XRT Technical Reference

Release 1.0.3

August 28, 2010

CONTENTS

Part I

Introduction

Welcome to *XRT*!

XRT is a software rendering system designed to flexibly create beautiful imagery for film and other high-end applications.

This technical reference manual documents:

- a C++ scene description API
- a Python binding
- a shading language
- a shader compiler
- a texture tool
- and of course, the renderer itself

Acknowledgements

This document is mostly based on *NVIDIA Gelato(R) Technical Reference* [1] (with some additions to support the *RenderMan(R) Specification 3.2*). It could have been written as a long list of the differences with the original specification but this would have been detrimental to its readability and consistency. It is also much more positive to list capabilities instead of deficiencies.

As a result, this document includes large parts from the Gelato specification document. My hope is that, in its present form, it will be considered as a necessary tribute to a well-written specification.

Feature highlights

Before you dig deeper in the manual, I'd like to call your attention to some of the really interesting features of *XRT*

- **Ray tracing.** *XRT* is capable of ray tracing of large scenes, including raytraced shadows, reflections and "ambient occlusion" visibility queries.
- **C++ API.** *XRT*'s main API is a modern, C++-based API. It is simple (few calls) and orthogonal (usually one best way to accomplish a task). There are only a few types of geometric primitives, but they are very general. All object and scene attributes (such as surface color or camera shutter) are set through a single `Attribute` call. Custom variables (such as shader parameters and geometric primitive "vertex variables") are set through a single, simple `Parameter` call.
- **State queries and saved state.** A program or plug-in making C++ API calls to *XRT* may ask for the current value of any graphics state attribute. There are also calls in *XRT*'s API's to save all or part of the current state, name it, and later restore all or part of that saved state. This makes it possible to easily transfer collections of attributes from one part of your scene hierarchy to another, non-descended, place in the hierarchy.
- **Scene format reader plug-ins.** *XRT* is "format agnostic." Rather than prescribing a single scene file format (forcing you to convert all data into that format), *XRT* has a simple API for scene format plug-ins. When a file is input, the DSO/DLL for that format is dynamically loaded and told to read the scene file. Thus, you may store your scene in any format for which you, or a third party provide such a plug-in, and you may freely mix different files in different formats within a single scene.
- **Python binding.** *XRT* includes a scene format plug-in that reads Python scripts that make calls to the C++ API. This provides an extremely flexible, fully scriptable method of scene input.
- **Layered shaders.** Instead of allowing only a single surface, displacement, and volume shader per object, *XRT* allows, for example, several surface shaders to be called in turn, with the user able to specify that one shader's outputs be connected to another shader's inputs. This allows one to compose the operations of component shaders without modifying (or even having access to) the source code of any of the shaders involved. For example, you can make any surface glossy by layering a "gloss" shader atop any other shader, without needing the source code to either.

[1] Nevertheless, *XRT does not claim* to be Gelato-compliant.

- **Place the camera, not the world.** The *XRT* universe starts off in world space. You may place a Camera within that world, just like you would place lights or objects. (In fact, you can even place multiple cameras, although currently only the first declared camera is rendered) There is no need to treat the camera as the original origin, carefully placing the world with the inverse transformation. Of course, if there is no Camera at all, *XRT* correctly infers that you intended the original coordinate system to be the camera, and that world space is marked by the `World` call.
- **Geometry sets.** It is possible to name groups of primitives (and of course, one primitive may be in many groups). These named geometry sets may be used to specify collections of primitives visible by a particular camera, used for ray tracing, comprising area lights, and potentially for many other future uses.
- **Image I/O plug-ins.** Similar to *XRT*'s scene format agnosticism, *XRT* also has no required image formats for either input or output. There is a simple API for writing plugins that read and write image formats. An image output plug-in can allow you to have the renderer write out image files in the format of your choice, and image input plugins can be used to read texture, display images, and convert images from one format to another. *XRT* ships with image I/O plug-ins for TIFF, JPEG, PPM, PNG, and Targa formats, as well as one for displaying to an interactive image viewer.

Part II

The Major APIs

CHAPTER

ONE

THE *XRT* C++ SCENE API

This chapter describes *XRT*'s C++ API for describing scenes. *XRT*'s API routines can loosely be broken down into those that alter the graphics state *attributes*, and those that create geometric primitives. Some attributes are properties that apply to the entire scene (for example, camera parameters), but other attributes are properties that may vary from object to objects (for example, color, shader assignments, and transformation). Geometric primitives, when declared, inherit the current attribute state (including transformations and shader assignments).

1.1 Basic Concepts

1.1.1 Classes, Namespaces, and Headers

The primary API for talking *XRT* is through a C++ class called `RendererAPI`, which is defined in the header file `rendererapi.h`.

A number of other header files contain useful types, classes, and functions:

- `type.h` is used by `rendererapi.h` and provides a way to express data types to certain *XRT* API routines.
- `errormanager.h` is used by `rendererapi.h` and provides a definition of overrideable error managers and handlers.

The definitions in the above header files are in `namespace xrt`. You may reference these types and functions either by explicitly using the `xrt::` prefix, or by simply stating

```
using namespace xrt;
```

after including the header files (then the prefixes are unnecessary). For brevity, the remainder of the *Technical Reference* will assume that you are `using namespace xrt`, and will therefore omit the `xrt::` prefix.

1.1.2 Renderer Object

The renderer itself is a C++ object of type `RendererAPI`. The `RendererAPI` class provides only an interface; it contains no data members nor non-virtual methods. Actual renderer implementations are assumed to be subclassed from `RendererAPI`, and thus inherit the `RendererAPI` interface.

Renderer objects may be created by calling `RendererAPI::CreateRenderer()`, which returns a pointer to a `RendererAPI` object. All subsequent communication to the renderer is through `RendererAPI` class methods called through the pointer to the renderer. Deleting the renderer frees all resources associated with the renderer. For example:

```
RendererAPI *r = RendererAPI::CreateRenderer();
r->Camera ("main");
... API calls through r ...
r->Render ("main");
delete r; // Finished with this renderer
```

Multiple renderers may be created in succession or simultaneously, and API calls to different renderers may be freely interleaved. API calls to one renderer do not have any semantic side effects upon other renderer objects, however multiple simultaneous renderers may have performance-related side effects upon each other if they perform functions that requires competition for system resources.

Renderers may come in different implementations, selected by an optional parameter to `CreateRenderer`. While the default renderer is expected to produce images of high quality, alternate renderer implementations may provide low-quality previews, or even perform tasks other than creating images (such as merely archiving the sequence of API calls for later playback with `Input`). Optional arguments to `CreateRenderer` may also be used to supply a user-defined error manager / error handler.

1.1.3 Hierarchical Graphics State

The renderer maintains a *graphics state machine*, which is a set of names and values, called attributes. Some of those parameters (such as image resolution, camera projection, and output files) apply to the entire scene or to a particular camera. Other parameters may be different for each geometric primitive. Per-object attributes include such things as shader assignments and transformations.

The attribute state is hierarchical, in the sense that attributes (and, if you want, just the transformations) may be pushed and popped using a stack.

A scene specification is mostly a series of declarations of geometric primitives, with calls to alter the attribute state between primitives.

1.1.4 Copy-on-write

When geometric primitives (such as patches) are declared, they permanently take on the characteristics of the *current* attribute state, including their object transformation. From that point onward, the primitive keeps a reference to that attribute set, which is used as the primitive is processed. Further changes to attributes may affect yet-to-be-declared primitives, but will not change the attributes of any primitives which have already entered the system. This is known as *copy-on-write semantics*, because once a geometric primitive references an attribute state, attempts to change (write) an attribute will copy the entire attribute state and change the copy, not the original set of attributes that is already referenced by the primitive.

1.1.5 Data Type Declarations

The renderer already knows about many parameters that you might want to pass, such as the `"fov"` attribute to specify field of view, or the parameter `"P"`, which is used to pass 3D positions of geometric control vertices. These names are all pre-declared, which means that the renderer recognizes the names, knows what type of data they represent, and knows how to use the data.

In order to make the renderer extremely expandable, there are a number of ways that users can pass arbitrary data, for example, to be later used by a user-supplied shader. The renderer won't know what type of data you are passing in these cases, but there are two ways that you can convey the type information: either by embedding the type declaration in the attribute or parameter name, or by passing an explicit `Type` class. For example,

```
r->Parameter ("vertex point P", &Pvalues);
```

Here we pass a parameter "P", which has data type point and has interpolation type vertex. Alternately, we could make the equivalent call that explicitly conveys the type using a Type structure:

```
r->Parameter ("P", Type(POINT,VERTEX), &Pvalues);
```

The definition of the Type class is:

```
class Type
{
public:
    // Construct from base type and optional interp (assume non-array)
    Type (BaseType basetype, Interpolation interp=CONSTANT);

    // Construct with array length and optional interp
    Type (BaseType basetype, int arraylen, Interpolation interp=CONSTANT);

    // Construct from a string (e.g., "vertex float[3]"). If no valid
    // type could be assembled, set basetype to UNKNOWN.
    Type (const char *typestring);
    ...
};
```

Valid base types include FLOAT, INT, COLOR, POINT, VECTOR, NORMAL, MATRIX (16 floats forming a 4x4 matrix), and HPOINT (4 floats forming a homogeneous point). Valid interpolation types include CONSTANT (the default, no interpolation), PERPIECE, LINEAR, and VERTEX.

The Type class is defined in the header file type.h, which is automatically included by rendererapi.h. Type is in namespace xrt.

1.1.6 Interpolation Type

A common task is to pass user data along with a geometric primitive, and have the renderer automatically interpolate the data over the primitive and make it available to the user's shaders. In addition to specifying the data type, as above, you may also specify the *interpolation type* (sometimes called *storage class*). In addition to being able to pass a single value for the entire primitive (as above, with no interpolation type specified), all primitives support the interpolation types vertex, which requires the same number of values as the control vertex positions ("P" or "Pw") and is interpolated in the same manner as the positions; and linear, which is linearly (or bilinearly) interpolated across the primitive, and therefore may have a different number of values than the control vertices. Some primitives also support the interpolation type perpiece, which supplies one value for each section of the primitive (one per face for a Mesh, one per curve for a Curves). It's easy to see how interpolation type is combined with data type in the following example:

```
int nverts[1] = {3}; // Number of vertices for each face
int verts[3] = {0, 1, 2}; // Vertex index sequence
float Pvals[3][3] = { /* points */ };
float Cvals[3][3] = { /* colors */ };
float temp = 98.6;

r->Parameter ("vertex point P", &Pvals);
r->Parameter ("vertex color C", &colvals);
r->Parameter ("float temperature", &temp);
r->Mesh ("linear", 1, nverts, verts);
```

1.1.7 Parameter Lists

Many API routines take a variable number of parameters. Among these routines are all geometric primitive and shader declarations. For example, the `Mesh` call above passes positions, colors, and user data named "temperature". Below is another illustration using a slightly different form of passing parameters, where the type is explicitly passed as a `Type` rather than implicitly passed as part of the parameter name:

```
float Pvals[3][3] = { /* points */ };
float widthvals[3] = { /* widths */ };

r->Parameter ("P", Type(POINT,VERTEX), &Pvals);
r->Parameter ("width", Type(FLOAT,VERTEX), &widthvals);
r->Points (3);
```

Using types embedded in the parameter names is equivalent to passing an explicit `Type`. You are free to use whichever is more convenient for your application. Humans tend to prefer to use the type/name strings, whereas various automatic or machine-driven translations prefer passing a `Type`.

1.2 Parameters

```
RendererAPI::Parameter (const char *name, float value)
RendererAPI::Parameter (const char *name, int value)
RendererAPI::Parameter (const char *name, const char *value)
RendererAPI::Parameter (const char *name, const float *value)
RendererAPI::Parameter (const char *name, const int *value)
RendererAPI::Parameter (const char *name, const char **value)
RendererAPI::Parameter (const char *name, const void *value)

RendererAPI::Parameter (const char *name, Type type, float value)
RendererAPI::Parameter (const char *name, Type type, int value)
RendererAPI::Parameter (const char *name, Type type, const char *value)
RendererAPI::Parameter (const char *name, Type type, const float *value)
RendererAPI::Parameter (const char *name, Type type, const int *value)
RendererAPI::Parameter (const char *name, Type type, const char **value)
RendererAPI::Parameter (const char *name, Type type, const void *value)
```

Saves a single named parameter into the "pending parameter" list for subsequent use by `Command`, `Camera`, `Output`, `Shader`, `Light`, or a geometric primitive. Any of these routines will use all of the pending parameters, and will clear the pending list. Note that the `Parameter` routine does not copy the data itself, so the user should be careful not to overwrite or free data until after the `Command`, `Camera`, `Output`, `Shader`, `Light`, or a geometric primitive call has been made.

The *value* may be a `float`, `int`, or `string` (passed as a `char *`), or a pointer to an array of `float`, `int`, or `char *` values. Data consisting of multiple float values (including `color`, `point`, `vector`, `normal`, `hpoint`, or `matrix` data) should be passed using the `float *` version. The data type and declaration must match - for example, "`float fov`" may be called by pasing a `float` or a `float *`, but any of the other varieties of `Parameter` will produce a runtime error. It's also possible to simply pass a raw `void *` pointing to the data, but in that case there is no compile-time type safety.

Two flavors of `Parameter` exist: one passes an explicit *type* in the form of a `Type` object, while the other deduces the type from the *name* itself (as described in Section *Data Type Declarations*).

EXAMPLES:

```
r->Parameter ("float Kd", 0.5);
r->Parameter ("Ks", Type(FLOAT), 0.25);
r->Shader ("surface", "plastic"); // Uses the Kd and Ks parameters

r->Parameter ("vertex point P", &P);
r->Parameter ("C", Type(COLOR,LINEAR), &C);
r->Patch ("linear", 4, 4); // Uses the P and C parameters
```

1.3 Renderers, Cameras, Outputs, and Rendering

`static RendererAPI*`
`RendererAPI::`**CreateRenderer** `(const char *type=NULL, xrt::ErrorManager *err=NULL)`

`CreateRenderer` creates a renderer of given type (if type is unspecified, a default *type* is assumed), returning a pointer to a `RendererAPI` object to which subsequent API requests may be sent. The renderer will be destroyed and its resources freed simply by deleting it.

XRT recognizes the following renderer *types*:

- No *type*, or an empty string for *type*, indicates the default renderer, which is a full high-quality rendering of the scene.

`RendererAPI::`**Camera** `(const char *name)`

Creates a camera located at the local (CTM) origin, facing in the direction of the local $+z$, with local $+x$ pointing toward the right of the screen and the local $+y$ pointing toward to top of the screen (note that this makes the camera inherently "left-handed").

There may be multiple `Camera` statements. All cameras that have corresponding `Output` statements will render images, but cameras without `Output` statements will not produce images (unless no `Output` statement is present at all in the scene, in which case the first camera declared will be automatically given a default `Output`).

Any pending parameters (set by `Parameter`) give camera and image formation attributes (summarized below and explained in detail in Section *Camera Attributes*). `Camera` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

All cameras must have distinct *names*. If `Camera` is called with the *name* of an existing camera, the old camera defintition will be replaced by the new camera definition.

EXAMPLE:

```
r->SetTransform (M);
r->Parameter ("float fov", 32.0f);
int res[2] = { 640, 480 };
r->Parameter ("int[2] resolution", &res);
r->Parameter ("float fov", fov);
r->Camera ("maincam");
```

Camera Attribute	Meaning (default value)
"string projection"	Projection ("perspective")
"float fov"	Vertical field of view (90)
"float[4] screen"	Portion of the projection plane to image (-xres/yres, xres/yres, -1, 1)
"float near"	Near clipping plane distance (0.1)
"float far"	Far clipping plane distance (1e6)
"int[2] resolution"	Image resolution (640, 480)
"float pixelaspect"	Pixel aspect ratio (1)
"float[4] crop"	Subimage to render (0, 1, 0, 1)
"float[2] shutter"	Shutter open and close time for motion blur (0, 0)
"float fstop"	f/stop for depth of field (default: no DOF)
"float focallength"	Lens focal length (default: no DOF)
"float focaldistance"	Distance to sharp focus (default: no DOF)
"int[2] spatialquality"	Number of subpixel antialiasing regions (4, 4)
"int temporalquality"	Number of time values for motion blur (16)
"int dofquality"	Number of lens values for motion blur (16)
"int[2] limits:bucketsize"	Size of pixel rectangles used as work units (32, 32)
"string bucketorder"	Order of bucket traversal ("horizontal")

RendererAPI::**Output** (const char *name, const char *format, const char *data, const char *camera)

Specifies an output image for rendered pixels, for a particular camera.

The *name* parameter is a string that gives the name of an image file, or other destination. Multiple simultaneous output images (presumably each with different data) may be specified simply by having multiple Output statements with different *names*. There is no set limit to the number of output images. If Output specifies an output name that already exists, the previous definition will be replaced by the new definition of that output.

The *format* is a string that specifies the type of file format to write.

If *format* is NULL or points to the string "null", the output will never be used (thus, an output may be effectively deleted by replacing it and using "null" for format).

The *data* parameter is a string that indicates what data to output, and may include any of the following:

- "rgb": output a 3-channel file containing color.
- "rgba": output a 4-channel file containing color and alpha (coverage/opacity).
- "z": create an image containing the z depth of the closest surface in each pixel.

Any pending parameters (set by Parameter) give output attributes (summarized below and explained in detail in Section *Output Attributes*). Output copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding Parameter calls).

Parameters respected by Output include:

"string filter"	The name of the pixel filter to use.
"float[2] filterwidth"	The width (in x and y) of the pixel filter to use.
"float gain"	The image gain (1 for none).
"float gamma"	The gamma correction (1 for no correction).
"float dither"	The dither amplitude (0 for no dither).
"int[4] quantize"	The *zero*, *one*, *min* and *max* quantization levels.
other	Additional data will be passed down to the image output format driver.

Please consult Section *Output Attributes* for detailed explanation of these parameters and their possible values.

