Design of Standardized Representations for Animated Scenes for Ray Tracing

David Whiteford
The University of Texas at Austin

1 Introduction

In computer graphics, the process of creating an image generally consists of several steps: a scene file is created, the scene file is loaded into a graphics system, and finally the system creates the image in a process called rendering. The scene file stores information about the scene from which the image will be created. More specifically, a scene file generally consists of a collection of objects, information about where the objects are and how they look, lights to illuminate the scene, the viewpoint from which the scene is seen, and if the scene is animated, information about how everything in the scene changes over time.

The file format used to represent the scene has a large impact on the system as a whole. If the format is not robust enough, then there will be some useful graphics effects that will be hard or impossible to represent within the file. If the format provides too many features, then it becomes harder to understand and integrate into a system. If the format does not organize the data it represents in a good way, then the system will have a hard time rendering the image in an efficient manner. Thus, a format suitable for the system being used and task being performed must be chosen.

Two of the most commonly used methods of rendering are Z-buffering and ray tracing. Z-buffering is a method of approximating what a scene looks like, and is relatively computationally inexpensive. Ray tracing, on the other hand, is more realistic, but is much more computationally expensive compared to Z-buffering. Even though ray tracing is more realistic, it is not feasible to do in real-time on modern hardware, which has led Z-buffering to be the most widely used algorithm for many applications, especially those that are real-time. As such, most of the graphics hardware out nowadays is designed specifically for Z-buffering. As hardware in general and algorithms advance, however, ray tracing is becoming more feasible. To make the switch from Z-buffering to ray tracing happen, different things need to happen. For example, new, more advanced algorithms must be discovered and faster hardware, maybe even hardware targeted towards ray tracing, must be developed. Also of importance is designing a file format suitable for this task, which this paper will focus on.

The goal of this paper, specifically, is to evaluate design alternatives for representing animated scenes to be used in ray tracers. First, existing graphics file formats that address related goals will be discussed and evaluated. Next, the exact requirements for a file format for animated scenes to be used in ray tracers will be assessed. Finally, a suitable file format, as well as thoughts on what would be required to implement support for the given format, will be presented.

2 Background

2.1 Ray Tracing

Ray tracing is a method used to create very realistic images. The idea behind ray tracing is to simulate light as it is transported throughout a scene. In a real scene, light is emitted from a light source over a continuous range of directions. This emitted light is called radiance. The irradiance of a point on a surface is defined as the integral of the incoming radiance over all directions. When the light reaches the surface of an object, the light is reflected from the surface, transmitted through the object, or both, also over a continuous range. This continues until the light hits an eye point or film, at which point it is seen by the viewer. The combination of irradiance over all points on the eye or film defines the image.

Finding the exact irradiance at a specific point requires solving a complex integral involving the objects and lights in the scene. This is not feasible, so ray tracers normally discretize the integral to approximate it. This is done by modeling light as rays and simulating these rays as they propagate through the scene. This is generally done in one of two ways: “shooting” or “gathering.” The “shooting” method involves casting rays from light sources and seeing where they end up, much like what happens in reality. The “gathering” method calculates irradiance by casting rays from a point and seeing how much incoming radiance arrives at the point from those rays. Either way, if many rays are shot out at each point, the complexity of the process will grow very quickly. Thus, another approximation which casts fewer rays but still retains most of the real-world effects is often necessary.

Whitted style ray tracing, due to Turner Whitted, is one example of this. Whitted style ray tracing uses a simplified version of the “gathering” method along with Phong shading. Rays are cast from the eye towards the scene. When a ray intersects with an object, the color of that point when viewed from the eye must be determined. Phong shading is performed by determining the contribution of light by the light sources that are directly visible from the point. Reflectance effects are approximated by casting a ray in the direction that the incoming ray would be reflected by the surface. If the object is at all translucent, a ray is cast through the object in the direction that the incoming ray would be refracted according to Snell’s Law. This way, most of the effects desired in ray tracing are reasonably approximated by sending out at most two rays at each intersection point.

