renderer. Thus, the primary purpose of the RiContext call in PRMan is to create RIB files from within a procedural DSO. It is important in this case to always call RiGetContext() at the beginning of the DSO, and then restore the context to its previous value using RiContext() at the end if you create or set any contexts within your DSO. The active context upon entry should always be the immediate rendering context unless you violate this recommendation.

```
  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);
```

```
  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).

If baking for re-rendering has been enabled with the options:

```
Option "render" "int rerenderbake" [1]
Option "render" "string rerenderbakedbdir" "dirname"
```

a database for re-rendering will be written to disk by the time RiWorldEnd returns.

RIB BINDING

```
  WorldBegin -
WorldEnd -
```

EXAMPLE

```
      RiWorldBegin ();
      RiGeometry ("teapot", RI_NULL);
    RiWorldEnd ();
```

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

```
  RiSolidBegin ()
RiSolidEnd ()

RiMotionBegin ()
RiMotionEnd ()
```

```
  RiObjectBegin ()
RiObjectEnd ()
```

Define an object prototype or object group definition that can be instantiated with ObjectInstance. This calls resets the graphics state so that no attributes are active. Attribute values existing at the time ObjectInstance is called will be inherited by the instance. Any attribute explicitly defined inside an ObjectBegin/ObjectEnd block will have higher precedence and will override those attributes. This is similar to behavior of ArchiveBegin/ArchiveEnd block overriding the attribute values present when using ReadArchive.

```
  RiAttributeBegin ()
RiAttributeEnd ()

RiTransformBegin ()
RiTransformEnd ()
```

These pairs save and restore the attributes in the graphics state, and save and restore the current transformation, respectively. All begin-end pairs (except RiTransformBegin-RiTransformEnd andRiMotionBegin-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.

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

Declares 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*, or *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*. 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 that 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]"
```

> ***Inline Declarations***
>
> *In addition to using RiDeclare to globally declare the type of a variable, the type and storage class of a variable may be declared* inline *. 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 inline 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 inline declaration. The scope of an inline 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 preceding RiDeclare, it is also legal to use an inline declaration.*

Mode Example

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 */
   RiDisplayChannel(...):
   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();
```

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, and non-linear transformations programmed in the Shading Language. 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 ( );
```

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]
```

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 );
```

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
```

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);
```

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);
```

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

Concatenate a scaling onto the *current transformation*.

RIB BINDING

```
  Scale sx sy sz
```

EXAMPLE

```
      Scale .5 1 1
```

```
  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);
```

Named coordinate systems

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

```
  RiCoordinateSystem ( RtToken space )
```

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 space
```

EXAMPLE

```
      CoordinateSystem "lamptop"
```

```
  RiScopedCoordinateSystem ( RtToken name )
```

Like RiCoordinateSystem, this function marks the coordinate system defined by the current transformation with the indicated *name* and saves it. Unlike that call, the marked transformation is saved on a separate stack, independent of the global list maintained by RiCoordinateSystem. This stack is pushed and popped by RiAttributeBegin andRiAttributeEnd calls (but not by RiTransformBegin and RiTransformEnd). Scoped coordinate system can then be referred to by name in subsequent shaders, or in RiTransformPoints and RiCoordSysTransform, just like global coordinate systems. When searching for a named coordinate system, a renderer should first check the scoped coordinate system stack; failing that, the global coordinate system list should be checked.

RIB BINDING

```
  ScopedCoordinateSystem space
```

EXAMPLE

```
      ScopedCoordinateSystem "lamptop"
```

```
  RiCoordSysTransform ( RtToken name )
```

This function replaces the current transformation matrix with the matrix that forms the *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"
```

```
  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);
```

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 ();
```

Motion

Some rendering programs are capable of performing temporal anti-aliasing and motion blur. Motion blur is specified through *moving transformations* and *m oving 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 -
```

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 ();
```

The table below 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 t0 are used to render each moving object.

Moving Commands

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

Resources

Resources generally encapsulate some part of the graphics state, or other information specific to the renderer such as a in-memory RIB archive. Resources are always named and have a type associated with them. Resources are unique in that they can exist outside the rest of the graphics state, and are thus not subject to standard scoping rules; instead, they have their own scoping block mechanism. An example of a resource is the ability to save the entirety of the current attribute state, and restore it at a future point, independent of the current attribute stack.

```
RiResource ( RtToken handle,  RtToken type, ...)
```

Creates or operates on a named resource (with name *handle*) of a particular type. The allowed operations for the resource are specified in the parameter list, and are specific to the type of resource being manipulated.

A named resource type which is recommended for all implementations of the RenderMan Interface is the encapsulation of the entirety of the current attribute state. This resource is selected by specifying "attributes" for the *type*. In this case, the parameter list must contain at least the parameter "string operation" which takes a value of "save" (in order to create the saved attribute state with the given *handle*) or "restore" (to restore a previously saved attribute state). Furthermore, when restoring the state, a further optional parameter is accepted: "string subset", which specifies the subset of the saved attribute state to restore. (i.e. "shading", "transform", or "all").