Any parameterlist tokens other than the ones above will be passed along to the image output format driver. Check the documentation for the specific image driver to see what optional parameters it can take.

EXAMPLE:

```
r->Output ("beauty.tif", "tiff", "rgba", "maincam");
r->Output ("Beauty", "iv", "rgb", "maincam");
```

The above commands create two display output streams: (1) a TIFF file `beauty.tif` containing the color and alpha of the image using default filtering and quantization, and (2) a live image viewer.

`RendererAPI::`**World** `()`

`World` marks the end of the section where scene-wide attributes, cameras, and outputs may be set, and the beginning of the section where geometric primitives may be declared. It also resets the CTM to `"world"` space.

If no `Camera` statement was ever encountered, a camera is added to the scene assuming that the current CTM is the world position and the camera's origin was the *original* CTM position (before any transformations were encountered).

`RendererAPI::`**Render** `()`
`RendererAPI::`**Render** `(const char *camera)`

`Render` signals the end of the scene geometry, and triggers final rendering of all the output images. All cameras that have corresponding `Output` calls will be rendered. Cameras that have no corresponding `Output` will not produce images, unless *no* `Output` statement is present at all in the scene, in which case the first camera declared will be automatically given a default `Output`. After a call to `Render`, the graphics state (including CTM) is restored to the way it was immediately following the `World` call.

Default "live" renderers are *blocking*, that is, the `Render` call will not return to the caller until all rendering is completed.

`RendererAPI::`**Command** `(const char *name)`

Executes a command signified by the token name.

Any pending parameters (set by `Parameter`) give optional parameters specific to that particular command. `Command` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

Note: *This routine is currently a stub and is provided for backwards compatibility with the Gelato specification*

`RendererAPI::`**Comment** `(const char *format, ...)`

For renderer implementations that are creating an archive file, insert a comment using the usual C/C++ `printf` formatting rules. The archiving renderer implementation, not the caller, is responsible for ensuring that the comment is output with the right syntax to be perceived as a comment in the given format (such as prepending "#" and outputting a linefeed at the end, if it is outputting Pyg).

Comment calls are ignored by "live" renderers that are creating images rather than command archive files.

EXAMPLES:

```
r->Comment ("this is a comment");
```

1.4 Attributes

Attributes are properties of the scene or of objects. Examples of scene attributes include image resolution and camera projection. Examples of object attributes include color, object transformation (position/orientation), and shader assignments.

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

```
RendererAPI::Attribute (const char *name, float value)
RendererAPI::Attribute (const char *name, int value)
RendererAPI::Attribute (const char *name, const char *value)
RendererAPI::Attribute (const char *name, const float *value)
RendererAPI::Attribute (const char *name, const int *value)
RendererAPI::Attribute (const char *name, const char **value)
RendererAPI::Attribute (const char *name, const void *value)

RendererAPI::Attribute (const char *name, Type type, float value)
RendererAPI::Attribute (const char *name, Type type, int value)
RendererAPI::Attribute (const char *name, Type type, const char *value)
RendererAPI::Attribute (const char *name, Type type, const float *value)
RendererAPI::Attribute (const char *name, Type type, const int *value)
RendererAPI::Attribute (const char *name, Type type, const char **value)
RendererAPI::Attribute (const char *name, Type type, const void *value)
```

The *value* may be a `float`, `int`, or `string` (passed as a `char *`), or a pointer to an array of `float`, `int`, or `char *` values. Data consisting of multiple float values (including `color`, `point`, `vector`, `normal`, `hpoint`, or `matrix` data) should be passed using the `float *` version. The data type and declaration must match - for example, `"float fov"` may be called by pasing a `float` or a `float *`, but any of the other varieties of `Attribute` will produce a runtime error. It's also possible to simply pass a raw `void *` pointing to the data, but in that case there is no compile-time type safety.

Two flavors of `Parameter` exist: one passes an explicit *type* in the form of a `Type` object, while the other deduces the type from the *name* itself (as described in Section *Data Type Declarations*).

Certain attributes apply to the entire scene and cannot vary from object to object. Attempts to set these attributes with `Attribute` after `World` will be ignored, and an error message will be printed. The individual attributes' descriptions will point out which ones cannot be set per-object.

EXAMPLES:

```
// Examples of types declared in the name
r->Attribute ("float fov", 45.0f); // Pass a float
r->Attribute ("string projection", "perspective"); // string
int res[2] = { 640, 480 };
r->Attribute ("int[2] resolution", res); // Pass int[2]
float temp = 98.6;
r->Attribute ("float user:temperature", &temp); // ptr to float

// Examples of passing an explicit type
r->Attribute ("fov", Type(FLOAT), 45.0f);
r->Attribute ("projection", Type(STRING), "perspective");
```

```
bool RendererAPI::GetAttribute (const char *name, float &value)
```

```
bool RendererAPI::GetAttribute (const char *name, int &value)
bool RendererAPI::GetAttribute (const char *name, char* &value)
bool RendererAPI::GetAttribute (const char *name, float *value)
bool RendererAPI::GetAttribute (const char *name, int *value)
bool RendererAPI::GetAttribute (const char *name, char **value)
bool RendererAPI::GetAttribute (const char *name, void *value)
```

Gets the value of a particular attribute (specified by name as a string) in the current attribute state. If the attribute is found, its value is written into the memory pointed to by *value*, and `GetAttribute` returns `true`. If the attribute name is not found or its type does not match the type declaration in *name*, `GetAttribute` returns `false` and the data in *value* is not altered. It is up to the user to ensure that value points to a big enough area for the data type requested.

EXAMPLE:

```
int resolution[2];
bool found = r->GetAttribute ("resolution", resolution);
```

```
RendererAPI::PushAttributes ()
RendererAPI::PopAttributes ()
```

Save and restore the per-object attribute state, including the current transformation (see `PushTransform` and `PopTransform`, section *Transformations*). Upon `PopAttributes`, the current attribute set is replaced by the attribute set that was in effect at the corresponding `PushAttributes`. It is perfectly legal to nest blocks delimited by `PushAttributes`/`PopAttributes`.

EXAMPLE:

```
r->PushAttributes()
r->PopAttributes()
```

```
RendererAPI::SaveAttributes (const char *name, const char *attrs = NULL)
```

Create a named alias for part or all of the current per-object attribute state in a global dic tionary of name/attribute state pairs. The *name* may be used with `RestoreAttributes`.

The optional string *attrs* is a comma-separated list of which attributes should be saved (passing NULL indicates that all attributes should be saved). Those attributes may be any of the named attributes described throughout this section, and also recognize the following special names:

`"transform"`	The current transformation
`"shaders"`	All shader assignments, except for lights
`"surface"`	Just the surface shader assignment
`"displacement"`	Just the displacement shader assignment
`"volume"`	Just the volume shader assignment
`"lights"`	The active light list
`"trimcurve"`	The trim curve for `Patch` primitives.
`"user"`	All of the `user:` attributes

EXAMPLE:

```
r->SaveAttributes ("leftarm");
r->SaveAttributes ("creature_shaders", "shaders,C,opacity");
```

```
RendererAPI::RestoreAttributes (const char *name, const char *attrs = NULL)
```

Replaces some or all of the current per-object attribute state with the saved attribute state with the given *name* (set by `SaveAttributes`).

The optional string *attrs* is a comma-separated list of which attributes, out of those saved by the corresponding `SaveAttributes`, should be restored (passing NULL indicates that all saved attributes should be restored). The meanings of the attribute names are as described by the documentation for `SaveAttributes`.

EXAMPLE:

```
r->RestoreAttributes ("leftarm", "C");
r->RestoreAttributes ("creature_shaders");
```

`RendererAPI::`**PushOptions** `()`
`RendererAPI::`**PopOptions** `()`

Save and restore the scene attribute state including:

- scene-wide attributes
- camera attributes (see section *Camera Attributes*)
- output attributes (see section *Output Attributes*)
- the active lights (see section *Lights*)
- the object definitions (section *Object Instancing*)

Upon `PopOptions`, the current scene state set is replaced by the scene state that was in effect at the corresponding `PushOptions`. It is perfectly legal to nest blocks delimited by `PushOptions`/`PopOptions`.

EXAMPLE:

```
r->PushOptions()
r->PopOptions()
```

`RendererAPI::`**Modify** `(const char *namepattern=NULL)`

Note: *This routine is currently a stub and is provided for backwards compatibility with the Gelato specification*

Scene-wide Attribute	Meaning (default value)
`"color ray:background"`	Color of rays that hit nothing (`0, 0, 0`)
`"int ray:maxdepth"`	Maximum ray recursion depth (`2`)
`"string path:input"`	Search path for scene files (`".:$XRT_HOME/inputs"`)
`"string path:texture"`	Search path for texture files (`".:$XRT_HOME/textures"`)
`"string path:shader"`	Search path for compiled shaders (`".:$XRT_HOME/shaders"`)
`"string path:generator"`	Search path for generator DSO's (`".:$XRT_HOME/plugins"`)
`"string path:imageio"`	Search path for image format input/output DSO's (`".:$XRT_HOME/plugins"`)

Per-Object Attribute	Meaning (default value)
"color C"	Default surface base color (1, 1, 1)
"color opacity"	Default surface opacity (1, 1, 1)
"string orientation"	Which way the normals face ("outside")
"int twosided"	Is the object visible from both sides? (1)
"string name"	Object name, used mainly for clear error reporting ("")
"string geometryset"	Name of active geometry sets ("camera")
"matrix transform"	CTM at shutter open time. (read only)
"string trimcurve:sense"	Whether to discard the inside or outside of trim curves on Patch primitives ("inside")
"int light:nsamples"	Number of area light samples (1)
"float shadow:bias"	Default bias for ray and mapped shadows (0.01)
"int ray:opaqueshadows"	Are ray-traced objects opaque regardless of their shaders (1)

1.5 Transformations

As it receives scene commands, the renderer keeps track of the *current transformation* (sometimes called the CTM for "current transformation matrix"). When a geometric primitive is declared, a copy of the CTM is permanently attached to that primitive, which will consider the CTM as its "object" space. Therefore, any spatial quantities (such as vertex positions "P") passed on the primitive are relative the CTM that was in effect at the time that the geometric primitive command was encountered. Thus, transformation commands affect the appearance of subsequently declared geometry, but do not change previously declared geometry.

A number of named coordinate systems are predefined, or implicitly defined by API methods such as World and Camera. The predefined coordinate systems are:

Name	Meaning
"world"	The coordinate system active at World.
"camera"	The coordinate system with its origin at the center of the camera lens, *x*-axis pointing right, *y*-axis pointing up, and *z*-axis pointing into the screen.
"screen"	The coordinate system of the camera's image plane (after perspective projection, if one is specified). Coordinate (0,0) in the center of the screen.
"raster"	2D pixel coordinates. The upper left corner of the image in "raster" space is (0,0), and the lower right corner is (*xres*, *yres*).
"NDC"	2D Normalized Device Coordinates. The upper left corner of the image in "NDC" space is (0,0), and the lower right corner is (1,1).

It is convenient to save the transformation state, modify the transformation and declare geometric primitives, then restore the transformation to its prior condition. A command is provided to perform this action:

RendererAPI::**PushTransform** ()
RendererAPI::**PopTransform** ()

> Save and restore the current transformation. Upon PopTransform, the current transformation is set to the transformation that was in effect at the corresponding PushTransform.
>
> It is perfectly legal to nest blocks delimited by PushTransform/ PopTransform.

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

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

RendererAPI::**SetTransform** (const float *M)
RendererAPI::**SetTransform** (const char *name)

Replace the current transformation with the 4x4 matrix supplied. If the `SetTransform` routine is called before `World`, then M is assumed to be relative to `"camera"` space, whereas a call to `SetTransform` after `World` assumes that M is relative to `"world"` space.

If a string is passed rather than a matrix, replace the current transformation with the named transformation, which may either be the name of a saved attribute state defined by `SaveAttributes` (which must, of course, have saved the transformation), or the name of a standard coordinate system (`"world"`, `"camera"`, `"screen"`, `"raster"`, `"NDC"`).

EXAMPLES:

```
Matrix4 M(1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 3, 0, 0, 1);
r->SetTransform ((const float *)&M)

r->SetTransform ("world");
```

`RendererAPI::`**AppendTransform** `(const float *M)`

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

EXAMPLE:

```
Matrix4 M(1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 3, 0, 0, 1);
r->AppendTransform ((const float *)&M);
```

(This example assumes that the `Matrix4` type consists of 16 contiguous `float` variables, and so can be safely cast to a `float *`.)

The `SetTransform` and `AppendTransform` routines, which replace and concatenate the CTM, respectively, are fully general. For several of the most useful and common transformations, there are specific routines that have a more compact, simpler syntax: `RendererAPI::`**Translate** `(float x, float y, float z)`

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

```
Matrix4 M(1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, x, y, z, 1);
r->AppendTransform ((float *)&M)
```

EXAMPLE:

```
r->Translate (2, 0, 0);
```

`RendererAPI::`**Rotate** `(float angle, float x, float y, float z)`

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

EXAMPLE:

```
r->Rotate (30, 0, 0, 1);
```

`RendererAPI::`**Scale** `(float sx, float sy, float sz)`

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

```
Matrix4 M(sx, 0, 0, 0, 0, sy, 0, 0, 0, 0, sz, 0, 0, 0, 0, 1);
r->AppendTransform ((float *)&M)
```

EXAMPLE:

```
r->Scale (1, 2, 1);
```

`RendererAPI::`**LookAt** `(float posx, float posy, float posz,`
`float atx, float aty, float atz, float upx, float upy, float upz)`

Concatenates a viewing transformation matrix onto the current transformation, given a camera position, target, and "up" vector.

The `LookAt` routine is designed to easily position and orient a camera. It is an error to use this routine without a corresponding `Camera` call. Without a `Camera` call, *XRT* will assume that the `LookAt` call actually defines the world position. See Section *Renderers, Cameras, Outputs, and Rendering* for more explanations.

This routine is identical to the sequence:

```
Point pos = Point (posx, posy, posz)
Point target = Point (atx, aty, atz)
Vector dir = normalize (target - pos);
Vector up = Vector (upx, upy, upz);
up = normalize (up);
Vector right = cross (up, dir);
up = cross(dir, right);
float M[4][4];
M[0][0] = right[0];  M[0][1] = right[1];  M[0][2] = right[2];  M[0][3] = 0;
M[1][0] = up[0];     M[1][1] = up[1];     M[1][2] = up[2];     M[1][3] = 0;
M[2][0] = dir[0];    M[1][2] = dir[1];    M[2][2] = dir[2];    M[2][3] = 0;
M[3][0] = posx;      M[3][1] = posy;      M[2][3] = posz;      M[3][3] = 1;
r->AppendTransform ((float*)M);
```

EXAMPLE:

```
r->Lookat (0, 0, 0, 1, 0, -2, 0, 1, 0);
r->Camera ("main");
```

1.6 Shaders and Lights

RendererAPI::**Shader** (const char *shaderusage, const char *shadername, const char *layername=NULL)
RendererAPI::**Shader** (const char *shaderusage)

Sets the shader specified by *shadername* to be used as the current shader of the given *shaderusage*. Valid tokens for *shaderusage* include `"surface"`, `"displacement"`, or `"volume"`, and must match the shader type that was declared in the shader source code.

If only the *shaderusage* is given (with no shader name), that shader usage will be cleared (i.e., no shader will be assigned for that usage).

The optional *layername*, if specified, gives a name to the shader layer (for later use with `ConnectShaders`). See `ShaderGroupBegin` for information on how to specify more than one shader of each type.

Any pending parameters (set `by Parameter`) give parameters specific to the particular shader being used. `Shader` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

EXAMPLE:

```
r->Parameter ("float Kd", Kd);
r->Shader ("surface", "plastic");
```

RendererAPI::**ShaderGroupBegin** ()
RendererAPI::**ShaderGroupEnd** ()
RendererAPI::**ConnectShaders** (const char *srclayer, const char *srcparam,

`const char *dstlayer, const char *dstparam)`

`ShaderGroupBegin` and `ShaderGroupEnd` denote that repeated calls to `Shader` within the block should create a group of shaders of the same type which will be executed in sequence on subsequent geometry. This lets you, for example, bind several `"surface"` shaders at once. Any layer names passed to `Shader` are valid only within the enclosing ShaderGroup block (and thus, calls to `ConnectShaders` must be inside the same ShaderGroup block).

`ConnectShaders` connects two shaders such that the source shader's parameter named by *srcparam* will be used as input for the destination shader's *dstparam* (overriding any shader defaults or other bindings for that parameter). *srclayer* and *dstlayer* both refer to shaders on the object, referenced by the layer names passed to calls to `Shader`. The srcparam either be the name of a parameter of *srclayer* (presumably, but not required to be, an `output` parameter) or the name of a global variable (such as `P`, `N`, `C`, etc.), and *dstparam* must be either the name of a parameter to *dstlayer* or the name of a global variable.

The source and destination parameters are presumed to be of the same data type, and if they are arrays, they must have the same array length. Type conversions are fairly flexible for non-arrays, though; in particular, any triple (color point, normal, vector) may connect to another triple, a float may connect to a triple (replicating the value for all three components), and a triple may connect to a float (passing just the first component). Failure of the names to be found or of the parameter data types to be compatible will result in an error message, and no connection being made.

It is possible to connect a single channel of a triple to a scalar type, or a scalar to one channel of a triple, or one channel of a triple to a channel of a triple. It is also possible to connect an individual array element to a scalar type, or a scalar to an array element, or an array element to another array element (in which case the two arrays need not be the same size). The element data types must still be compatible with each other, but like scalars, triple-to-triple, float-to-triple and triple-to-float conversions are done automatically).

These routines may be used to create a layered light source, but in that case it would be `Light` calls inside the `ShaderGroupBegin` / `ShaderGroupEnd` block, and all of the `Light` calls are expected to have the same *lightid*. (See the description of `Light`, below, for details.)