When done naïvely, even Whitted style ray tracing is expensive. One of the most time-consuming aspects of ray tracing is finding the nearest object that a ray intersects. To find which object that is, a naïve solution would simply test for intersection against every object in the scene and the closest object would be chosen. The use of spatial sorting acceleration structures, such as octrees, *k*d trees, and BSP trees, can greatly improve the efficiency of this process. These structures sort objects based on where they are in the scene. This way, intersection tests only need to be performed on objects that are near the ray. Also, intersection tests can be performed starting from the objects closest to the origin of the ray, so that as soon as an intersection is found, we know we have found the closest object and can stop. Acceleration structures play a very key role in making ray tracing feasible on modern hardware.

2.2 Skinning

Often in computer graphics we wish to represent complex, realistic looking figures, such as humans. To represent such a figure, different approaches can be taken. One such approach is to use a triangle mesh, which is simply a set of triangles that are grouped together to form one object. This works great for representing an object during a single frame, but representing a detailed object during animation using this method is harder. One solution is to create a separate triangle mesh for each position that the object can be in, and use the appropriate mesh during the appropriate frame. This solution is not a good one for various reasons. For one, storing multiple copies of a detailed object will use up much more memory than desired. Also, this method requires that the artist spend time producing a triangle mesh for each pose. Another solution is to partition the triangle mesh into a hierarchy of separate parts. For example, a human mesh can start with a torso, have a head, arms, and legs attached to the torso, have a hand attached to each arm, and have a foot attached to each leg. This way, when the character needs to change positions, all that needs to be done is to change the position of each body part relative to its parent in the hierarchy. This solution is also not a great one, because during animation, the body parts connecting to each other will not blend well, so the joints will not be visually compelling.

The solution that combines the best of both of these other methods is called skinning. Skinning is the process of binding a mesh "skin" to a hierarchy of bone joints. Each vertex of the mesh is bound to a number of joints using different weights. The weighted average of the joints determines the position of the vertex. During animation, the movement of the joints is specified, and thus the positions of the vertices move smoothly with the joints. Using this method you can create objects that deform much more smoothly and naturally than objects with rigid transformations.

2.3 Shaders

Most file formats and rendering systems allow the user to specify the material of an object, which in turn determines how an object appears. The material often includes attributes such as the color of the object, different ways in which the object reflects light, and the translucency of the object. This is useful in many cases and allows for a decent approximation of how objects appear in real life, but for more realistic looking images, better tools are required. Shaders are often used to create customized, and often very realistic, appearances of objects.

A shader is usually written by a user, and is used to determine what the outgoing radiance from a point on an object is. Shaders most often take the form of a small function or program that can be embedded into the description of a scene itself. A shader can be as simple or complex as the user desires, and can include things such as the color of the object itself, light coming in directly from light sources, light reflecting off of other objects in the scene, the angle that the object is being viewed from, and even effects that are not realistic at all if the user desires.

3 Previous Work

Given the immense number of applications that use or create graphics, it is no surprise that there are also many different formats for scene representations being used. This section will discuss some of these formats.

3.1 RenderMan Interface

The RenderMan Interface, or RI, was designed by Pixar in the 1980's, and is said to be the "Postscript of 3D Graphics." The idea was to create a standardized representation for 3D scenes that was general enough to fit the needs of most applications that use 3D graphics. It will be good to look at the RenderMan Interface because it will give us an idea of how to design a 3D graphics file format to be general, and because it includes a powerful shading language. RI comes in two forms. One is an API that specifies a collection of function calls that tell the application what to draw. These functions can be implemented in any language, and can thus be called from within any language for which the RenderMan Interface has been implemented. The other form is the RenderMan Interface Bytestream, or RIB, which is the file format version of RI. Files encoded in this way can be stored using either an ASCII representation or a binary representation. There is essentially a one to one mapping from RIB to the RI API. RIB allows any scene to be represented and stored in a way that will allow it to be rendered in any system implementing the RenderMan Interface.