RIB BINDING

```
Resource handle type
```

EXAMPLE

```
Color 0 1 0
Surface "marble"
Resource "greenmarble" "attributes" "string operation" "save"
Sphere 1 0 1 360
Color 1 0 0
Surface "plastic"
Cone 0.5 0.5 360
Resource "greenmarble" "attributes" "string operation" "restore" "string subset" "shading"
Cylinder 0.5 0 1 360
```

In this example, a resource named "greenmarble" of type "attributes" has been created with the "save" operation. A green marble sphere is then immediately defined. The attribute state is then altered and a red plastic cone is created. Finally, the previously saved resource "greenmarble" is restored with the "restore" operation. Depending on the implementation, this restores the shading part of the attribute state such that the subsequent cylinder is green and uses a marble shader, instead of being red and plastic.

The following table describes which parts of the graphics state are manipulated by "string subset". In addition, PRMan allows for multiple subsets to be combined using comma separated lists. Hence, restoring the "shading,geometry" subset will be the same as restoring both the "shading" and "geometry" subsets.

Subset Name	Attributes Restored by Subset
shading	• All shaders (RiSurface, RiDisplacement, RiAtmosphere, RiInterior,RiExterior) • RiColor • RiGeometricApproximation • RiMatte • RiOpacity • RiScopedCoordinateSystem (see note below) • RiTextureCoordinates • Attribute "derivatives" "centered" • Attribute "displacementbound" • Attribute "irradiance" • Attribute "grouping" • Attribute "shade" • Attribute "trace" • Attribute "visibility" • Attribute "user" (see note below)
transform	Transformation calls (RiConcatTransform, RiTranslate, etc.)
geometrymodification	• RiDetail • RiDetailRange • RiOrientation • RiReverseOrientation • RiSides • Attribute "cull" • Attribute "sides" • Attribute "stitch" • Attribute "dice"
geometrydefinition	• RiBasis • RiSolidBegin • RiSolidEnd • RiTrimCurve • Attribute "identifier" • Attribute "trim"
hiding	Attribute "hide"

- **Note**

> *PRMan currently has limitations related to user attributes and scoped coordinate systems: user attributes or coordinate systems whose values are overridden while in anAttributeBegin-AttributeEnd scope may not save the correct value (the last value set in the current scope will be restored). In other words:*

```
AttributeBegin
   Attribute "user" "string mytex" ["grass.tex"]
   Resource "grass" "attributes" "string operation" "save"
   Attribute "user" "string mytex" ["flowers.tex"]
AttributeEnd
Resource "flowers" "attributes" "string operation" "save"
Resource "grass" "attributes" "string operation" "restore" "string subset" "shading"
```

When the named attribute state "grass" is restored, the attribute user:mytex will contain flowers.tex, not grass.tex. This limitation may be addressed in a future release.

Resources can be explicitly saved and restored with the following commands:

```
  RiResourceBegin ()
RiResourceEnd ()
```

Push and pop the current set of resources. Resources defined (named) in the current ResourceBegin scope will cease to exist at ResourceEnd. If a resource is defined outside any ResourceBegin/End scope, that resource is deemed to be global and will persist indefinitely, or at least until FrameEnd depending on the nature of the resource. Otherwise, pushing and popping of resources must be properly nested with respect to various Begin-End constructs.

RIB BINDING

```
  ResourceBegin -
ResourceEnd -
```

EXAMPLE

```
Color 1 0 0
Surface "plastic"
Resource "foo" "attributes" "string operation" "save"
ResourceBegin
  Color 0 1 0
  Surface "marble"
  Resource "foo" "attributes" "string operation" "save"
ResourceEnd
```

In this example, two resources, both named "foo" and of type "attributes", have been created with the "save"operation. The first resource is global and (depending on the implementation) stores attribute state: namely, that the color is red and the surface is plastic.

The second resource's lifetime is scoped by the ResourceBegin and ResourceEnd calls and stores attribute state: green color and marble surface. Hence due to the scoping, references to "foo" within the ResourceBegin/End block will resolve against the second resource (green, marble). After the ResourceEnd, the second resource has been destroyed and the first resource is again in scope, and hence references to "foo" will resolve against the first resource (red, plastic).

Conditional Evaluation

```
RiIfBegin ( RtToken expression )
RiElseIf ( RtToken expression )
RiElse
RiIfEnd
```

These calls form the basis of a simple conditional evaluation mechanism that allow RIB archives to be constructed with a degree of context sensitivity.

Normally the elements in a RIB archive are selected by the application that is authoring the archive. Complex logic and data manipulations are properly the domain of a true programming language with RenderMan Interface binding. When RIB variations are needed in these cases, they are regenerated by the authoring application.