EXAMPLE:

```
r->ShaderGroupBegin ();
r->Parameter ("string texturename", txname);
r->Shader ("surface", "texmap", "layer1");
r->Parameter ("float Kd", Kd);
r->Shader ("surface", "plastic", "layer2");
r->ConnectShaders ("layer1", "out", "layer2", "Ks");
r->ShaderGroupEnd ();
```

Example of automatic type conversion:

```
r->ConnectShaders ("layer1", "floatvar", "layer2", "colorvar");
```

Examples of array element and color channel connections:

```
r->ConnectShaders ("layer1", "floatarrayvar[3]", "layer2", "floatvar");
r->ConnectShaders ("layer1", "colorvar", "layer2", "colorarrayvar[1]");
r->ConnectShaders ("layer1", "floatvar", "layer2", "colorvar[1]");
r->ConnectShaders ("layer1", "colorvar[1]", "layer2", "floatvar");
```

`RendererAPI::`**Light** `(const char *lightid, const char *shadername, const char *layername=NULL)`

Makes a new light source or replaces an existing light source. The light source is also added to the list of active light sources in the attribute state (that is, it will illuminate subsequent geometry). The light source (or layer) will use the light shader specified by *shadername*, and its `"shader"` space will be the CTM at the time of the `Light` call.

lightid parameter is a unique identifier for the light. If *lightid* is the same as that of an existing light, the old definition of that light will be replaced by the new definition of the light. If *shadername* is NULL or points to the string `"null"`, the light will never be used, effectively removing it from the scene.

When within a `ShaderGroupBegin` / `ShaderGroupEnd` block, `Light` will append a layer to the light being specified. All layers of the light must use the same *lightid*. The optional *layername*, if specified, gives a name to the layer for later use with `ConnectShaders`.

Any pending parameters (set by `Parameter`) give parameters specific to the particular shader being used. `Light` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

If a geometry set is passed to the light via `Parameter` as the variable `"string geometry"`, the set of geometric primitives defines an area light source.

EXAMPLES:

```
float lightcolor[3] = { .9, 1, .7 };
r->Parameter ("color lightcolor", lc);
r->Light ("fill1", "pointlight");

r->Parameter ("float intensity", 10.0f);
r->Parameter ("string geometry", "myareageom");
r->Light ("key", "uberlight");
```

RendererAPI::**LightSwitch** `(const char *lightid, bool onoff)`

Add or remove a light from the active light list. The *lightid* is the identifier passed to a previous call to `Light`. If *onoff* is `false`, the light is removed from the active light list, and therefore does not shine on subsequently declared geometry. If *onoff* is `true`, the light is added to the active light list (if it not already active), and therefore will shine on subsequently declared geometry.

EXAMPLE:

```
r->Light ("key", "spotlight");
...
r->LightSwitch ("key", false);
r->Patch (...); // key light will not shine on this patch
```

1.7 Motion Blur

Geometry and transformations may be motion-blurred over the course of a rendered frame to simulate how a real camera captures light over a finite interval rather than instantaneously. The shutter interval of the camera (i.e., the time range over which light exposes the image) is specified to the `Camera` call (see Section *Renderers, Cameras, Outputs, and Rendering*) as the optional `"float[2] shutter"` parameter.

Both geometric primitives and transformations may be blurred (i.e., described in a changing way over time) using the `Motion` method described below. The camera shutter interval need not be identical to the times given to `Motion` calls, and multiple `Motion` calls that apply to the same object need not have identical time values.

Motion blur will only take effect if a camera shutter interval is specified. If the camera is not given a `"shutter"` parameter, all objects will be drawn in their configuration at the earliest time of their motion descriptions.

RendererAPI::**Motion** `(int ntimes, float time0, ...)`
RendererAPI::**Motion** `(int ntimes, const float *times)`

`Motion` marks the start of a *motion block* and specifies (either with a variable parameter list or an array) *ntimes* time values. It is expected that the `Motion` call is immediately followed by

1. exactly *ntimes* transformation calls (e.g., `Translate`, `AppendTransform`, etc.) of the same name but with different numerical parameters; or
2. *ntimes* geometric primitives (including `Parameter` calls, if necessary) that differ in the values of their parameters.

Each of the *ntimes* transformations or primitives corresponds to the position or shape at the respective time passed to the `Motion` call.

Motion-blurred transformations will be linearly interpolated over each motion segment. At times before the first motion time, the transformation will be identical to that of the first motion time; at times after the last motion time, the transformation will be identical to that of the last motion time. That is, transformations simply do not move outside their specified time intervals.

Motion-blurred geometric primitives also linearly interpolate their control vertices or parameters over each motion segment. Primitives *do not exist* outside the range of their motion segments. Thus, objects whose motion descriptions only partially overlap the camera's shutter interval may indeed only be imaged for part of the shutter interval.

For motion-blurred geometric primitives, all *ntimes* primitives must be the same primitive type and the same "shape." That is, a primitive may not "morph" between two types (say, from a `Mesh` into a `Patch`), nor may it change the list of parameters or their sizes (e.g., the primitive must have the same number of control vertices at each time). Only the *numerical values* of the parameters may change over time.

Note: *deformation blur is not yet implemented*

EXAMPLE:

```
r->Motion (2, 0.0, 1.0/48.0);
r->Translate (0, 0, 0);
r->Translate (0, 0, 1);
```

The above example makes a single translation, but with two time values, so that the transformation translates one unit in z over the time interval between $t = 0$ and $t = 1/48$.

1.8 Geometric Primitives

The renderer supports 0-, 1-, and 2D geometric primitives. The 0D primitives are points, useful for particle systems. The 1D primitives are line or curve segments, useful for hair. The 2D primitives are broken down into patches, meshes and quadrics.

Primitives inherit current attributes

Geometric primitives, when declared, inherit the current attribute state (including transformations and shader assignments). Therefore, all spatial (point, normal, vector, matrix) data passed to the primitive, including control vertices, are expressed in the object coordinate system, and will be transformed to a common space by the renderer. Subsequent changes to attributes do not affect previously-declared primitives.

Control vertices and primitive variables

With the exception of quadrics, all other primitives have their shapes specified by a series of *control vertices*. The actual shape of the object is defined by some kind of interpolation of the control vertices. Control vertices are passed via `Parameter` as the variable `"vertex point P"`. Some primitives allow rational (4-D homogeneous) points, which are passed as the variable `vertex hpoint Pw"`. All primitives defined by control vertices must have either `"P"` or `"Pw"` (but not both).

A primitive may have use `Parameter` to override the default surface color or surface normal by attaching the variables `"color C"` or `"normal N"`, respectively. If no `"C"` is attached, the default surface color will be the value of the `"C"` Attribute (see Section *Surface Appearance Attributes*). It is probably not useful to override the default

normal "N" for truly curved-surface primitives (such as Patch or Mesh("catmull-clark") but is frequently used when using faceted geometry such as Mesh("linear") to provide smoothed normals. .

Other *primitive variables* may be attached to geometric primitives via Parameter. These are automatically interpolated, according to the interpolation type, and passed along to become the values of any identically-named parameters of any of the surface, displacement, or volume shaders attached to the primitive.

Primitive variables have several choices of interpolation type. All primitives support interpolation type vertex, which requires the same number of values as the control vertex positions ("P" or "Pw") and is interpolated in the same manner as the positions; and linear, which is linearly (or bilinearly) interpolated across the primitive, and therefore may have a different number of values than the control vertices. Some primitives also support the interpolation type perpiece, which supplies one value for each section of the primitive (one per face for a Mesh, one per curve for a Curves).

1.8.1 Patches

RendererAPI::**Patch** (const char *interp, int nu, int nv)

Specifies a rectangular array of nu * nv control vertices forming a mesh that is piecewise linear or cubic. In this simple form, the mesh is parameterized uniformly on [0,1] using the specified interpolation type: "linear" for piecewise linear, or one of several piecewise cubic types, including "bezier", "bspline", and "catmull-rom".

It is important for *nu* and *nv* to be appropriate for the interpolation type (e.g., >= 2 for linear, >= 4 for cubic, 4 + 3 *i* for Bezier).

Pending parameters (set by Parameter) give control vertices and primitive variables to be interpolated across the surface. The parameters must include either 3D control vertices (passed as the "P" parameter), or 4D hpoint control vertices ("Pw") to indicate a rational patch.

Primitive variables with interpolation type vertex require *nu* * *nv* values (i.e., the same number of values as "P" or "Pw"; linear primitive variables require 4 values, which are interpolated bilinearly across the patch; and primitive variables without an interpolation type require a single data value which does not vary across the patch.

Patch copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding Parameter calls).

EXAMPLE:

```
float P[] = { 1, 0, 0,  1, 1, 0,  0, 1, 0,  -1, 1, 0,  -1, 0, 0,
                        -1, -1, 0, 0, -1, 0, 1, -1, 0,  1, 0, 0,   1, 0, -3,
                        1, 1, -3, 0, 1, -3,-1, 1, -3, -1, 0, -3, -1, -1, -3,
                        0, -1, -3,1, -1, -3,1, 0, -3 };


r->Parameter ("P", Type(POINT,VERTEX), P);
r->Patch ("bspline,linear", 9, 2);
```

RendererAPI::**Patch** (int nu, int uorder, const float *uknot, float umin, float umax, int nv, int vorder, const float *vknot, float vmin, float vmax)

Specify a rectangular array of *nu* * *nv* control vertices forming a NURBS mesh. In this more complex form, the user may explicitly specify *u* and *v* orders, knot vectors, and parameter subranges. The length of uknot must be *nu* + *uorder* and the length of vknot must be *nv* + *vorder*. The surface is defined over the parametric range [*umin*... *umax*, *vmin*... *vmax*].

Pending parameters (set by Parameter) give control vertices and primitive variables to be interpolated across the surface. The parameters must include either 3D control vertices (passed as the "P" parameter),

or 4D `hpoint` control vertices (`"Pw"`) to indicate a rational patch.

Primitive variables with interpolation type `vertex` require `nu` * `nv` values; `linear` primitive variables require 4 values, which are interpolated bilinearly across the patch; and primitive variables without an interpolation type require a single data value which does not vary across the patch.

`Patch` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

EXAMPLE:

```
float uknot[] = { 0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 4 };
float vknot[] = { 0, 0, 1, 1 };
float Pw[] = { 1, 0, 0, 1,  1, 1, 0, 1,  0, 2, 0, 2, -1, 1, 0, 1,
                        -1, 0, 0, 1, -1, -1, 0, 1, 0, -2, 0, 2, 1, -1, 0, 1,
                        1, 0, 0, 1,  1, 0, -3, 1, 1, 1, -3, 1, 0, 2, -6, 2,
                        -1, 1, -3, 1,-1, 0, -3, 1,-1, -1, -3, 1, 0, -2, -6, 2,
                        1, -1, -3, 1, 1, 0, -3, 1 };
r->Parameter ("vertex hpoint Pw", Pw);
r->Patch (9, 3, uknot, 0, 4, 2, 2, vknot, 0, 1);
```

`RendererAPI::`**TrimCurve** `(int nloops, const int *ncurves, const int *n, const int *order, const float *knot, const float *min, const float *max, const float *uvw)`

Trim curves define regions in a `Patch`'s parametric space that will be removed (or will be kept, with the remainder removed, depending on the setting of the `"trimcurve:sense"` attribute; see Section *Trim Curve Control*).

Sets the current *trim curve* that will apply to subsequently defined `Patch` primitives. This API call actually sets an attribute, and thus the trim curve is saved and restored with the `PushAttributes`, `PopAttributes`, `SaveAttributes`, and `RestoreAttributes` routines. Trimming will be turned off for subsequent geometry if *nloops* is zero (in which case the values of the other arguments are unused).

Trim curves define regions in a `Patch`'s parametric space that will be removed. The region is defined by *nloops* closed loops, each of which consists of *ncurves[i]* 2D nonuniform rational curve segments (i.e. NURBS curves). Thus, the total number of trim curve segments is the sum of all elements in *ncurves*. In respective order of the loops and curve segments, *n[j]* is the number of rational control points, *order[j]* is the order of the curve segment, *min[j]* and *max[j]* are the minimum and maximum parametric values of the curve segment. The *knots* array contains the knot vectors for all the curve segments (with each segment j having *n[j]* + *order[j]* knot values). The *uvw* array contains the control points (with each segment *j* having *n[j]* control points). The control points themselves each consist of three floating-point numbers giving a *rational (u,v,w)* 2D position in parametric space of the patch.

The symmetry between the specification of trim curves as NURBS curves and the specification of `Patch` primitives as NURBS surfaces should be apparent.

EXAMPLE:

```
int ncurves[] = { 1 };
int n[] = { 9};
int order[] = { 3 };
float uknot[] = { 0, 0, 0, .25, .25, .5, .5, .75, .75, 1, 1, 1 };
float min[] = { 0 };
float max[] = { 1 };
float uvw[] = { 1, .5, 1,  1, 1, 1,  1, 2, 2,  0, 1, 1,
                                0, .5, 1,  0, 0, 1,  1, 0, 2,  1, 0, 1,  1, .5, 1 };

r->TrimCurve (1, ncurves, n, order, knots, min, max, uvw);
```

1.8.2 Meshes

`RendererAPI::`**Mesh** `(const char *interp, int nfaces, const int *nverts, const int *verts)`

Specifies a connected mesh of faces. A mesh is not constrained to have rectangular connectivity - each face may have any number of vertices (3 or more) and any number of faces may share a vertex.

The *interp* parameter may be `"linear"` to indicate a linear interpolation (i.e., a flat polygon mesh), or `"catmull-clark"` to indicate Catmull-Clark subdivision should be used to smooth the mesh. Other interpolation schemes may be added in the future.

Note: *subdivision surfaces are not supported*

The *nfaces* parameter is the number of faces in the mesh. The array *nverts* [0.. *nfaces* -1] contains the number of vertices in each face. The array *verts*, whose length is the sum of all the entries in *nverts* [], contains the vertex indices, in order, of all faces.

Pending parameters (set `by Parameter`) give control vertices, primitive variables to be interpolated across the surface, or other parameters that control mesh behavior. The parameters must include 3D control vertices (passed as the `"P"` parameter); other parameters are optional.

Any `vertex` primitive variable must have a number of entries high enough to accommodate the highest vertex index in *verts* []. All `linear` primitive variables must have the same number of data items as the length of the *verts* array, and are in the same order as the vertex indices themselves. `Mesh` allows `perpiece` variables, which have a number of data items equal to *nfaces*, one per face. As usual, primitive variables without an interpolation type supply only one value for the entire mesh.

Several special parameters control behavior and shape of the mesh, rather than merely providing user data that will be interpolated:

`"perpiece string __attributes"`

Indicates the attribute states (previously named with `SaveAttributes`) to apply to each face. This allows the specification of a single mesh that has different attributes on a per-face basis, such as having multiple sets of faces each of which has a different surface shader. If this optional parameter is not specified, all faces will inherit the default attribute state of the primitive (that is, the attributes locally active at the time of the `Mesh` statement).

`"string[N] __attributes"`
`"perpiece int __attributesindex"`

The functionality of these two parameters is identical to that of `"perpiece string __attributes"`, but allows an alternate specification: an array of attribute names, and a per-face integer index into that array. This may be a more convenient or more compact representation for some applications. It is an error for any face to specify an index that is greater than or equal to the length of the `__attributes` array.

These special primitive variables control the behavior of the mesh, and are not passed down to the shader parameters as are ordinary user-defined primitive variables.

`Mesh` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

EXAMPLE:

```
int nverts[] = { 4, 4, 4, 4, 4, 4, 4, 4, 4};
int verts[] = { 0, 1, 5, 4,    1, 2, 6, 5,    2, 3, 7, 6,
                                4, 5, 9, 8,    5, 6, 10, 9,   6, 7, 11, 10,
                                8, 9, 13, 12,  9, 10, 14, 13, 10, 11, 15, 14 };
float points[] = { -3, 0, -3,  -1, 0, -3,  1, 0, -3,  3, 0, -3,
```

```
                                    -3, 0, -1,  -1, 0, -1,  1, 0, -1,  3, 0, -1,
                                    -3, 0, 1,   -1, 0, 1,   1, 0, 1,   3, 0, 1,
                                    -3, 0, 3,   -1, 0, 3,   1, 0, 3,   3, 0, 3 };
    r->Parameter ("vertex point P", (float *)&points);
    r->Mesh ("catmull-clark", 9, nverts, verts);
```

1.8.3 Point and Curve Primitives

`RendererAPI::`**Points** `(int npoints)`

`Points` primitives are for particle systems. *npoints* is the number of distinct point particles.

Pending parameters (set by `Parameter`) give control vertices and primitive variables to be interpolated across the primitive. The parameters must include 3D positions of the points (passed as the `"P"` parameter).

Primitive variables with interpolation type `vertex` require *npoints* values; primitive variables without an interpolation type require a single data value which does not vary from point to point.

If the primitive variable list contains a floating-point variable named `"width"`, the values will be used to determine the diameter of the point primitives. The width is measured in object space units, and defaults to 1.0 if no value is supplied. The `"width"` parameter may either be `vertex`, specifying the width of each point individually, or no interpolation type to specify a single width for all points in the primitive.

`Points` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

EXAMPLE:

```
RendererAPI *r;

float positions[4][3] = { ... }
float widths[4] = { .01, .02, .01, .04 };
r->Parameter ("P", positions);
r->Parameter ("vertex float width", widths);
r->Points (4);
```

`RendererAPI::`**Curves** `(const char *interp, int ncurves, int vertspercurve)`
`RendererAPI::`**Curves** `(int ncurves, int vertspercurve, int order, const float *knot, float vmin, float vmax)`

`Curves` primitives look like thin tubes or ribbons and are very inexpensive to render, making them ideal for hair, fur, grass, struts seen from far away, etc.

This primitive draws *ncurves* individual curves, each of which is formed by *vertspercurve* vertices.