The RenderMan Interface is great in some areas, but not so much in others. For example, it includes only rudimentary support for animation. Each frame of animation is specified one at a time using the RiFrameBegin() and RiFrameEnd() functions. In between the two functions, the entire scene that must be drawn during that frame must be specified completely. It is not possible to simply modify the parts of the scene that have changed for use in the next frame. This makes it very difficult for an application to exploit the frame to frame coherence that is often present in animations. RI also does not support the skinning method described above.

The RenderMan Interface does include a robust shading language called the RenderMan Shading Language. It is a procedural shading language that is syntactically similar to the C programming language. There are useful built-in functions supplied by the language that make many things, such as ray tracing, possible. The illuminate() function, for example, provides a way to specify the outgoing radiance from a specific point. This function is normally used in light shaders, which specify how the lights in a scene act. A cone and a statement are passed into the illuminate() function, and the statement is executed with respect to each point within the cone that the light source illuminates. Whereas illuminate() is the "shooting" function, illuminance() is the "gathering" function. A cone and a statement are passed into illuminance(), and the statement is executed with respect to each object within the cone that illuminates the point. This can include light sources, as well as other objects that reflect light onto the point. The job of the illuminance() function is to gather the incoming radiance and convert that to an outgoing radiance. The phong() function, which provides a quick and easy way to perform Phong shading, is also included.

3.2 Macromedia Flash

Macromedia Flash is a file format that is designed primarily for animations that will be shared over the internet, and is geared towards 2D graphics. Flash will be interesting to look at because it is supposed to be elegant and handle animation well. Since Flash was designed with efficiency in mind, the files are stored using a binary encoding. This allows them to be quickly transferred over the internet, and to be read and processed efficiently. The file consists of a header, a series of tags, and an end tag. Tags, other than the end tag, consist of one of two types. Definition tags define some object and add that object to a dictionary. Control tags perform actions, which can include using objects in the dictionary.

Flash handles animation very well. A display list is used to represent the scene at any given time. The display lists consists of objects at certain depths, with each depth being occupied by at most one object. Initially, control tags are used to add objects to the display list. A ShowFrame tag then displays all of the objects on the display list. The display list is maintained from frame to frame, meaning the entire scene does not have to be re-specified for every frame. All that is required is to add, remove, or update objects as needed from frame to frame.

3.3 PBRT

PBRT is a physically based ray tracer developed by Matt Pharr and Greg Humphreys. Looking at it will give us an idea of what is involved with a photorealistic ray tracer. PBRT is a ray tracer that attempts to render images as close to physical reality as possible, which it does through things such as accurately modeling materials of surfaces, scattering light through mediums such as fog and murky water, allowing for different camera and film models, and outputting the final image using an image format that allows for arbitrary brightness of light. Similar to the RenderMan Interface, scenes are given to PBRT through the use of an API, but can also be stored in a file using a file format corresponding to the API. When a scene file is given to PBRT, it will parse and load the file, and then make the appropriate calls to the renderer itself. Also similar to RenderMan, PBRT does a great job of rendering a single image, but does not include much support for animation. The entire scene must be re-specified for each frame, and no convenient method of moving the objects in the scene around is provided.

3.4 X3D

X3D, or Extensible 3D, is a standard developed by the Web3D Consortium that defines an abstract representation for real-time 3D graphics. It also defines multiple encodings for the abstract representation, including XML and binary encodings. XML, or Extensible Markup Language, is a standard which describes a general, very flexible plain text file format for use in storing and exchanging data. X3D defines a core component required in all X3D compliant runtime systems, as well as many additional components which each provide a specific set of features that can be selectively supported by each system.

Being geared towards real-time graphics, X3D has good support for animation. In X3D applications, there is a notion of time. Static objects, which are objects that don't move or change over time, are defined without regard to time. For dynamic objects, on the other hand, the attributes of the object, such as where it is and what it looks like, are defined with respect to time. The system then displays the objects accordingly. Support for skinning is included through the Humanoid Animation component of X3D. The joints, skin vertices, and weights as described above, as well as many more attributes, can be defined. X3D also enables the use of embedded scripts in languages such as Java and ECMAScript. These scripts can be referenced from other files or in some cases, such as with ECMAScript, they can be embedded directly into the X3D file itself. X3D interacts with the scripts by calling certain functions when certain events happen, at which point the function will execute and possibly change data in the scene.