However, sometimes a previously generated RIB archive is reused in a different context and it may need some internal ability to adapt to each context. These RIB archives allow chunks of "frozen" geometry or scene state to be stored and then accessed from a higher level "driver" RIB file. The archive can be reused from multiple places in the same frame or across multiple frames. For example, an object whose shape remains the same across many frames might be placed in an archive, it's position and orientation might be animated by specifying a different transformation matrix in each per-frame driver file before referencing the object's archive. This modularity has many benefits, including the potentially large savings in per-frame RIB generation time when the archived object is very complex.

There are situations in which it is very useful to allow certain internal aspects of an archive to be altered based on the current external driver file state or parameters. For example, the archive might select entirely different surface shaders depending on which "rendering pass" is active. The driver file might define the current pass with a user Attribute setting:

```
Attribute "user" "string renderpass" ["shadow"]
Procedural "DelayedReadArchive" ["archive.rib"] [0 1 0 1 0 1]
```

Then, the archive can use the conditional evaluation calls to decide which shaders to apply:

```
AttributeBegin
IfBegin "$user:renderpass == 'shadow'"
   Surface "null"
ElseIf "$user:renderpass == 'beauty'"
   Surface "rmarble"
Else
   Surface "plastic"
IfEnd
Sphere 1.0 -1.0 1.0 360.0
AttributeEnd
```

Similarly, a single archive might be used by several versions of the renderer, such as during testing or when an asset is shared between productions. The "$renderer" namespace provides a "version" object with several useful values: major, minor, and build.

```
IfBegin "$renderer:version.major >= 21"
    Bxdf "simple/PxrHair" "hair" "color diffuseRootColor" [1.0 0.5 0.25]
Else
    Color [1.0 0.5 0.25]
    Surface "plastic"
IfEnd
```

Expression syntax:

The conditional expressions evaluated by IfBegin and ElseIf are similar to those found in C and many scripting languages. The entire expression evalutes to a numeric result, and if the result is non-zero then the associated branch of the If-Else block becomes active. The expression operators work on values that are string or numeric literals, or renderer state variables. A typical set of arithmetic, relational, and logical operators are provided, plus a few additional functions:

State Variables	look up Attribute and Option values	$name
Arithmetic	take numbers, return numbers	+ - * / **
Bit Mask	bit-wise integer And, Or, Xor	& | ^
Relational	take numbers or strings; return 1 if the relation holds, 0 if it doesn't; strings are compared using strcmp()	== != < <= >= >
String Match	glob-style matching, the pattern can contain '*' and '?' wildcards.	string =~ pattern
Logical	treats non-zero as true, zero as false; return 1 if the logical assertion holds, 0 if it doesn't	&& || !
Grouping		(subexpression) 'string literal'
Variable Existence	returns 1 if the state variable exists, 0 if it doesn't	defined(name)
Concatenation	combines strings	concat(string , string)
Computed Variable Names	look up name given by subexpression	$(subexpression)

State variable names are looked up by searching the Attribute stack, then the Options, then RendererInfo. The search can be restricted by prepending an additional "namespace" qualifier:

```
$Frame                        -->   finds the Option (current frame number)
$limits:eyesplits             -->   finds the Attribute
$Attribute:limits:eyesplits   -->   also finds the Attribute
$Option:limits:eyesplits      -->   finds the Option
```

Traditional static archives will continue to be the right choice in most situations. With proper archive "factoring" the creator of the driver file can select appropriate archives as needed and hardcode their names in the driver. Conditional evaluation becomes useful when factoring isn't possible or when it can help reduce the number of required archives

RIB BINDING

```
 IfBegin expression
 ElseIf expression
 Else
 IfEnd
```

EXAMPLE

Using Computed Variable Names (i.e. $(subexpression)) in a conditional RIB statement. The use of computed variable names is similar to evaluating expressions created from the contents of other variables (like performing 'eval' or 'expr' from some shells).

> *prman x.rib main.rib ---> renders a yellow matte sphere*
>
> *prman y.rib main.rib ---> renders a magenta plastic sphere*

x.rib

```
Attribute "user" "string abc" ["x"]
```

y.rib

```
Attribute "user" "string abc" ["y"]
```

main.rib

```
##RenderMan RIB
version 3.03
FrameBegin 1
Format 128 128 1
Display "/tmp/t.tif" "tiff" "rgba"
Projection "perspective" "fov" [45]
WorldBegin
LightSource "distantlight" 1 "from" [1 1 -1]

Attribute "user" "float x1" [11]
Attribute "user" "float x2" [12]
Attribute "user" "float y1" [101]
Attribute "user" "float y2" [102]

AttributeBegin
Attribute "identifier" "name" ["mysphere"]
Translate 0 0 2.75
IfBegin "$($abc$Frame) > 100"
Color 1 0 1
Surface "plastic"
Else
Color 1 1 0
Surface "matte"
IfEnd
Sphere 1.0 -1.0 1.0 360.0
AttributeEnd
WorldEnd
FrameEnd
```