In the first, simpler form, the curves are piecewise linear or piecewise cubic, and are parameterized uniformly on [0,1]. The *interp* parameter describes the means of interpolating `vertex` variables, including `"P"`, along each individual curve. An *interp* value of `"linear"` indicates piecewise linear curves, whereas `"bezier"`, `"bspline"`, or `"catmull-rom"` indicate a piecewise-cubic interpolation of the specified basis. The *vertspercurve* must be an appropriate number for the particular interpolation type (i.e., >= 2 for linear, >= 4 for cubic, and 4 + 3 *i* for Bezier).

The second form allows complete specification of arbitrary order and knot vector, much like `Patch`. The length of *knot* must be *vertspercurve* + *order*. Each individual curvelet follows a NURBS curve defined over the parametric range [*vmin* ... *vmax*].

Note: *NURBS Curves are not implemented*

Pending parameters (set by `Parameter`) give control vertices and primitive variables to be interpolated across the curves. The parameters must include either 3D control vertices (passed as the `"P"` parameter), or 4D `hpoint` control vertices (`"Pw"`) to indicate rational curves.

The number of values required for any `vertex` primitive variable is `ncurves` * `vertspercurve`. Primitive variables with interpolation type `linear` require 2 * `ncurves` values, one for each curve endpoint, and will be linearly interpolated along the length of each curve. `perpiece` primitive variables require `ncurves` values, one for each curve, and will be constant along each individual curve. Primitive variables without interpolation types, as usual, require a single value that will be used for all the curves in the primitive.

`Curves` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

An optional floating-point `"width"` parameter specifies the diameter of each curve in object space, and may be `perpiece` (one width value for each individual curve), `linear` (two width values for each individual curve, varying linearly along the length of the curve), `vertex` (same number of values and interpolation method as the vertex positions), or without an interpolation type (if all individual curves have the same diameter). If no `"width"` parameter is supplied, the diameter of all curves will be 1.0.

The individual curves, though actually shaped like ribbons, always face toward the viewing position (camera or ray) so that they subtend the full width, thus looking as if they were a thin tube rather than a ribbon. However, if you supply a `linear` or `vertex "N"` parameter, the curves will appear as ribbons whose orientation is fixed to be perpendicular to the normals supplied. Keep in mind that the primitive is designed for very thin (possibly subpixel) tubes or ribbons. Curves with exceptionally large widths may no longer look as good, due to the kinds of approximations used to render large numbers of them efficiently.

EXAMPLE:

```
// Make two 4-point Bezier curve segments that taper from a width
// of 0.1 at the base to 0.05 at the tip.
float positions[8][3] = { ... }
float widths[4] = { .1, 0.05, 0.1, 0.05 };
r->Parameter ("vertex point P", &positions);
r->Parameter ("linear float width", &widths);
r->Curves ("bezier", 2, 4);
```

1.8.4 Quadrics

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

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

Pending parameters (set by `Parameter`) give primitive variables to be interpolated across the primitive. Primitive variables with interpolation type `linear` require four values; primitive variables without an interpolation type require a single data value which does not vary across the patch.

The quadrics are the only geometric primitive that are specified without control vertices (`"P"` or `"Pw"`). Because of this, `vertex` primitive variables are not accepted.

`RendererAPI`::**Cone** `(float height, float radius, float thetamax)`

Creates a cone with an open base on the *x*-*y* plane and apex at (0, 0, *height*).

`Cone` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

EXAMPLE:

```
r->Cone (3.0, 1.0, 360.0)
```

`RendererAPI::`**Cylinder** `(float radius, float zmin, float zmax, float thetamax)`

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

`Cylinder` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

EXAMPLE:

```
r->Cylinder (1.0, -0.5, 1.2, 360.0)
```

`RendererAPI::`**Disk** `(float height, float radius, float thetamax)`

Creates a disk parallel to the *x*-*y* plane with *z* = *height*.

`Disk` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

EXAMPLE:

```
r->Disk(0.0, 2.0, 360.0)
```

`RendererAPI::`**Hyperboloid** `(float x1, float y1, float z1, float x2, float y2, float z2, float th`

Create a hyperboloid by sweeping the line segment joining points (*x1*, *y1*, *z1*) and (*x2*, *y2*, *z2*) about the *z*-axis with the given sweep angle *thetamax*.

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

`Hyperboloid` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

EXAMPLE:

```
r->Hyperboloid(1.0, 0.0, 0.0, 0.0, 0.5, 2.0, 360)
```

`RendererAPI::`**Paraboloid** `(float topradius, float zmin, float zmax, float thetamax)`

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

`Paraboloid` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

EXAMPLE:

```
r->Paraboloid(3.0, 0.0, 6.0, 360.0)
```

`RendererAPI::`**Sphere** `(float radius, float zmin, float zmax, float thetamax)`

Creates a sphere with the given *radius*, centered at the origin of the local coordinate space. The *zmin* and *zmax* parameters can cut off the top and bottom of the sphere if they are not equal to +- *radius*.

If *thetamax* < 0 or *zmax* < *zmin*, the sphere is turned "inside-out" in the expected way.

`Sphere` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

EXAMPLE:

```
r->Sphere (1.0, -1.0, 1.0, 360.0);
```

`RendererAPI::`**Torus** `(float majorradius, float minorradius, float phimin, float phimax, float t`

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

`Torus` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

EXAMPLE:

```
r->Torus(3.0, 0.5, 0.0, 360.0, 360.0)
```

1.8.5 Custom Geometric Primitives

`RendererAPI::`**Shape** `(const char* name)`

Causes the renderer to create a geometric primitive using a Shape DSO/DLL. *name* is presumed to be the name of a shared library (DSO or DLL) in the "plugin" path (see Section 4.3.6).

`Shape` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

EXAMPLE:

```
r->Parameter ("float radius", 1.0);
r->Shape ("sphere");
```

1.9 Constructive Solid Geometry

All of the previously described geometric primitives can be used to define a solid by bracketing a collection of surfaces with `SolidBegin` and `SolidEnd`. This is often referred to as the *boundary representation* of a solid.

When specifying a volume it is important that boundary surfaces completely enclose the interior. Normally it will take several surfaces to completely enclose a volume since, except for the sphere, the torus, and potentially a periodic patch or patch mesh, none of the geometric primitives used by the rendering interface completely enclose a volume. A set of surfaces that are closed and non-self-intersecting unambiguously defines a volume. However, XRT performs no explicitchecking to ensure that these conditions are met.

The inside of the volume is the region or set of regions that have finite volume; the region with infinite volume is considered outside the solid. For consistency the normals of a solid *must* always point outwards.
`RendererAPI::`**SolidBegin** `(const char* operation)`

Starts the definition of a solid. *operation* may be one of the following tokens: `"primitive"`, `"intersection"`, `"union"`, `"difference"`.

Intersection and union operations form the set intersection and union of the specified solids. Difference operations require at least 2 parameter solids and subtract the last *n-1* solids from the first(where *n* is the number of parameter solids). When the innermost solid block is a "primitive" block, no other `SolidBegin` calls are legal. When the innermost solid block uses any other operation, no geometric primitives are legal.

`RendererAPI`::**SolidEnd** `()`

Terminates the definition of the solid.

A single solid sphere can be created using

```
r->SolidBegin("primitive");
r->Sphere(1.0, -1.0, 1.0, 360.0);
r->SolidEnd();
```

Note that if the same sphere is defined outside of a `SolidBegin - SolidEnd block`, it is not treated as a volume-containing solid.

A solid hemisphere can be created with

```
r->SolidBegin("primitive");
r->Sphere(1.0, 0.0, 1.0, 360.0);
r->Disk(0.0, 1.0, -360.0);
r->SolidEnd();
```

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

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

```
r->SolidBegin("difference");

r->SolidBegin("primitive");
r->Sphere(1.0, -1.0, 1.0, 360.0);
r->SolidEnd();

r->SolidBegin("primitive");
r->Disk(2.0, 0.5, 360.0);
r->Cylinder(0.5, -2.0, 2.0, 360.0);
r->Disk(-2.0, 0.5, -360.0);
r->SolidEnd();

r->SolidEnd();
```

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

Attributes may be changed freely inside solids. Each section of a solid's surface can have a different surface shader and color.

1.10 Object Instancing

A list of geometric primitives may be retained by enclosing them with `ObjectBegin` and `ObjectEnd`.

Transformations, and even Motion blocks, may be used inside an Object block, though they obviously imply a relative transformation to the coordinate system active when the Object is instanced. All of an object's attributes are

inherited at the time it is instanced, not at the time at which it is created, unless they have been already set within the Object block.

`RendererAPI::`**ObjectBegin** `(const char *name)`

Starts the definition of a named object. The *name* is used to reference the definition. If *name* has been used to define a previous object, that object is replaced by the new definition.

`RendererAPI::`**ObjectEnd** `()`

Ends the definition of the current object.

`RendererAPI::`**ObjectInstance** `(const char *name)`

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

EXAMPLE:

```
r->ObjectBegin("teapot");
...
r->ObjectEnd();
...
r->ObjectInstance("teapot");
```

1.11 Procedural Geometry Generators and Scene Files

`RendererAPI::`**Input** `(RendererAPI::Generator *procedure, const float *boundingbox=NULL)`

Submits to the renderer an already-created object, derived from the `RendererAPI::Generator` class:

```
class Generator
{
public:
        Generator () { }
        virtual void ~Generator () = 0;
        virtual bool bound (float *bbox) { return false; }
        virtual void run (RendererAPI *rend, const char *params) = 0;
};
```

If boundingbox is `NULL`, the procedure's `bound()` routine will be called, which will either write a bound into `bbox[0..5]` and return `true`, or will return `false` without setting a bound. The bound is an axis-aligned bounding box (6 floats: xmin, xmax, ymin, ymax, zmin, zmax) in the local coordinate system.

The generator's `run` method will be invoked if and when the renderer needs the contents of the bounds. If no bounds are supplied to `Input` and also the generator's `bound()` function returns `false`, no bounds are available and so the generator will be invoked immediately.

The generator must be dynamically allocated with `new`, because *XRT* is going to `delete` it when the renderer no longer needs it. Once you pass a particular Generator object to `Input()`, it belongs to the renderer, and the caller is no longer responsible for deleting it.

`RendererAPI::`**Input** `(const char *name)`
`RendererAPI::`**Input** `(const char *name, const float *boundingbox)`

Causes the renderer to read commands from a named source, which may be a scene file, a Generator DSO/DLL, the output of a program or shell command, or simply the string itself. Note that scene file

readers and generators are really the same thing, since scene files merely invoke generators whose job is to read that file format.

The first word (up to a space) of *name* is presumed to be the name of a shared library (DSO or DLL) in the "generator" path (see Section 4.3.6). The library will be loaded and the following function in the DSO, with C linkage, will be called to construct and return a `Generator` object:

```
extern "C"
{
        RendererAPI::Generator *create (const char *command);
}
```

The Generator returned is then is handled as if were given directly to the renderer, with the remainder of *name* (after the first word) supplied as the parameter to the Generator's `run` method.

If name is a scene file in the "input" path (see Section *Search Paths*), then that file name is used as the sole argument to a Generator whose name is the format of the file. The format is presumed to be the file extension (i.e., the characters in the filename following the last period). In other words,

```
Input("teapot.pyg")
```

is equivalent to

```
Input("pyg teapot.pyg")
```

EXAMPLES:

```
// Read commands from pyg file, immediately
r->Input ("teapot.pyg");

// Note that the above is equivalent to:
r->Input ("pyg teapot.pyg");
```

1.12 Error Management

Errors may occur during the execution of *XRT* API calls. These may be caused by incorrect input (such as inconsistent or invalid data passed to the API routine), system errors (such as not finding a requested texture on disk), or for other reasons. The default behavior is to print error messages to `stderr`, or to log them to a file (specified by `Attribute ("string error:filename")`.

Client applications that need custom error handling may provide their own error manager and/or handler to the renderer. An *error manager* is a class declared in `errormanager.h`. You cannot subclass it, but you can create an `ErrorManager` for each `RendererAPI` renderer, or create just one `ErrorManager` and share it among multiple renderers. Think of it as a wrapper for an error handler. The error handler as a "functor" (a class that behaves like a function) that is really just a callback function for processing an error or warning. You can create your own error handler functor from scratch, which can receive the callbacks and perform whatever action you prefer.

The basic sequence is to subclass `ErrorHandler` to create a low-level handler with the desired behavior, create an `ErrorManager` wrapping the handler, then pass the `ErrorManager` to `CreateRenderer`.

These classes and methods are all defined in `errormanager.h` and are within namespace `xrt`.

ErrorHandler class

```
class PUBLIC ErrorHandler
{
public:
    virtual ~ErrorHandler () {}
```

```
    virtual void operator() (ErrCode errcode, const char *msg);
};
```

This minimal class definition is a simple callback for error messages. Alternative handlers may be created by subclassing `ErrorHandler` and replacing the virtual `operator()` (and optionally, the virtual destructor, if you add data members that need to be properly destroyed).

When the callback is made, the *errcode* is an error code describing the type of error, and *msg* is the actual error message. Valid error codes currently consist of: `INFO` (information only), `WARNING` (warning, may not really be wrong), `ERROR` (probably something wrong, but renderer will attempt to recover), `SEVERE` (severe, probably unrecoverable, error), `MESSAGE` (message, no error, usually for debugging).

EXAMPLE:

```
using namespace xrt;

class TrivialHandler : public ErrorHandler
{
public:
    virtual void operator() (ErrCode code, const char *msg)
    {
        std::cerr << "Err " << (int)code << ": \"" << msg << "\"\n";
    }
};
```

ErrorManager methods

`static ErrorManager *`
`ErrorManager::`**Create** `(ErrorHandler *handler=NULL, int verbosity=VERBOSITY_NORMAL)`

Creates an `ErrorManager` that wraps the given `ErrorHandler`, using the specified verbosity level. If *handler* is NULL, a default handler will be created.

The *verbosity* may take on one the values: `VERBOSITY_QUIET`, the handler will only be passed errors; `VERBOSITY_NORMAL`, the handler will be passed errors and warnings; `VERBOSITY_INFO`, the handler will be passed errors, warnings, and also various informational messages.

`int ErrorManager::`**Verbosity** (void) const‘‘

Retrieves the current verbosity of the error manager, one of: `VERBOSITY_QUIET`, `VERBOSITY_NORMAL`, `VERBOSITY_INFO`.

`void ErrorManager::`**Verbosity** `(int verbosity)`

Changes the verbosity of the error manager. Valid values for *verbosity* are `VERBOSITY_QUIET`, `VERBOSITY_NORMAL`, or `VERBOSITY_INFO`.

`ErrorHandler * ErrorManager::`**Handler** `(void) const`

Returns a pointer to the error handler currently associated with the error manager.

`void ErrorManager::`**Handler** `(ErrorHandler *handler)`

Changes the error handler associated with this error manager. This merely replaces the error manager's pointer to the error handler, but does not free the previous error handler.

`void ErrorManager::`**Info** `(const char *format, ...)`
`void ErrorManager::`**Warning** `(const char *format, ...)`
`void ErrorManager::`**Error** `(const char *format, ...)`
`void ErrorManager::`**Severe** `(const char *format, ...)`
`void ErrorManager::`**Message** `(const char *format, ...)`

These are the `ErrorManager` routines that accept errors at various levels. The verbosity setting determines which will be supressed and which will be passed on to the handler.

RendererAPI methods

`static RendererAPI *`
`RendererAPI::`**CreateRenderer** `(const char *type=NULL, ErrorManager *err=NULL)`

When a non-NULL pointer to an `ErrorManager` is passed to `CreateRenderer`, that error manager will used by the renderer.

`ErrorManager & RendererAPI::`**Err** `(void)`

Returns a reference to the renderer's `ErrorManager`. This reference allows you to manipulate the renderer's error manager, for example, by replacing its error handler, changing its verbosity level, or even directly issuing `Error` or other calls that will go through the error handler.

Error Manager/Handler Example

```
using namespace xrt;

class TrivialHandler : public ErrorHandler
{
public:
    virtual void operator() (ErrCode code, const char *msg)
    {
        std::cerr << "Err " << (int)code << ": \"" << msg << "\"\n";
    }
};


TrivialHandler *myhandler = new TrivialHandler;
ErrorManager *mymanager = new ErrorManager (myhandler);
RendererAPI *rend = CreateRenderer (NULL, mymanager);
...
delete rend;
delete mymanager;
delete myhandler;
```

1.13 Example Scene Specification

```
// Make a NURBS cylinder
void
make_nurbs_cylinder (|XRT|API *r)
{
    float uknot[] = { 0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 4 };
    float vknot[] = { 0, 0, 1, 1 };
    float Pw[] = { 1, 0, 0, 1,   1, 1, 0, 1,   0, 2, 0, 2,  -1, 1, 0, 1,
                  -1, 0, 0, 1,  -1, -1, 0, 1,  0, -2, 0, 2,  1, -1, 0, 1,
                   1, 0, 0, 1,   1, 0, -3, 1,  1, 1, -3, 1,  0, 2, -6, 2,
                  -1, 1, -3, 1, -1, 0, -3, 1, -1, -1, -3, 1, 0, -2, -6, 2,
                   1, -1, -3, 1,  1, 0, -3, 1 };

    r->Parameter ("vertex hpoint Pw", Pw);
    r->Patch (9, 3, uknot, 0, 4, 2, 2, vknot, 0, 1);
}

int
main (int argc, char **argv)
```

```
{
    // Create a renderer object
    |XRT|API *r = |XRT|API::CreateRenderer ();

    // Check if we created the renderer okay. This might have failed,
    // for instance, if a license wasn't found.
    if (r == NULL) {
        fprintf (stderr, "Couldn't create renderer! Exiting...\n");
        exit (1);
    }

    // Set camera and image parameters
    r->Attribute ("string projection", "perspective");
    float shutter[2] = { 0, 1 };
    r->Attribute ("float[2] shutter", &shutter);
    int res[2] = { 640, 480 };
    r->Attribute ("int[2] resolution", &res);
    r->Attribute ("float fov", 45);

    // Specify an output image
    r->Output ("test.tiff", "tiff", "rgba", "camera");

    r->World ();    // Signal the end of the camera section

    // Set an ambient light
    r->Parameter ("float intensity", 0.01);
    r->Light ("amb1", "ambientlight");

    // Set a point light
    r->PushTransform ();
    r->Translate (2, -1, 11);
    r->Light ("pt1", "pointlight");
    r->PopTransform ();

    // Set a color attribute
    float C[3] = { 0, 1, 0 };
    r->Attribute ("color C", &C);

    // Set specular-highlight color
    float specCol[3] = { 1, 1, 1 };
    r->Parameter ("color specularcolor", &specCol);
    r->Shader ("surface", "plastic");

    // Motion-blurred transformation
    r->Motion (2, 0.0, 1.0);
    r->Translate (-0.1, -1 ,12);   // corresponding to time 0
    r->Translate (0.1, -1 ,12);    // corresponding to time 1

    r->Rotate (50, 1, 0, 0);
    make_nurbs_cylinder (r);

    r->Render ( "camera" );

    delete r;
}
```

CHAPTER

TWO

PYG: A PYTHON-BASED SCENE FILE FORMAT

Although *XRT* does not dictate one specific scene file format, it does propose and provide a scene-reading plugin for a Python [1]-based scene file format called Pyg (standing for "Python for Gelato"). This chapter describes the use of Python as scene input.