Despite the rich set of features X3D supports, it does not quite meet our needs, as it is targeted towards real-time 3D graphics and not ray tracing. Thus, it is missing certain features that are important to realistic ray tracing, such as an index of refraction for objects, a sufficiently general camera model, and support for a shading language suitable for ray tracing.

3.5 COLLADA

COLLADA, which stands for Collaborative Design Activity, is another standard that attempts to facilitate the process of storing and exchanging graphics between people and applications. It is similar to X3D in its overall structure and functionality, but certain aspects of it are more appealing. COLLADA is encoded using XML, and handles animation and skinning in a fashion similar to that of X3D. It supports features that are helpful in realistic ray tracing, such as an index of refraction and a realistic camera model. There is also more of a community backing for COLLADA. One of the selling points for COLLADA is that developers of major graphics applications, such as 3ds Max and Maya, can contribute to the design of COLLADA. This makes COLLADA interesting because this way it will support all of the features needed by these applications, and thus facilitate the exchange between them. Also like X3D, however, COLLADA lacks support for a shading language suitable for ray tracing.

3.6 Maya

Maya is a very popular 3D graphics modeling application. Looking at the file format used by Maya will be of interest because it is designed towards storing data about a scene in a modeler, as opposed to a scene in a renderer. A Maya file can be stored in one of two ways; binary or ASCII. If it is stored in ASCII, it is represented as a series of instructions in the MEL programming language. This is different from the API approach used above, however, because only a few of the MEL instructions are supported, so it is much simpler and more similar to a normal file format.

A Maya file is broken up into distinct parts that come in order; Header, (Non-procedural) File references, Requirements, Units, Nodes, attributes, and parenting, Disconnections, Connections, and Procedural references. The Header contains general information about the file. The (Non-procedural) File references section contains files that are included into and can be used by the current file. The Requirements section specifies what requirements are needed by the file, such as a certain version number. The Units section defines which units are used to measure length and angles. The Nodes, attributes, and parenting section is where the bulk of the file will usually reside. This is where many objects in the scene, such as geometry and lights, are declared. The Disconnections and Connections sections disconnect or connect attributes from or to each other. If two attributes are connected, then when the source attribute changes, the destination attribute is changed to match the source. The Procedural references section declares references to external MEL scripts, which are run after the Maya file is loaded.

Maya handles animation using a method called keyframe animation, which will be discussed in the next section. Basically, it is a great way to compactly and easily specify the movement of an object throughout an animation. It also supports the model skinning method as described above.

4 Design Considerations

There are many design decisions that will affect which features are included in the final file format, as well as how it looks.

4.1 General

To promote usability, there should be built-in support for basic things that are useful for 3D graphics in general, such as simple geometry, triangle meshes, materials, and lights. Simple geometry should include spheres, cubes, cones, cylinders, and other common shapes. Materials should include at least emissive, ambient, diffuse, specular, shininess, reflectivity, translucency, and index of refraction attributes, as well as support for texture mapping. At least ambient, point, and directional lights should be included.

4.2 Encoding

The way a scene file is stored has a large impact on the size, speed, and usability of the file. There could be a number of different encodings.

A binary encoding is compact and efficient, but is hard to work with. Tools are needed to create and modify files, as you can't just read or edit them using a text editor. This is inconvenient and makes debugging harder. Also, there could be compatibility issues because of different endianness on different machines.

Plain text is easy to work with, but is bigger and slower than a binary representation. If XML is used, the plain text encoding will also be very well structured. XML is more universal, so there are already existing tools that can be used to parse and create files. Also, it is human readable and editable, which makes development and testing much easier.

Creating an API, such as the kinds the RenderMan Interface and PBRT use, gives more flexibility as far as being able to use the features of a programming language. For example, a for-loop could be used to instantiate several copies of a sphere in different locations, whereas with a normal file each copy would have to be individually stated. Scenes represented in this way are generally less portable, though. Also, a scene file written in a programming language would have to be recompiled after every update, which can be a hassle.