2.1 Motivation

In addition to making direct calls to a renderer through the C++ API documented in Chapter *The XRT C++ Scene API*, it is often useful to store renderer commands and object models in *scene files*. Scene files are very helpful in a variety of circumstances:

- Sometimes it is simply easier to have a program (or human) output an ASCII file of renderer commands, rather than make direct C++ API calls to a renderer.
- You may not want the scene generation program and scene rendering program to be linked together and resident in memory at the same time (for reasons including reduction of RAM necessary, execution safety, or licensing issues).
- By generating parts of a scene once to a file, then having the renderer read the file when needed, you can save the cost of repeatedly generating the models if the frame must be repeated. The savings can be substantial by reusing the generated model for all frames of an animation, if the object does not deform or change over time.
- By storing parts of a scene in an ASCII format, it can be convenient to write filters in scripting languages such as Perl or Python whose function is to transform your scene in some way prior to being given to the renderer. This kind of trickery can make up for many deficiencies in modeling systems.

The *XRT* API does not dictate one specific scene file format. Rather, *XRT* allows user plugins that read scene formats and make corresponding C++ API calls, as documented in Section 12.

However, *XRT* does define a scripted scene file format based on the Python language and includes a reader plugin for this format. This format is named "Pyg", and files in this format should have the extension `.pyg`. Thus, the generator which reads files of this type is `pyg.generator.so`.

Pyg's proceduralism, flexibility, and access to Python's standard library are distinctive and attractive scene-file features. Although it is a relatively slow format to parse, this is perhaps not of critical importance in applications where flexibility is most required. For example, Pyg may be an interesting format for "master" scene-files -its procedural nature can allow sophisticated control over larger subsidiary scene-files (such as geometry) which may be stored in some moreoptimized form. *XRT*'s DSO-based `Input` call can allow all these different file types to coexist transparently.

[1] Python is a widely-available, freely-distributed programming language. See http://www.python.org.

Pyg files are Python programs, with associated bindings to allow the Python scripts to call the RendererAPI class methods. The remainder of this section documents the Pyg binding.

2.2 Basics

Pyg files are Python scripts. The scripts have access to a renderer object called `XRTRenderer`. This object has a method corresponding to each C++ API call. For example, corresponding to the C++ `Attribute` method is:

```
XRTRenderer.Attribute ("float fov", 45)
```

Also defined for each API call is a defined function that does not need to be referenced from `XRTRenderer`, for example,

```
Attribute ("float fov", 45)
```

The above implicitly sets the option for `XRTRenderer`.

Every C++ API call has an identically-named Python API call that performs the same functions, and which generall takes the same arguments in the same order. Exceptions are listed below.

Unlike C, there is never need to use pointer indirection to pass data arguments. Float or string arguments may be passed directly. More complex data (such as points, matrices, or arrays) may be passed as Python *sequences*. The sequences may be tuples (delimited by parenthesis), lists (delimited by brackets []), or any other object that obeys the sequence protocol (such as *xrange*).

For example, the `Attribute` to set `"C"` to a color may be called as:

```
Attribute ("color C", (1, 0.5, 0.5))
```

Because the script is Python, it is also legal to assign the sequence to a variable, then pass the variable as the data argument:

```
pink = (1, 0.5, 0.5)
Attribute ("color C", pink)
```

For methods that take parameters (which in C++ would be passed using the `Parameter` API call), the Python API allows passing parameters on the API call itself, as alternating strings and values, following the required arguments.

For example, in Python, the `Shader` call may be called like this (almost exactly mimicking the C++ convention):

```
Parameter ("float Kd", 0.5)
Parameter ("float Ks", 0.75)
Parameter ("string texturename", "grid.tx")
Shader ("surface", "paintedplastic")
```

or the parameters may be passed as part of the `Shader` call itself:

```
Shader ("surface", "paintedplastic", "float Kd", 0.5, "float Ks", 0.75,
        "string texturename", "grid.tx")
```

2.3 API Calls

Because there is a nearly one-to-one correspondence between the C++ and Python API calls, we provide only brief functional descriptions of each routine, below. We try to point out differences between the C++ and Python entry points, where they exist, and refer the reader to Chapter *The XRT C++ Scene API* for details on the functionality of each routine.

Note: *The Python bindings for the XRT extensions defined in Chapter The XRT C++ Scene API are not yet fully implemented.*

AppendTransform `(matrix m)`

Concatenate a matrix (given as a sequence of 16 numbers) onto the CTM.

Attribute `(string name, object value)`

Set an attribute to the given value. Depending on the attribute being set, the value may be a number, string, or sequence.

Camera `(string name, ...params...)`

Creates or replaces a camera.

Optional camera attributes may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs at the end of the `Camera` call itself.

Command `(string name)`

Invoke a renderer command.

Optional controls for certain commands may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs at the end of the `Command` call itself.

Comment `(string commenttext)`

For a renderer that is creating an command archive file, adds a comment in the appropriate output format. This call is ignored by "live" renderers producing images.

ConnectShaders `(string srclayer, string srcparam, string dstlayer, string dstparam)`

Connect the named parameters of two shader layers together. This call must occur between `ShaderGroupBegin` and `ShaderGroupEnd`.

Curves `(string interp, int ncurves, int nvertspercurve, ...params...)`
Curves `(int ncurves, int nvertspercurve, int order, sequence knot, number vmin, number vmax, ...params...)`

Creates a `Curves` primitive. The knot value is a sequence (such as a sequence).

Interpolated parameters may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs at the end of the Curves call itself.

GetAttribute `(string name)`

Returns the named attribute. Aggregate types, such as points, colors, or arrays, are returned as sequences whose elements are simple numbers or strings. Note that this is somewhat different than the C++ binding for `GetAttribute` (which passes a pointer to store the value, rather than returning it). If the attribute is not found, the return value will be `None`.

Input `(string name)`
Input `(string name, sequence boundingbox)`

Reads commands from a named source, which can either specify a scene file or the name of a Generator DSO/DLL. If the *boundingbox* is specified (as a sequence of six numbers), the input will only occur if and when the renderer needs to know the contents of the bounding box.

Light `(string lightid, string shadername, [string layername,] ...params...)`

Creates or replaces a light source. The layer name is optional.

Shader parameters may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs.

LightSwitch `(string lightid, boolean onoff)`

Turns an existing light on or off for subsequent primitives.

LookAt `(number posx, number posy, number posz, number atx, number aty, number atz, number upx, number upy, number upz)`
LookAt `(vector pos, vector at, vector up)`

Concatenates a viewing transformation matrix onto the current transformation, given a camera position, target, and "up" vector, either specified as 9 numbers (*posx*, *posy*, *posz*, *atx*, *aty*, *atz*, *upx*, *upy*, *upz*), or 3 sequences containing 3 numeric elements.

Mesh `(string interp, int sequence nverts, int sequence verts, ...params...)`

Creates a `Mesh` primitive.

Interpolated parameters may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs at the end of the `Mesh` call itself.

Unlike the C++ `Mesh` call, there is no need to pass the number of faces in the mesh - the *nverts* sequence gives the number of vertices for each face, and Python can discern the number of faces from the length of the sequence. The *verts* parameter is a sequence that contains the vertex indices of all vertices of all faces. The length of sequence *verts* should be the sum of all the elements in the *nverts* sequence.

Modify `(string namepattern)`

Turns on modify mode, so that subsequent changes affect all attribute states whose `"name"` attribute matches the regular expression namepattern.

Motion `(number time0, number time1, ...)`
Motion `(number sequence times)`

Starts a `Motion` block. Note that unlike the C++ binding, you do not need to pass the *number* of time values - Python can discern the number of knots directly by the number of parameters or the length of the sequence.

ObjectBegin `(string name)`

Creates or replaces a named object.

ObjectEnd `()`

Ends the definition of the current object.

ObjectInstance `(string name)`

Creates an instance of a previously named object.

Output `(string name, string format, string dataname, string camera, ...params...)`

Specifies an output image for rendered pixels, for a particular camera.

Optional parameters may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs at the end of the `Output` call itself.

Parameter `(string name, object value)`

Add a pending parameter, to be used by a subsequent call to `Camera`, `Output`, `Shader`, `Light`, or a geometric primitive. Depending on the declaration of the parameter being set, the value may be a number, string, or sequence.

Patch `(string interp, int nu, int nv, ...params...)`
Patch `(int nu, int uorder, number sequence uknot, number umin, number umax, int nv, int vorder, number sequence vknot, number vmin, number vmax, ...params...)`

Creates a patch specified by a rectangular array of control vertices.

Interpolated parameters may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs at the end of the `Patch` call itself.

Points `(int npoints, ...params...)`

Creates a `Points` geometric primitive.

Interpolated parameters may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs at the end of the `Points` call itself.

PopAttributes `()`

Restore the attribute state to the values it had when the corresponding `PushAttributes` call was made.

PopTransform `()`

Restore the transformation state to the values it had when the corresponding `PushTransform` call was made.

PushAttributes `()`

Save the attribute state.

PushTransform `()`

Save the transformation state.

Render `([string camera])`

Render the scene. The camera name is an optional parameter; if not specified, the default camera is rendered.

RestoreAttributes `(string name)`
RestoreAttributes `(string name, string attrs)`

Replaces some or all of the current attribute state with the saved attribute state with the given *name* (set by `SaveAttributes`).

Rotate `(number angle, number x, number y, number z)`
Rotate `(number angle, vector axis)`

Prepend the current transformation with a rotation of *angle* degrees about the axis defined by (*x*, *y*, *z*). The axis may also be defined as a sequence of 3 numeric values.

SaveAttributes `(string name)`
SaveAttributes `(string name, string attrs)`

Create a named alias for part or all of the current attribute state in a global dictionary of name/attribute state pairs. The *name* may be used with `RestoreAttributes`.

Scale `(number x, number y, number z)`
Scale `(vector scale)`

Prepend the current transformation with a scale factor of (*sx*, *sy*, *sz*). The scale values may also be defined as a sequence of 3 numeric values.

SetTransform `(matrix m)`
SetTransform `(string spacename)`

Replace the current transformation, either with the 4x4 matrix supplied as a sequence of 16 numbers, or with the named transformation.

Shader `(string shaderusage, string shadername, [layername,] ...params...)`
Shader `(string shaderusage)`

Sets the shader specified by shadername to be used as the current shader of the given *shaderusage*. The optional *layername* is a string that identifies the shader layer.

Shader parameters may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs.

If only the *shaderusage* parameter is passed, the shader assignments for that usage are cleared.

ShaderGroupBegin `()`

Begin a group of shader layers.

ShaderGroupEnd `()`

End a group of shader layers.

Shape `(string name)`

Creates a custom primitive from a Shape DSO/DLL.

Interpolated parameters may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs at the end of the `Sphere` call itself.

Sphere `(number radius, number zmin, number zmax, number thetamax, ...params...)`

Creates a `Sphere` primitive.

Interpolated parameters may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs at the end of the `Sphere` call itself.

Translate `(number x, number y, number z)`
Translate `(vector translation)`

Prepend a translation onto the CTM, either specified as three numbers (*x*, *y*, *z*), or a sequence containing 3 numeric elements.

TrimCurve `(int sequence ncurves, int sequence n, int sequence order, number sequence knot, int sequence min, int sequence max, number sequence uvw)`

Sets the current trim curve.

Note that unlike the C++ binding, there is no parameter giving the number of loops - it is inferred from the length of the *ncurves* sequence that gives the number of curves for each loop.

World `()`

Marks the end of scene-wide attributes, tags the CTM to be `"world"` space, and marks the beginning of per-object information.

2.4 Example Pyg Scene File

This is a Pyg version of the C++ API example listing in Section *Example Scene Specification*. It features a motion-blurred NURBS cylinder, and two lights.

```
def nurbscyl():
    uknot = (0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 4)
    vknot = (0, 0, 1, 1)
    Pw = ( 1, 0, 0, 1, 1, 1, 0, 1, 0, 2, 0, 2, -1, 1, 0, 1, -1, 0, 0, 1,
          -1, -1, 0, 1, 0, -2, 0, 2, 1, -1, 0, 1, 1, 0, 0, 1, 1, 0, -3, 1, 1,
           1, -3, 1, 0, 2, -6, 2, -1, 1, -3, 1, -1, 0, -3, 1, -1, -1, -3, 1,
           0, -2, -6, 2, 1, -1, -3, 1, 1, 0, -3, 1 )
    Patch (9, 3, uknot, 0, 4, 2, 2, vknot, 0, 1, "vertex hpoint Pw", Pw)
```

```
Attribute ("string projection", "perspective")
Attribute ("float[2] shutter", (0, 1))
Attribute ("int[2] resolution", (640, 480))
Attribute ("float fov", 45)

Output ("test.tif", "tiff", "rgba", "camera")

World ()
Light ("amb1", "ambientlight", "float intensity", 0.1)

PushTransform ()
Translate (1, 0, 9)
Light ("pt1", "pointlight", "float intensity", 1.0)
PopTransform ()

Attribute ("color C", (1, 0.5, 0.5))
Shader ("surface", "plastic", "float Ks", 0.9, "float Kd", 1)

Motion (0.0, 1.0)
Translate (1, 0, 12)
Translate (1, 0, 13)

Rotate (50, 1, 0, 0)
nurbscyl ()

Render ()
```

Assuming that the above code was in a file called `cyl.pyg`, it could be rendered with the following command:

```
xrt cyl.pyg
```

2.5 Calling xrt from Python

In addition to naming a Pyg file to render on the xrt command line, it is also possible to run an ordinary Python script that makes *XRT* API calls. The necessary steps to do this are:

1. The environment variable `$PYTHONPATH` needs to point to the location of the `xrt` module, which is stored in `$XRT_HOME/lib`.

   For Windows:

   ```
   set PYTHONPATH=$XRT_HOME/lib;$PYTHONPATH
   ```

2. The Python file needs to:

   ```
   import xrt
   ```

CHAPTER

THREE

XRT ATTRIBUTES AND COMMANDS

This chapter documents all of the attributes recognized by *XRT* for use with the `Camera`, `Output`, and `Attribute` API functions.

3.1 Camera Attributes

These attributes set camera and image properties, and are normally set as optional parameters to the `Camera` command. They may also be set by `Attribute`, but in that case apply to the next camera that will be declared.

`"string projection"` [*projectionname*]

Sets the type of projection used by the camera. *XRT* recognizes the projections `"perspective"` and "orthographic". The default is `"perspective"`.

`"float fov"` [*angle*]

Sets the vertical field of view used by the camera. This parameter only has an effect if the projection is `"perspective"`. The angle is measured in degrees. If not set, the default (for a perspective camera) is 90.

`"float[4] screen"` [*xmin xmax ymin ymax*]

Specifies the region of `"screen"` space (points projected onto the $z = 1$ plane in camera coordinates) that is mapped to the image area. The $x = xmin$ line in `"screen"` space corresponds to the left edge of the raster image, $x = xmax$ to the right edge, $y = ymin$ to the lower edge, and $y = ymax$ to the upper edge.

If this attribute is not set, the default values are: (- *frameaspectratio*, *frameaspectratio*, -1, 1)

(The frame aspect ratio is defined as *xresolution*/ *yresolution*.)

In other words, the default behavior is that the image is centered around the z axis, with the y dimension running from -1 to 1, and the x dimension determined by the frame aspect ratio.

`"int[2] resolution"` [*xres yres*]

Sets the full resolution (in pixels) of the image to be rendered. The values are integers giving the horizontal and vertical resolution, respectively. The default is [640 480].

`"float pixelaspect"` [*ratio*]

Specifies the aspect ratio (width/height) of the pixels. The default value, 1.0, indicates square pixels.

`"float[4] crop"` [*xmin xmax ymin ymax*]

Designates a subregion of the image pixels to be rendered, bounded by *xmin*, *xmax* horizontally and *ymin*, *ymax* vertically. The *xmin*, *xmax*, *ymin*, *ymax* arguments are floating point numbers, expressed in "NDC"

coordinates (that is, with the origin in the upper left corner and with coordinates ranging from 0 to 1 across and down the image, respectively). The default is for the entire image to be rendered (*0 1 0 1*).

The pixels output will range in *x* from `ceil(xres*crop[0])` to `ceil(xres*crop[1]-1)`, and in *y* from `ceil(yres*crop[2])` to `ceil(yres*crop[3]-1)`, inclusive. The values are clamped to the range `[0, xres-1]` and `[0, yres-1]`. Geometry outside the region may be processed and rendered as necessary to ensure that adjacent nonoverlapping crop windows will exactly match up, including properly filtering across the boundary.

`"float near"` [*n*]
`"float far"` [*f*]

Sets the near and far clipping planes. Geometry whose *z* coordinate in camera space is less than *near* or greater than *far* will not be visible. There are also some computations in which `"camera"` space *z* values are normalized using the clip plane values (for example, `"screen"` space *z* or the return value of the shading language `depth()` function). The default is near = `0.1`, far = `1.0e6`.

`"float[2] shutter"` [*open close*]

Specifies the time range in which the camera's shutter is open, allowing moving objects to form a blurred image. Unlike a real camera, longer shutter times will not increase the amount of light exposure or change the brightness of the image. If *open* = *close*, the scene will be rendered with no motion blur. The default, if no `"shutter"` attribute is given, is for the scene to be rendered at time 0.0 with no motion blur.

`"float fstop"` [*fstop*]
`"float focallength"` [*focallength*]
`"float focaldistance"` [*focaldistance*]

Collectively, these attributes specify the parameters of the camera that lead to "depth of field" effects, which simulates a camera lens with a particular focal length and f/stop, focused on objects at a given distance. The *focallength* and *focaldistance* parameters are measured in units of `"camera"` space. If *fstop* is `1e30` (effectively infinity), a pinhole camera will be used, resulting in a perfectly sharp image at all distances (this is the default behavior if no `"fstop"` attribute is specified).

Consider the Pyg example:

```
Camera ("maincam", "fstop", 8, "focallength", 4, "focaldistance", 200)
```

If the scene was modeled such that `"camera"` space had units of centimeters, the command above sets up an f/8, 40mm lens focused on objects 200 cm from the camera.

`"string sampler"` [*samplername*]

Sets the antialiasing strategy used by the camera. Valid arguments are:

`"uniform"`

`"stratified"`

`"int[2] spatialquality"` [*x y*]

Sets the quality level of the spatial antialiasing for geometric edges to a minimum of *x* * *y* subregions per pixel. The default is (`2, 2`).

`"int temporalquality"` [*n*]

Sets the quality level of the temporal antialiasing (motion blur) to a minimum of n different time values sampled. The default is 16 temporal samples.

`"int dofquality"` [*n*]

Sets the quality level of the depth of field to a minimum of n different lens values sampled. The default is 16 different lens samples.

`"int[2] limits:bucketsize"` [*x y*]

Sets the size (in x and y pixels) of the screen buckets that represent units of work for the renderer. The default is (32, 32). Ordinarily, there should be no reason to override the default, but advanced users may wish to tune performance on problematic scenes by adjusting this attribute.

`"string bucketorder"` [*direction*]

Sets the traversal order in which buckets are rendered on screen. Valid arguments are:

`"horizontal"`

Top to bottom, and within each row, alternatively, left to right and right to left (this is the default).

`"vertical"`

Left to right, and within each column, alternatively, top to bottom and bottom to top. For some large scenes with a wide-screen aspect ratio, using `"vertical"` bucket order may render the scene faster and require much less memory.

`"spiral"`

Start at the center of the screen and proceed outward in a spiral pattern.

3.2 Output Attributes

This section describes output image attributes, which may be set per output image as optional parameters to the `Output` command. They may also be set by `Attribute`, but in that case apply to the next output that will be declared.

`"string filter"` [*filtername*]

`"float[2] filterwidth"` [*xw yw*]

Final image pixels are produced by taking a weighted average of the contribution of nearby subregions, including those from other pixels. The weights are determined by a pixel filter. The default is to use a `"gaussian"` filter with width 2 in each direction. This should be adequate for most images, but you can override with a custom filter and width (which may be different for each `Output` specified).

The filter shape may be specified by a combination of the `"filter"` parameter, which takes a string giving the name of the filter, and `"filterwidth"`, which takes an array of two floats specifying the x and y support widths of the filter.

For ordinary data (color, etc.), the filters supported are: `"gaussian"`, `"box"`, `"triangle"`, `"catmull-rom"`, `"sinc"`, `"blackman-harris"`, `"mitchell"`, and `"b-spline"`. For depth (z) data only, you may use the filters `"min"`, `"max"`, or `"average"`.

Note that `"gaussian"`, `"mitchell"`, `"box"`, `"triangle"`, `"b-spline"`, and `"blackman-harris"` actually get "wider" (and thus blurrier) as the filter width increase. The `"sinc"` and `"catmull-rom"` filters use the width to "window" the existing function, without changing the shape.

For certain filters, only particular widths make sense - the `"catmull-rom"` filter should always have width 4, and `"sinc"` should have a whole-number width (4 is a good value).

`"float gamma"` [*gamma*]
`"float gain"` [*gain*]

Before passing pixels to the image display driver (and thus prior to any quantization), the renderer transforms all `color` data according to following formula:

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

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

`"int[4] quantize"` [*zero one min max*]
`"float dither"` [*ditheramplitude*]

All renderer data is computed and sent to the image driver as floating-point data. For those image formats that must store pixel values as integers, the image driver is expected to perform the conversion according to the following formula:

pixelval = round(*zero* +(*one* - *zero*) * *floatval* + *ditheramplitude* * random())
pixelval = clamp(*pixelval*, *min*, *max*)

This has the effect of scaling the values so that a value of 0.0 gets an integer output value of *zero* and a value of 1.0 gets an integer output value of *one*, a pseudo-random dither of amplitude *ditheramplitude* is added to eliminate banding artifacts, and the integer value is clamped to lie between *min* and *max*.

The values *zero*, *one*, *min*, and *max* are specified, respectively, as an array of 4 ints passed as the `"quantize"` parameter to `Output`. The *ditheramplitude* value is passed as the float argument to the `"dither"` parameter.

It is expected that image drivers honor the convention of using the range of *min* and *max* to determine the bit depth of the resulting output image: if both are <= 255, 8-bit integer (per channel); if both are <= 65535, 16-bit integer; otherwise 32-bit integer. If all four numeric parameters are 0, then no integer quantization is performed and a floating-point image is output. If *ditheramplitude* is 0 (as it should be for floating-point images), no dithering is performed. If the behavior of any particular image driver deviates from this convention, it should carefully document its behavior.

The default quantization is `(0, 255, 0, 255)` and the default dither is `0.5`, meaning that output will be 8 bits per channel.

3.3 Scene-wide Attributes

The attributes described in this section apply to the entire frame being rendered. They should be set only before the `World` call, since they cannot be changed for each object.

3.3.1 Ray Tracing and Global Illumination

`"int ray:maxdepth"` [*d*]

Sets the maximum ray tracing recursion depth for rays spawned by `environment()` calls in shaders. A value of 0 means that no rays will ever be traced by `environment()`, even if the shaders request it; 1 means that you can see reflections, but not reflections of reflections; 2 means that you can see reflections of reflections, but not reflections of reflections of reflections; and so on. The default value is 2.

`"color ray:background"` [*c*]

Sets the color for rays that hit nothing. The default is black (0,0,0).

3.3.2 Search Paths

Various external files may be needed as the renderer is running, and unless they are specified as fully-qualified file paths, the renderer will need to search through directories to find those files. There exist attributes to set the directories in which to search for these files.

`"string path:input"` [*pathlist*]

`"string path:texture"` [*pathlist*]

`"string path:shader"` [*pathlist*]

`"string path:generator"` [*pathlist*]

`"string path:imageio"` [*pathlist*]

Sets the search path that the renderer will use for files that are needed at runtime. The different search paths recognized by *XRT* are:

`"input"`

scene files for `Input` calls, defaults to `".:$XRT_HOME/inputs"`.

`"texture"`

texture, shadow, and environment maps, defaults to `".:$XRT_HOME/textures"`.

`"shader"`

compiled shaders, defaults to `".:$XRT_HOME/shaders"`.

`"generator"`

DSO's/DLL's for `Input` calls, defaults to `".:$XRT_HOME/plugins"`.

`"imageio"`

DSO's/DLL's for custom image format input/output plugins, defaults to `".:$XRT_HOME/plugins"`.

Search path types in *XRT* are specified as colon-separated lists of directory names (much like an execution path for shell commands). There are two special strings that have special meaning in *XRT*'s search paths:

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

`$VAR`, `${VAR}`, `$(VAR)`, and `%VAR%` are replaced by the value of environment variable `VAR`, if it exists (for any environment variable).

For example, you may set your generator path as follows (using Pyg):

```
Attribute ("string path:generator", "$HOME/lib/$ARCH:&")
```

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

3.4 Per-object Attributes

The attributes described in this section apply to individual objects, and may be set at any time (before or after `World`). They may be changed for each geometric primitive, and may be saved and restored with `PushAttributes`, `PopAttributes`, `SaveAttributes`, and `RestoreAttributes`.

3.4.1 Current Transformation

`"matrix transform"`

When retrieved by `GetAttribute`, this returns the current transformation matrix (CTM) that transforms points from local to world coordinates. When the CTM is motion-blurred, the matrix returned will be the transformation at shutter open time. If the attribute is queried before any camera is declared, the matrix will be for time 0. This attribute may be retrieved with `GetAttribute`, but may not be set by `Attribute`.

3.4.2 Name

`"string name" [""]`

Tells the render a user-chosen name for subsequent geometric objects. This allows the renderer to print the object name when reporting errors.

3.4.3 Geometry Sets

All geometric primitives are in one or more named *geometry sets*. Objects will only be visible to a particular camera if it is present in the geometry set that has the same name as the camera, or if it is present in a geometry set named `"camera"` (meaning visible to all cameras). The semantics of all other geometry sets are user defined - the most common use is to specify a group of primitives to ray-trace against, or a set of geometric primitives that comprises an area light source.

`"string geometryset"` [*setmod*]

This attribute changes the list of active geometry sets – that is, which geometry lists subsequently-declared primitives will be added to. The argument, *setmod*, is a commaseparated list of named geometry sets. A + character leading the name of a geometry set will cause the set to become active. A leading − character will cause the named set to become inactive. No + or − will cause only the named sets to be active, and all others inactive.

EXAMPLES:

```
// Make geometry also appear in the "reflections" set (in addition
// to whichever sets it was in before)
r->Attribute ("geometryset", "+reflections");

// Make geometry NOT appear in the "shadow" set (but still in all other
// sets it was in before)
r->Attribute ("geometryset", "-shadow");

// Make geometry appear only to camera "maincam" (no other sets)
r->Attribute ("geometryset", "maincam");
```

3.4.4 Surface Appearance Attributes

`"color C"` [*rgb*]

Sets the default surface color `C` that will be available in the shader (if it is not overridden by supplying a value on the geometric primitive). The default is [1 1 1], indicating that subsequent surfaces are white.

EXAMPLE:

```
float C[3] = { 1, 0.5, 0.5 };
r->Attribute ("C", &C);
```

`"color opacity"` [*rgb*]

Sets the default surface `opacity` that will be available in the shader (if it is not overridden by supplying a value on the geometric primitive). The default is [1 1 1], indicating that subsequent surfaces are completely opaque.

`"string orientation"` [*orient*]

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

`"outside"`

same as the coordinate system's handedness (default)

`"inside"`

opposite the coordinate system's handedness

`"lh"`

left handed orientation (regardless of CTM handedness)

`"rh"`

right handed orientation (regardless of CTM handedness)

`"reverse"`

change to the opposite of the previous orientation

EXAMPLE:

```
char *orient = "outside";
r->Attribute ("orientation", &orient);
```

`"int twosided"` [*onoff*]

A nonzero value of *onoff* (the default) indicates that subsequent geometry should be vis ible from both sides.

A value of zero for *onoff* indicates that the renderer may discard subsequent backfacing geometry (i.e., those whose normals point away from the camera). For closed, opaque geometry whose surface normals always point to the outside of the object (or whichever side the camera will be on), backfacing geometry can be culled without changing the appearance of the image, and thus may be used as an optimization hint.

3.4.5 Trim Curve Control

`"string trimcurve:sense"` [`"inside"`]

Determines whether trim curves applied to `Patch` primitives will discard the geometry inside the trim region (if the value is `"inside"`, which is the default), or outside the trim region (if the value is `"outside"`).

3.4.6 Ray Tracing Controls

`"int ray:opaqueshadows"` [`0`]

When nonzero, forces subsequent objects to be treated as opaque when computing raytraced shadows. This results in faster ray-traced shadows, since the object need not be shaded to determine if it blocks all light along the shadow ray. If 0 (the default), the shader will be run at the hit point in order to determine the opacity of the object for ray traced shadow intersections.

CHAPTER

FOUR

SHADING LANGUAGE

XRT Shading Langage is an implementation of RenderMan(R) Shading Language as defined in the *RenderMan(R) Specification 3.2*

Note: *The remainder of this chapter is very sketchy and will be fleshed out in future versions*

4.1 Unimplemented features

- Message passing is only partially implemented.
- `match()` is not supported.

4.2 Extensions

RenderMan(R) Specification 3.2 was issued in 2000. Since then, Pixar has extended its shading language at numerous times without releasing a new specification to the public. However, bits and pieces of information are available here and there, for instance, in *3Delight* manual (http::://www.3delight.com).

This section documents extensions supported by *XRT*.

TBD.

Part III

Using *XRT*

CHAPTER

FIVE

RUNNING *XRT*

5.1 `xrt` Command Line Operation

Invoking *XRT* from the command line is done as follows:

`xrt` [*options*] *file1 ... filen*

Specifying multiple files is identical to specifying a single file that is a concatenation of all the files.

5.2 Environment variables

XRT's behavior is influenced by the following *environment variables* of the command shell:

`XRT_HOME`

This should point to the *XRT* installation directory. The default search paths all refer to subdirectories within the installation directory – for example *XRT* will search for certain shaders in `$XRT_HOME/shaders`. Most critically, the scene file format readers (for at least the startup file) must be in `$XRT_HOME/lib`.

CHAPTER

SIX

CAMERAS AND IMAGE OUTPUT

This chapter covers the basic details of how the CG camera is placed in the scene, and the various options that must be set to determine image resolution and framing, camera attributes, image quality, exactly what data are saved, and how you can determine the image file types and other properties.

6.1 The Camera

6.1.1 Positioning the Camera

Objects in the scene are positioned relative to `"world"` space or some other local coordinate system. This is the result of your having translated or rotated those objects to place them in the scene.

The camera also has a certain position and orientation relative to the world. A special coordinate system called `"camera"` space is centered about the camera, with the x axis pointing to the camera's right, the y axis pointing up, and the z axis pointing in the direction that the camera is looking. Note that this is a "left handed" camera coordinate system.

XRT allows two ways to position the camera in the scene:

1. Assume that at the beginning of the frame that you start out in `"world"` space, set the camera position and orientation as you would with any other object (relative to `"world"` space), then make a `Camera` call. Upon hitting the `World` call, the coordinate system will be restored to `"world"` space. For example:

   ```
   # initially start in world space
   SetTransform (...) # Position the camera
   Parameter (...)    # Set camera parameters
   Camera ("main")    # Instance a camera in the current position
   World ()           # Restore CTM to be world space again
   ...
   ```

   There may be multiple cameras in the scene, although at present, *XRT* only will create images from the first `Camera` declared.

2. If *no* `Camera` call is made prior to the `World` call, then it is assumed that that the initial state was `"camera"` space, and that the CTM at the time of the `World` call represents `"world"` space. In other words, a lack of `Camera` statement implies that all the transformations prior to `World` are placing the world relative to the camera. Note that this is the way that certain other API's (including OpenGL and RenderMan) naturally operate, so this mode allows for easy translation in cases where only one camera is needed.

   If there is no `Camera` call in the scene (and thus, the renderer will implicitly create a camera once it hits `World`), all the optional camera parameters are also settable with the ordinary `Attribute` command.

```
# initially we are in camera space
SetTransform (...) # Position the world relative to the camera
Attribute (...)    # Set camera parameters with Attribute
World ()           # Establishes world space
...
```

6.1.2 Camera Projection

A three-dimensional scene is reduced to a two-dimensional image by *projection*. In any projection, all points along a "line of sight" correspond to the same 2D location in the final image. *XRT* supports both *perspective* (lines of sight converging at a point) and *orthographic* (parallel lines of sight) camera projections. Along any line of sight, the closest object to the camera will be the one seen (although if it is partially transparent, you may also see other objects behind it).

The projection is selected using the `"projection"` camera attribute (see Section *Camera Attributes*). By default, cameras use the `"perspective"` projection.

You may set the camera projection by specifying it as a parameter to the `Camera` call, as in this Pyg example:

```
Parameter ("string projection", "orthographic")
Camera ("main")
```

If no `Camera` command is used (implying that the camera was at the initial scene origin), then the projection may be set by an `Attribute` at any point prior to the `World` command:

```
Attribute ("string projection", "orthographic")
...
World ()
```

Perspective projection

Perspective projections are the default camera projection for *XRT*. Perspective projections are similar to the projection used in an ordinary camera (though not exactly - real cameras always have additional distortions, but in the rare cases where this is important, it is usually corrected as an image-processing operation, not during rendering). With a perspective projection, an object will appear bigger when it is closer to the camera.

You will almost always want to use a perspective projection for "final" images from the main camera. Perspective projections should also be used when generating shadow maps for light sources that project light from a single point (like a spotlight).

The perspective projection also responds to the camera parameter `"fov"` which sets the field-of-view angle in degrees. For example, the following Pyg command sets a perspective projection with a 30 degree field of view:

```
Parameter ("string projection", "perspective")
Parameter ("float fov", 30)
Camera ("main")
```

Orthographic projection

Orthographic projections are primarily used for reproducing certain architectural or engineering drawing methods, and for creating shadow maps for "distant" light sources (those whose light emanates in parallel rays). With an orthographic projection, an object will appear the same size no matter what its depth from the camera (see Figure 7.3).

Orthographic projections do not respond to the `"fov"` parameter. The default orthographic projection is almost certainly too small a view, and you will need to adjust the `"screen"` parameter in order to correctly frame your scene for an orthographic view.

6.1.3 Motion Blur

Real cameras have a shutter that stays open for a certain amount of time to expose the film. The longer the shutter stays open, the more moving objects will appear as blurred streaks on the film. This effect is critical to avoiding strobing when rendering frames for an animation. The shutter interval may be set with the `"shutter"` camera attribute, which takes the opening and closing times. For example,:

```
Parameter ("float[2] shutter", (0, 1.0/48))
Camera ("main")
```

instructs the camera to open the shutter at time 0 and close it again 1/48 later.