Given that at this point this scene representation would be primarily used as a research tool, it makes sense to choose the more user-friendly XML encoding. If space efficiency and loading time became an issue, a binary encoding might be more suitable and could be developed.

4.3 Animation

Proper support for animation is an important feature. With only rudimentary animation support, the user must re-specify the entire scene for each frame. Thus, the internal acceleration structures must be rebuilt every frame, which is expensive. Proper support for animation would include a mechanism to update the scene rather than re-specify the entire scene every frame. This would allow the ray tracer to incrementally update the acceleration structures instead of rebuilding them every frame, which would be more efficient. Even if the acceleration structures were rebuilt every frame, having an update mechanism would still remove the need to destroy and reinitialize each object every frame.

Proper support for animation does not only mean having an update mechanism. There should also be features that allow the artist to easily specify how objects in the scene will actually be animated. One very important and useful way of doing this is through the use of keyframe data. Keyframe data specifies where something, an object or vertex of an object for example, is at certain points in time. The find where the object is at any given time, the system interpolates between the positions of the surrounding keyframe points. This interpolation can be simple linear interpolation, or can be more complex ways of interpolating data, such as Bezier curves or

B-splines. Keyframe animation is one of the primary methods for animating objects in many applications. X3D, COLLADA, and Maya all support this.

For generality, it would be useful to allow embedded scripts, written in a language such as Javascript, either stored in a separate file and referenced or in the scene file itself. This would allow the user to animate objects in any way desired, whether it be interpolating between keyframe points, evaluating an analytical function, or writing a complex program to move objects around in a customized way. Embedded scripts can be used for a variety of other purposes, as well, not just animation.

4.4 Skinning

In order to facilitate the creation of compelling characters and objects, support for skinning should be included in the scene representation. As can be seen by the review of the file formats above, the blend-weight skinning method is the method that is used in most applications. It's fairly simple, but it's still effective enough to make very convincing animations. The joints and vertices would be specified in the file as described above, with each vertex bound to a number of joints. The joints would then be moved by one of the animation mechanisms described above, which would in turn move the entire character.

4.5 Shaders

Shaders have become an important part of ray tracing. The writer of a ray tracer cannot predict all of the effects that the user will want to use, and even if each effect desired by a user could be included as part of the ray tracer, this would make it bulkier than necessary for most purposes. A good shading language provides a powerful, general way to let users develop a wide variety of effects themselves.

There are a few things that help to make a shading language good. For one, it should provide plenty of functionality. This includes things such as being able to access the surface material and the lights in the scene, cast rays from any point to any point, and use control statements such as "if," and "for." In addition to functionality, the language should be intuitive and easy to use. One way to make it intuitive is to use a syntax that is already familiar to most people. The similarity of the syntax of the RenderMan Shading Language to C is an excellent example of this. Also, there should be plenty of built-in functions, such as various math functions and functions to handle common shading algorithms. All of these things contribute to the user being able to easily write a shader to accomplish whatever task is desired.

5 Proposed File Format

Based on the design considerations above, none of the formats described above support all of the desired features for representing dynamic scenes to be used for ray tracing. This does not mean that an entirely new format is required, however. COLLADA does support many of the desired features, and is closer to supporting all of the features than any of the other formats discussed. It is encoded using XML, has great support for animation and skinning, and has plenty of basic features useful in both ray tracing and 3D graphics in general. The only thing it is missing is support for a proper shading language. The RenderMan Shading Language is very powerful and meets all of the needs described above for a shading language for ray tracing. Thus, if proper support for the RenderMan Shading Language could be added to COLLADA, the combination would be suitable for representing dynamic scenes to be used for ray tracing.

6 Thoughts on Implementation

Implementing support for this format would require a decent amount of work. The main tasks to be done are loading the COLLADA file, loading the RenderMan shaders, and interacting with the rendering system to actually render the scene.