The units (seconds, frames, etc.) do not matter, but they are expected to be calibrated to the same scale as the times specified by `Motion` for any moving or deforming objects.

Most motion picture film cameras leave the shutter open for approximately half of the interframe time. For example, for a 24 frame-per-second film, the actual shutter interval is typically 1/48 of a second.

For real cameras, the longer the shutter stays open, the more light strikes the film, and therefore the brighter the resulting image will be. This is not true for the synthetic camera - the image will be no brighter or dimmer, no matter what the `"shutter"` attribute specifies.

Moving objects in time

Individual objects may be blurred one of two ways: via *transformation blur* or *deformation blur*. For transformation blur, the object transformations themselves (the position and orientation of the object or part) are changing over time. For deformation blur, the positions of the vertices comprising the object move their position over time.

To explain how a transformation is blurred, consider a simple translation of 3 units in *x*:

```
Translate (3, 0, 0)
```

If, for example, the object should translate by 3 units at time 0 and by 3.5 units at time 1, the previous `Translate` would be replaced by the following *motion block*:

```
Motion (0, 1)
Translate (3, 0, 0)
Translate (3.5, 0, 0)
```

The arguments to the `Motion` function are a series of *n* time values (usually two, though it may be any number). It is then expected that the `Motion` function be followed by *n* API calls of the same name, but differing only in parameter values, each such call corresponding to one of the time values passed to the preceeding `Motion` call.

Any of the transformation routines (`Translate`, `Rotate`, `Scale`, `AppendTransform`, `SetTransform`) may be blurred in this manner.

It is also possible to blur the shape of the geometry itself by using a `Motion` call followed by *n* calls to a geometric function call (such as `Mesh`, `Patch`, etc.). Each geometric call must be the same function and "shape" (e.g., you may not "morph" a `Patch` into a `Sphere`, nor may you change the number of vertices or faces between subsequent `Mesh` calls within a motion block). For example:

```
Motion (0, 1)
Parameter ("vertex point P", (0,0,0, 0,0,1, 1,0,0, 1,0,1))
Patch ("linear", 2, 2)
Parameter ("vertex point P", (0,1,0, 0,0.5,1, 1,0,0, 1,0,1))
Patch ("linear", 2, 2)
```

In the example above, a bilinear batch is deforming over time. Note that the *n* time values passed to `Motion` are followed by *n* full geometric primitives - including both parameters as well as the geometric primitive itself.

Multi-segment motion blur

`Motion` calls usually specify two times, and therefore be followed by two transformation calls or geometric primitives. This results in *linear motion blur*, in which any particular point on the object moves in a straight line over the course of a frame. Considering that each final frame will be seen by viewers for only a fraction of a second, linear motion blur is almost always sufficient.

However, it is sometimes desirable to have an object trace out a more complex path over the course of a frame. As a practical matter, this is only necessary for objects undergoing rapid rotation (think "helicopter rotor") for which linear motion would look obviously wrong. Of course, this is very simple in *XRT*, merely requiring a larger number of times in the motion block. For example:

```
Motion (0, 0.25, 0.5, 0.75, 1)
Rotate (15, 0, 0, 1)
Rotate (30, 0, 0, 1)
Rotate (45, 0, 0, 1)
Rotate (60, 0, 0, 1)
Rotate (75, 0, 0, 1)
Input ("rotor.pyg")
```

6.1.4 Depth of Field

Depth of field refers to the way objects at a particular distance from the camera appear in sharp focus, while objects that are closer or farther away will appear blurred. It is a physical phenomenon caused by the finite aperture of a camera, and other focusing attributes of the lens system.

By default, *XRT* has depth of field turned off, meaning that all objects are in sharp focus, regardless of their depth in the scene. The depth of field effect can be turned on and adjusted with three camera attributes: `"fstop"`, `"focallength"`, and `"focaldistance"`.

The f/stop is the ratio of focal length to lens aperture, much as you would see f/stop settings on a real camera lens - it lets you control the aperture size. The focal length is the distance from the lens opening to the film plane. The focal distance is the depth from the camera at which objects appear in sharp focus. Both the focal length and focal distance are measured in the same units as `"camera"` space.

For example, if you had constructed your scene so that `"camera"` space units were meters, then the following command would specify an f/4 aperture on a 50mm lens, set to focus sharply objects that were 3.6 meters from the camera:

```
Parameter ("float fstop", 4)
Parameter ("float focallength", 0.05)
Parameter ("float focaldistance", 3.6)
Camera ("main")
```

For real cameras, the wider the aperture (i.e., the smaller the f/stop number), the more light enters the camera, and therefore the brighter the resulting image will be. This is not true for the synthetic camera - the image will be no brighter or dimmer, no matter what the depth of field settings.

6.1.5 Clipping

In addition to objects being not visible to the camera if it is too far to the right or left, top or bottom (that is, off-screen), you can also have the camera ignore objects that are too near to, or too far from the camera. This is something that obviously cannot be done with a real camera, but it can be very useful and often comes in handy with the CG camera. Objects are ignored if their `"camera"` space z values are less than the *near* plane, or if their `"camera"` space z values are greater than the *far* plane.

The z clipping planes can be set with the `"near"` and `"far"` camera attributes. For example, to set the hither plane to $z = 0.1$ and the yon plane to $z = 10{,}000$:

```
Parameter ("float near", 0.1)
Parameter ("float far", 10000)
Camera ("main")
```

There is some benefit to attempting to set the clipping carefully. Tightly bounding the depth of interest in your scene can preserve more computational precision in some parts of the rendering process.

6.2 Image Resolution and Framing

6.2.1 Image Resolution

The image resolution refers to the number and shape of the pixels in the final image. The `"resolution"` camera attribute (see Section *Camera Attributes*) can be used to set the *x* and *y* resolution, which are whole numbers that give the size of the final image, in pixels. The `"pixelaspect"` camera attribute describes the ratio of the width to height of an individual pixel (the default value, 1.0, indicates square pixels). For example, to render an image with 640 * 480 square pixels:

```
Parameter ("int[2] resolution", (640, 480))
Parameter ("float pixelaspect", 1)
Camera ("main")
```

The aspect ratio of the frame is determined by the x and y resolution. Therefore, setting the `"resolution"` not only determines the number of pixels in the image, but also the "shape" of the resulting image.

6.2.2 The Screen Window

Once the scene is projected to a 2D plane, only a subset of the plane is actually turned into the image. That subset is called the screen window. This is directly settable by the `"screen"` camera attribute, which by default is

(-*frameaspectratio*, *frameaspectratio*, -1, 1)

where *frameaspectratio* = *xres/yres*.

Since the default is for the screen window to be centered and with the frame aspect ratio, it is usually not necessary to set the `"screen"` camera attribute. However, there are two instances where it is critical: (1) For an orthographic camera, the `"screen"` attribute is almost certainly required to ensure proper framing of the image. (2) the `"screen"` attribute can be used to distort or shear the image by making the screen window's aspect ratio not match the frame aspect ratio, or by using an off-center screen window.

6.2.3 Crop Window

It is often very useful to render a subset of image pixels, particularly if you are debugging or adjusting part of the scene and do not wish to wait for the entire image to rerender every time a tweak is made. This is easily accomplished by setting the `"crop"` camera attribute, which takes *x* minimum and maximum, and *y* minimum and maximum range, expressed as a portion of the total image (i.e., 0-1). For example, to render the upper-right quadrant of the image only:

```
Parameter ("float[4] crop", (0.5, 1, 0, 0.5))
Camera ("main")
```

Even though the crop window is expressed with floating-point numbers, it will be rounded in such a way as to result in a whole number of pixels. *XRT* is careful to round and to filter at the edges so that adjacent crop windows will match up exactly without seams. For example, rendering one image with:

```
Parameter ("float[4] crop", (0, 1, 0, 0.5))
```

and a second image of the same scene with:

```
Parameter ("float[4] crop", (0, 1, 0.5, 1))
```

will *exactly* render all pixels, without repetition and without seams if the two images are assembled together.

6.3 Image Output

Once pixel values are derived by filtering the data from the pixel subregions, an image must be written to disk in some format, or displayed on some device.

6.3.1 Outputs and Channels

Upon rendering, *XRT* produces one or more *image outputs*. You can think of each output as a separate image of the scene (from the same camera). Each output stream may contain different data - for example, one output stream may consist of color and alpha (RGBA), while another output stream may contain z depth information.

Each image output consists of one or more *channels*. A channel is a single "pane" of data, such as red or blue. A greyscale-only image is a 1-channel image, an ordinary color RGB image is a 3-channel image, and an RGBA image is a 4-channel image.

6.3.2 Image I/O Plugins

There are many different types of image file formats or devices on which to display images. *XRT* uses programs called *Image I/O plugins* to handle image output and display [1]. There is one such plugin for each file format or display type. *XRT* comes with image I/O plugins that understand how to write TIFF, OpenEXR, JPEG, and Targa files, and how to display the image on a computer screen (`"iv"`).

The basic way to specify what data goes to which file and in what format is with the `Output` command:

> `Output` (*name, format, dataname, camera, ...params...*)

The *name* is the filename of the file, the *format* is the file format or display device (actually the name of the imageio plugin), *data* is the name of the data to write to the output stream, and *camera* is the name of the camera from which to image this output. The optional *params* (which may be passed beforehand using the `Parameter` call) control various aspects of the image output, including format-specific options.

As an example, to instruct *XRT* to write RGB color data from camera `"main"` to a TIFF file named `"myfile.tif"`:

```
Output ("myfile.tif", "tiff", "rgb", "main")
```

To write an image with RGB and alpha (coverage) as a 4-channel image:

```
Output ("myfile.tif", "tiff", "rgba", "main")
```

To display the image "live" to a framebuffer display using *XRT*'s `iv` display tool:

```
Output ("myfile.tif", "iv", "rgba", "main")
```

[1] Image I/O plugins also handle image *input* into the renderer.

6.3.3 Bit depth, quantization, and dither

XRT computes pixel values with floating-point precision, but not all output formats support floating-point data. Therefore, the Image I/O plugin may need to convert the raw pixel data to an integer (whole number) representation. This process is known as *quantization*.

The `Output` command takes an optional parameter `"quantize"` that gives the quantization mapping. The `"quantize"` parameter takes an array of four integers that specify the *zero* level, *one* level, *min*, and *max* values.

When floating-point numbers are converted to integers, the number of bits per channel is known as the *bit depth*. The bit depth is computed automatically from the *max* quantization value: if $max \leq 255$, an 8-bit file is created; otherwise, if $max \leq 65535$, a 16-bit file is created; otherwise, a floating-point output file is created. Also, if all of *zero*, *one*, *min*, and *max* are 0, floating-point output will be selected.

For example, to write `"myfile.tif"` as 8-bit integers (this is a typical output format, and also the default):

```
Parameter ("int[4] quantize", (0, 255, 0, 255))
Output ("myfile.tif", "tiff", "rgb", "main")
```

If 8 bits per channel are not enough precision for your application, you could generate a 16 bit per channel image with the following command:

```
Parameter ("int[4] quantize", (0, 65535, 0, 65535))
Output ("myfile.tif", "tiff", "rgb", "main")
```

To aid in reducing artifacts that result from the float-to-integer conversion, you can add a random *dither* to the image. This is just noise that helps to soften the edges and reduce objectionable banding in the image. The dither amplitude can be set by `Output` using the optional parameter `"dither"`. The default dither level is 0.5. The main reason to override this default is in the case of floating-point images, which do not need dither and therefore should have their dither set to 0. For example, to output color pixels with full floating-point precision (and no dither):

```
Parameter ("int[4] quantize", (0, 0, 0, 0))
Parameter ("float dither", 0)
Output ("myfile.tif", "tiff", "rgb", "main")
```

6.3.4 Filters

As described later in Section *Antialiasing and Filtering*, selection of a pixel filter can be accomplished by setting the `"filter"` (which takes a string giving the filter name) and `"filterwidth"` (which takes two floats that specify the *x* and *y* support widths of the filter). For example, to use the Catmull-Rom filter with width 3:

```
Parameter ("string filter", "catmull-rom")
Parameter ("float[2] filterwidth", (3, 3))
Output ("myfile.tif", "tiff", "rgb", "main")
```

6.4 *XRT*'s Bundled Image I/O Plugins

As discussed in Section *Image Output*, the `Output` command takes the name of a Image I/O plugin, which is a plugin that actually writes the pixels in a particular format. *XRT* ships with several image I/O plugins (and hence, can write to those formats). Users or third parties may expand the formats by writing DSO's/DLL's. This section describes *XRT*'s bundled image I/O plugin types.

6.4.1 "tiff" plugin

6.4.2 "jpg" plugin

The bundled `"jpg"` image I/O plugin saves files in JPEG format. We generally do not recommend rendering images directly into JPEG format because JPEG is restricted to 8-bit per channel, 3-channel images, and is "lossy."

6.4.3 "png" plugin

6.4.4 "tga" plugin

6.5 Antialiasing and Filtering

Antialiasing refers to the renderer's efforts to correctly capture details smaller than a pixel (including geometric edges) and to give a smooth appearance to the blur that results from motion or depth of field. There are several options that control the basic time versus quality tradeoffs when performing antialiasing, described below.

Edge antialiasing quality

The most basic antialiasing control is the *spatial quality*, which describes the number of subpixel regions (in x and y) comprising each pixel, for example:

```
Attribute ("int[2] spatialquality", (4, 4))
```

Dividing pixels into smaller regions, each of which is solved separately, is important to antialiasing because smaller regions are geometrically simpler (contain fewer objects and edges) and therefore easier to approximate with certain simplifying assumptions. More subpixel regions will yield higher quality, but will take slightly longer to render.

Filtering

The several spatial subpixel regions must be combined to form the final discrete pixels. To do so with high quality, each pixel gets a weighted average of nearby regions (including regions outside the boundaries of the pixel. This process is known as *filtering*. Filtering has two aspects: the shape of the filter (specified by the name of the filtering function), and the width of the region to which the filter is applied. The filtering can be set by the `"filter"` and `"filterwidth"` image attributes. A 2 * 2 Gaussian filter (the default) may be specified by:

```
Parameter ("string filter", "gaussian")
Parameter ("float[2] filterwidth", (2, 2))
Output ("myfile.tif", "tiff", "rgb", "main")
```

Some people find the 2 * 2 Gaussian filter to be overly blurry. If that is the case, you could try a Gaussian filter with thinner width (but we wouldn't recommend using a width of less than 1.5), or you could use a different filter shape. The Catmull-Rom filter has nice edge sharpening properties, and can be specified as:

```
Parameter ("string filter", "catmull-rom")
Parameter ("float[2] filterwidth", (3, 3))
Output (...)
```

On the other hand, if it is important that pixels equally weight all regions and not consider any spatial regions outside the pixel boundary, then you would want to specify the (infamously low quality) box filter:

```
Parameter ("string filter", "box")
Parameter ("float[2] filterwidth", (1, 1))
Output (...)
```

Feel free to experiment with different filter functions and window sizes, to achieve a "look" that is right for your project. The available filters are listed in the formal description of the `"filter"` attribute in Section *Output Attributes*.

CHAPTER

SEVEN

USING SHADERS

7.1 Shader Basics

Shaders are small user-supplied programs that describe materials and lights.

7.1.1 Types and usages of shaders

Every shader has a *shader* type given in the declaration in the shader's source code. The shader type may be one of: `surface`, `displacement`, `volume`, or `light`. Each type of shader may access a certain set of variables, and some shader types have restricted access to function calls or syntactic structures (for example, only light shaders may have emit statements).

At render time, various shaders may be bound to different pieces of geometry for a variety of *shader usages*. Shader usage refers to what functionality the shader is expected to provide, and exactly when in the rendering process it is executed. Shader usages currently include:

- `"displacement"` shaders may move the surface positions or alter their normals to make "dents" or other fine shape changes to an object. Displacements are calculated once, are the first shaders to run on any surface, and are independent of the viewing direction.
- `"surface"` shaders determine the color and opacity of the object, as viewed from a particular direction.
- `"volume"` shaders are run after the surface shader, and are allowed to modify the color and/or opacity in order to account for atmospheric effects along the viewing ray.
- `"light"` shaders are run when surface or volume shaders have `illuminance()` statements (or calls to functions that implicitly run the lights, such as `diffuse()` and `specular()`), and determine how much energy arrives at a point due to a particular light source.

Lights are instantiated with the `Light` API call, whereas the other shader usages are bound to specific pieces of geometry with the `Shader` call. The shader usage must match the shader type declared in the shader's source code.

All objects are required to have a surface shader, and if none is specified in the scene, a `defaultsurface` shader will be used. Objects are not required to have displacement or volume shader, and by default they do not. Objects are also not required to have any lights, though of course if there are no lights, objects will appear black unless their surface shaders assign colors regardless of the amount of light shining on them.

7.2 Compiling Shaders with `slc`

Like many other programming language systems, shaders must be *compiled*. That is, they must be translated from human-readable form ("source code") into an encoded version that is ready for the renderer to process ("object code").

This extra step also serves another purpose – it allows the shader compiler to check your shader for errors before it is in the middle of rendering a frame.

XRT shaders are compiled using a utility called `slc`:

`slc [`*options*`]` *sourcefile*

The source file may have any name you wish, but by established convention, *XRT* shader source code is stored in a file whose name is the same as the name of the shader [1], with the extension `.sl`. For example, if you had a "plastic" shader, you would store its source code in the file `plastic.sl`. `slc` can compile only one source file per invocation.

Assuming that there is no error found in the shader at compile time, `slc` will write the resulting object code to a file called *shadername*`.shader.so`, where *shadername* is the name of the shader.

7.2.1 Command line arguments

The `slc` program takes the following command line arguments:

`-I` *path*

Just like a C compiler, the `-I` option, followed immediately by a directory name (without a space between `-I` and the path), will add that path to the list of directories which will be searched for any files that are requested by any `#include` directives inside your shader source. Multiple directories may be specified by using multiple `-I` options.

7.2.2 Using the preprocessor

`slc` uses the "C preprocessor" (`/usr/bin/cpp` on Linux systems). This allows you to use the usual C/C++ preprocessor directives such as `#include`, `#define`, and so on.

If your shader uses the `#include` preprocessor directive to "include" another file, `slc` will need to know where to find the file. By default, it will only look in the current directory. You can specify extra directories to search for included files by using the `-I` command-line argument. For example,:

```
slc -I/usr/local/shaders/include myshader.sl
```

will look in the directory `/usr/local/shaders/include` for any `#include`'d files. You can specify multiple directories with multiple `-I` arguments.

Since shaders are passed through the preprocessor, you can also define and use macros (with `-D` or with `#define` in the source code) or use "conditional compilation" (`#if`, `#ifdef`, `#ifndef`, `#else` , `#endif`).

[1] The name of the shader is the name that follows the `surface`, `displacement`, `volume`, or `light` statement in the shader

CHAPTER

EIGHT

TEXTURES

8.1 Converting images to texture with `maketx`

Note: *XRT does not support environment maps, shadow maps or high quality filtering. Therefore, the* `maketx` *utility has a very limited scope.*

8.1.1 General Options

The following `maketx` command-line options can be used for any kind of map:

`-format` *name*

Specifies the format to write texture files - that is, the ImageIO plugin to use when writing the texture file. The default is to use TIFF.

`-p` *searchpath*

Specifies a searchpath for image files. The searchpath is a colon-separated list of directories to search for input files. If no searchpath is specified, input files are assumed to either be in the current directory or are absolute paths.

8.1.2 Texture Maps

The `maketx` program can be used to convert 2D image files – in any format for which you have an `imageio` plugin – to texture files.

`maketx [` *options* `]` *imagefile -o texturefile*

Options include:

`-smode` *wrapmode*

`-tmode` *wrapmode*

`-mode` *wrapmode*

Sets the default *wrap mode* of the texture to one of: `periodic`, `black`, `clamp`, or `mirror`. The `-smode` and `-tmode` flags specify wrapping behavior separately for the *s* and *t* directions, while `-mode` specifies both at the same time. If none of these options are set, the default wrap mode will be `black`.

The wrap mode specifies the behavior of the texture when outside the [0,1] lookup range. Note that this merely sets the *default* wrap mode for a texture. A shader may completely override the wrap mode by using the optional `"wrapmode"` parameter to the `texture()` function.

8.2 Texture Formats

This section documents the formats created by `maketx`. These are the preferred texture formats read by *XRT*, but other formats may be used where noted.

8.2.1 TIFF Texture Common Features

`maketx` uses TIFF files as its preferred texture format. This subsection details the specifications that, unless noted later, are common to all texture file types (plain textures, shadows, environ- ment maps, etc.). This section is only describing the *output* of `maketx` (i.e., the texture format itself read directly by *XRT* when rendering), not the *input* of `maketx` (the much wider variety of image formats that may be converted to *XRT* textures).

8.2.2 Plain Textures

Plain textures have the following additional TIFF tags (using the nomenclature of `libtiff`):

`TIFFTAG_PIXAR_TEXTUREFORMAT` (TIFF tag: 33302, type: string)

> Contains the value `"Plain Texture"`, to identify an ordinary 2D texture. *XRT* Image I/O plugins may read or write this tag using the parameter `"string textureformat"`.

`TIFFTAG_PIXAR_WRAPMODES` (TIFF tag: 33303, type: string)

> Contains the the horizontal and vertical wrap modes, separated by a comma (or just one mode name, if horizontal and vertical use the same wrap mode). Valid wrap modes include: `"black"`, `"clamp"`, `"periodic"`, `"mirrow"`. *XRT* Image I/O plugins may read or write this tag using the parameter `"string wrapmodes"`.

8.2.3 Other Texture Formats

The previous sections describe the tiled, multiresolution TIFF files that *XRT* typically uses for texture, environment, and shadow maps. This is the preferred format, and the format that `maketx` writes by default. But it is by no means required.

XRT will directly read alternate texture formats, assuming that the appropriate Image I/O plugin for reading the format is available.

Using the optional `-format` command-line argument, you can have `maketx` write texture files in other formats.

CHAPTER

NINE

WRITING PLUGINS

9.1 Generator Plugins and Scene File Readers

`Input ( command )` will dynamically load and execute a generator plugin from a DSO/DLL. The renderer will use a dynamic library named *name* `.generator.so` (or, under Windows, *name* `.generator.dll`), where *name* is the first word (up to a space) of *command*.

In the case that the *command* passed to `Input` is actually the name of a scene file in the "input" search path, then *command* is replaced with "*format filename*" where *filename* is the full path of the file, and *format* is the format of the file (given simply by *filename*'s extension). It is presumed that there is a *format* generator which will read commands from the named file (as its sole argument) and make the appropriate *XRT* API calls.

For example,:

```
Input ("teapot.obj")
```

is equivalent to:

```
Input ("obj teapot.obj")
```

Either of these commands will cause the renderer to load a shared library named "`obj.generator.so`" (or "`obj.generator.dll`" under Windows), which is presumed to be somewhere in the "generator" search path.

Generator DSO/DLL's are expected to:

1. Implement a class that is a subclass of a `Generator` class defined as:

   ```
   class Generator
   {
   public:
           Generator();
           virtual ~Generator();
           virtual void run (RendererAPI *renderer, const char *params);
   };
   ```

   The subclass, which publicly inherits from `Generator`, must define a `run` method. Optionally, it may implement a replacement constructor and destructor, as well as any additional data or function methods that its implementation requires.

2. Contain a C-linkage function called `createGenerator` that returns a pointer to a newly allocated and constructed `Generator` object. Using the `EXPORT` macro (defined in `export.h`) ensures that the renderer can correctly reference the symbol.

So, for example, below is a skeleton to implement a generator that acts as a scene reader for "`.obj`" files. The compiled C++ module should be stored in a file named "`obj.generator.so`" (or "`obj.generator.dll`").

```
class ObjReader : public RendererAPI::Generator
{
public:
        ObjReader();
        virtual ~ObjReader();
        virtual void run (RendererAPI *renderer, const char *filename);
};

ObjReader::ObjReader (void)
{
        // Implementation of ObjReader constructor goes here
}

ObjReader::~ObjReader (void)
{
        // Implementation of ObjReader destructor goes here
}

void
ObjReader::run (RendererAPI *renderer, const char *filename)
{
        // Here we read "Obj" from filename, and make RendererAPI calls
        // to renderer in order to communicate the commands in the file
        // to |XRT|.
}

extern "C"
{
        EXPORT RendererAPI::Generator* createGenerator (const char *command)
        {
            return new ObjReader;
        }
};
```

9.2 Shape Plugins

`Shape(` *name* `)` will dynamically load and execute a shape plugin from a DSO/DLL. The renderer will use a dynamic library named *name* `.shape.so` (or, under Windows, *name* `.shape.dll`), which is presumed to be somewhere in the "plugins" search path.

Part IV

Appendices

CHAPTER

TEN

GLOSSARY

Aliasing Undesirable image artifacts related to the rendering process inadequately sampling and representing high frequencies in the image.

Alpha An extra channel in an image giving a measure of *coverage* of a pixel, to aid in image compositing. An alpha of 1.0 means fully opaque; 0.0 means fully transparent.

Ambient occlusion A technique that uses ray-tracing to compute, for each point in the scene, how much of the hemisphere above the point is exposed to the sky (and thus should be illuminated by "ambient" light) versus how much of the hemisphere is occluded by local objects (and is thus not ambiently illuminated).

Anisotropic Something that has a directional dependence. When talking about surface reflectivity, it refers to a BRDF that depends on the rotational orientation of a surface, as well as the angles of the incoming and outgoing light.

Antialiasing Ways of combating aliasing artifacts. Generally encompasses capturing geometric edges without "jaggies," motion blur, depth of field, and otherwise adequately sampling high frequencies in the image.

API Applications Programming Interface. An API is a set of data types and procedures that define the public interface to a library or program.

Artifact A visible imperfection in a computer graphics image, particularly one that betrays the fact that the image is CG and is not a real photograph or physical artwork. Aliasing, polygonal silhouettes, oversimplistic shading, and Mach bands are typical examples of artifacts.

Associated alpha For pixels represented by RGB and alpha (coverage), the practice of *premultiplying* the RGB values by the alpha values. This makes the computations for image compositing simpler.

Attribute Properties that apply to either the scene as a whole or to individual geometric primitives. Examples of attributes include image resolution, object color, shader assignments, and so on.

Backfacing Surfaces whose surface normals face away from the camera viewpoint.

Bake / Baking To turn something that would be computed on-the-fly (perhaps repeatedly or expensively) into a static representation that could cheaply used repeatedly at runtime.

For example, *texture baking* usually means taking shader computations that would expensively be done on every frame, computing the results once and storing them in a texture map, then replacing the expensive shader operations with a simple texture lookup.

Beauty pass The ultimate rendering pass that pulls all the prior passes together to form the final image.

Blinn's law The observation, common in the early days of CGI but first stated by Jim Blinn, that an artist is willing to wait a fixed amount of time for an image to render, and that faster hardware or algorithms simply results in more complex images that take the same amount of time to render.

BRDF Bidirectional reflectance distribution function. A BRDF is a formula whose inputs are the incoming (L) and outgoing (V) directions on a surface, and its output is the portion of light coming from L that is scattered toward V . A BRDF is the heart of a *local illumination model*.

Caustics Areas of intense light that result from curved reflectors or refractors (objects that act like mirrors or lenses, respectively) that focus light.

CGI Computer graphics imagery, especially computer graphics imagery that is produced for use in motion pictures (such as for special effects).

Channel A single "pane" of data in an image. For example, an RGBA image consists of four channels: red, green, blue, and alpha.

CPU Central processing unit - traditionally, the main computational unit that executes software on a computer. Examples of CPUs include the Intel Pentium 4 and the AMD Opteron. See also *GPU*.

CTM Current transformation matrix - at any stage of reading scene input, the 4 * 4 matrix that describes the current "local" coordinate system (relative to the world). Transformation routines (such as `AppendTransform`, `Rotate`, etc.) modify the CTM. The CTM may be temporarily saved and restored with `PushTransform` and `PopTransform`, and also implicitly by `PushAttributes` and `PopAttributes`.

Deformation blur Motion blur of the shape of an object, by blurring the positions of the object's control vertices. This can describe movement that is not "rigid." (See *transformation blur*.)

Depth of field The property of physical cameras that only a limited range of distances can be in focus at any one time.

Displacement bound Extra space added to the bounding box of an object to account for the fact that displacement might make the primitive "poke out" of the original bounds.

Displacement shading Allowing shaders to alter the shape of surface geometry, usually to add fine detail.

DLL Dynamically linked library - a library file that may be loaded dynamically by a program at runtime. *XRT*'s ImageIO, Generator, and shadeop plugins are implemented as DLLs. "DLL" is a name specific to MS Windows; in the Unix world, these are called "DSOs."

DSO Dynamic shared object - a library file that may be loaded dynamically by a program at runtime. *XRT*'s ImageIO, Generator, and shadeop plugins are implemented as DSO's. Note that DSO's are called DLL's under MS Windows.

Fill light A dim, usually nonspecular, light that fills in areas of a scene not illuminated by a key light.

Frame-parallel rendering Using multiple machines on a network to each render a separate frame of an animation. Contrast to *network-parallel rendering* and *multithreading*.

Frontfacing Surfaces whose surface normals face toward the camera viewpoint.

Gamma correction A nonlinear scaling of the values in an image to compensate for the property of all physical display devices that they react to input values in a non-linear way.

Geometric primitive An individual piece of geometry, such as a `Patch`, `Mesh`, `Sphere`, and so on.

Geometry set A named collection of geometric primitives, either corresponding to objects visible to a camera, or a group that may be ray traced. An object may be present in any number of geometry sets. The `Attribute "string geometryset"` controls which geometry sets are active.

Global illumination Calculation of how light affects an entire scene, especially the contribution of light reflected between surfaces (as opposed to coming straight from a light source). In *XRT*, this refers to ray-traced shadows and reflections, indirect illumination, caustics, and ambient occlusion.

Global variable In RSL, any of the built-in variables describing the shading situation, for example, `P`, `N`, `u`, and so on.

GPU Graphics processing unit - the main computational unit of a programmable graphics card, such as the NVIDIA Quadro FX line. In addition to quickly drawing shaded triangles, GPU's are very good at performing mathematical computations in a highly parallelized manner. See also *CPU*.

HDRI Short for High Dynamic Range Imagery, it refers to images that can capture the entire dynamic range of a real scene. In short, a floating-point image or environment map, as opposed to using 8-or 16-bit integers to represent light levels in an image.

IBL Short for *Image-Based Lighting*.

Image-based lighting Lighting a scene by data captured from a real-world scene, usually in the form of an *HDRI* environment map.

Indirect illumination Light that reflects diffusely off an object before illuminating another object in the scene.

Isotropic Means "the same in all directions." When referring to surface reflectivity, it means a BRDF that depends only on the angles of the incoming and outgoing light relative to the surface normal, without regard to the rotational orientation of the surface about its normal.

Key light A major source of illumination in a scene, usually resulting in hard shadows and specular highlights.

Local illumination model A formula that, given the directions and intensities of light impinging on a surface, computes the amount of light scattered away from the surface in a particular direction (such as toward the camera). Synonyms: *Local reflection model*, *BRDF*.

Local reflection model Same as *Local illumination model*.

Metadata Annotations embedded in a shader that do not change the operation of the shader code itself, but describe its contents. *XRT* allows metadata about a shader as a whole, as well as metadata specific to each of the shader's parameters.

MIP-map A texture map for which the results of filtering the texture with a series of larger and larger filters (typically sized in powers of two) has been precalculated and stored with the map, in order to speed render-time texture access. "MIP" actually stands for "multum in parvo" (Latin for "much in little").

Modeler A program that allows users to specify the shape of geometric objects and to place objects, lights, and cameras in a virtual scene.

Moore's law The observation that computing power (as measured by the time it takes to perform certain fixed benchmark calculations on new computers) increases exponentially over time, and that historically it has doubled every 18 months or so over a very long time span. Contrast with *Blinn's law*.

Motion block A transformation or geometric primitive that changes over time. Specifically, a `Motion` statement (with n time values passed to it), followed by exactly *n* identical transformations or by exactly *n* geometric primitives (of the same time), each corresponding to the position or shape at one of the time values, respectively.

Motion blur The property of physical cameras that objects that move relative to the camera leave a streak in photographs, proportional to the length of time the camera shutter is open.

Multithreading Using multiple CPU's or GPU's in a single computer to contribute to the computation of a single rendered frame. Contrast to *frame-parallel rendering* and *network-parallel rendering*.

Network-parallel renderin Using multiple machines on a network to contribute simultaneously to rendering a single frame. Contrast to *frame-parallel rendering* and *multithreading*.

Parametric coordinates. The (typically) two values that uniquely specify a point on a parametric surface, such as a `Patch`.

Pass A final image may require several separate invocations of the renderer - *passes* - which may include generating shadow depth maps, reflection maps, ambient occlusion images, caustic photon maps, diffuse databases for subsurface scattering, etc. The ultimate pass that pulls all the prior passes together to form the final image is called the *beauty pass*.

Primitive Short for *geometric primitive*.

Primitive Variables Data attached to a geometric primitive (per primitive, facet, corner, or vertex), interpolated by the renderer, and that can be accessed in a shader.

Projection A transformation which "flattens" space by removing one dimension, for example, converting points in a 3D space into positions on a 2D object (such as a plane or the surface of a sphere).

Radiosity A global illumination method involving solution by finite element methods. Radiosity solutions usually make the assumption that all surfaces are perfectly diffuse. Sometimes *radiosity* refers colloquially to any global illumination algorithm.

Ray casting A method of global visibility determination, that computes the intersection of viewing "rays" with scene geometry for any purpose.

Ray tracing A method of rendering. Ray tracing solves hidden surfaces, shadows, and reflections by computing the intersection of viewing "rays" and scene geometry.

Reference geometry A description of geometry in a canonical pose. As the "real" geometry is deformed by animation, the reference geometry can be used for shading calculations to ensure that any patterns computed by the shader will stick to the surface as it deforms.

Renderer A program that takes a description of a scene (camera, objects, materials, lights) and produces an image.

RSL RenderMan(R) Shading Language.

Scanline rendering A family of rendering methods that involve projecting geometry into screen space, and handling geometric primitives in image order.

Shadeop Short for *shading operation*, shadeops are the built-in operators and functions in RSL (in other words, the operators and functions that the SL compiler already knows about).

Shader A computer program that describes the appearance of a surface, light, or volume.

Shader type One of `surface`, `displacement`, `volume`, or `light`, or `shader` (indicating a generic shader). The type of a shader is given in its declaration (in the shader source) and determines which global variables it may access and which operations it may legally perform (for example, light shaders may not alter `P`, and volume shaders may not emit light).

Shader usage One of `surface`, `displacement`, `volume`, or `light`, the shader usage explains in what stage of the rendering pipeline a shader is executed. The shader loaded for a given usage must have a matching shader type (or, if a generic shader, must still only perform operations legal for the usage).

Shading quality A measure of how frequently color values are computed on surfaces. Larger values imply that shading is computed more frequently, therefore more total shading calculations will be performed (with the expected increase in cost). Smaller values will result in fewer total invocations of the shader, thus rendering in less time and memory, but with lower image quality.

Space A synonym for "coordinate system."

Subsurface scattering Illumination that scatters internally through a translucent material such as marble or skin, often re-emerging quite far from where it entered the material.

Texel TEXture ELement. A texel is one pixel in a texture map (including shadow and environment maps).

Texture mapping Taking colors (or other data) from a stored image file and applying the pattern to a surface to give added detail.

Transformation The placement of an object (or light, camera, etc.). Transformation includes translation, rotation, and scaling of an object, and is accomplished with the API routines described in Section 2.5.

Transformation blur Motion blur of the position/orientation of an object. This can describe "rigid" motion of an object but does not allow an object do bend or deform. (See *deformation blur*.)

INDEX