Most of the work of loading the COLLADA file is already done. There is a COLLADA parser written in C++ that is available for download from the COLLADA website which will parse everything other than the RenderMan Shading Language enhancements. Support for this will need to be added. The bulk of the work will be done by including domRSL_*.{cpp,h} and domProfile_RSL.{cpp,h} files, which should be fairly similar to the related files for other shading languages, such as CG and GLSL. To see the structure and behavior of including a RenderMan shader in a COLLADA file, refer to Appendix A. Once the COLLADA parser is done, a new schema file for the format incorporating the changes in Appendix A should be produced. Once this is done, support must still be added to parse and run the actual code of the shaders. Source code for doing this is supplied by many open source projects, one such project being Pixie, which is a RenderMan-like photorealistic renderer. The source code for compiling and running a shader should be taken and incorporated into the COLLADA parser.

Now that the file is loaded, the scene must be given to the underlying rendering system to be rendered. To do this, a runtime system should be developed which will interact with the rendering system in various ways. The runtime system needs to be able to do certain things based on how the rendering system works. The following paragraphs will describe what would need to be done to include support for this file format in a few different types of rendering systems.

The first rendering system will be PBRT. Given that PBRT does not support animation, the runtime system will have to handle it. The runtime system will keep track of the state of the scene at any given point in time. Each time a frame should be drawn, the runtime system will specify the entire scene to PBRT through the API that PBRT provides. This means that although the file format supports animation, PBRT will not be able to take advantage of all of the benefits, particularly those related to efficiency. When a situation arises in which a shader needs to be run, such as a ray hitting the surface of an object which has a shader associated with it, PBRT should call the runtime system to handle it. While the shader is being processed, it might actually be necessary to make calls back to PBRT, such as when a new ray needs to be traced. Thus, the code that runs the shaders must be augmented to support this. One final observation is that skinning is not directly supported in PBRT. Thus, the runtime system should keep track of any skinned models in the skinned model format, but must convert this to a triangle mesh format when specifying it to PBRT.

Another choice for a rendering system is a RenderMan based ray tracer, such as Pixie. Many aspects of including support for our file format into Pixie are similar to those related to PBRT; animation and skinning should be handled in the same way. The major difference comes from how shading is handled. In Pixie, the RenderMan Shading Language is inherently supported, so instead of Pixie calling the runtime system to evaluate a shader, it will simply do it itself. This makes it significantly easier to use something like Pixie as opposed to PBRT.

Another type of rendering system available is a ray tracer specifically designed to run in real-time. The benefit of this type of system is that it can take advantage of the optimizations available with a file format that supports animation well. If the rendering system supports the same animation features that COLLADA supports, such as keyframing, then all that needs to be done is to specify the scene to the rendering system once and let it handle the animation. Otherwise, the runtime system will still need to keep track of the scene as it is animated, but it need only re-specify the objects that have changed from frame to frame. Also, if skinning is supported, the rendering system, as opposed to the runtime system, can keep track of the skinned model. If the real-time ray tracer does not support the RenderMan Shading Language by default, it will have to be integrated as with PBRT.

Based on these observations, although it would most likely be easier to include support for our file format into Pixie, it would be better overall to be able to do so for a general real-time ray tracer. This makes sense, as real-time ray tracing was the original goal of the format.

7 Conclusion

The file format I ended up choosing as suitable for representing animated scenes to be used for ray tracing was COLLADA with added support for the RenderMan Shading Language. This format is a good choice for a few reasons. First of all, it is as functional as we need and want it to be, as it has all of the desired features discussed above in the design considerations section. Next, the format is already created for the most part, so there is no need to create an entirely new format. This is good because having multiple file formats that accomplish the same tasks will only serve to make digital content less standardized and portable. Finally, there is a good amount of community support behind COLLADA and the RenderMan Shading Language, which is good for different reasons. For one, there are existing tools to deal with them, as we have seen in the implementation section above. Also, there is existing content available for use, such as an abundance of shaders written in the RenderMan Shading Language. Finally, because COLLADA is backed by major graphics software developers, it is likely to change to reflect the changing needs of graphics applications as time progresses.

As ray tracing animated scenes becomes more and more feasible, we need to make some changes to make the shift possible, one of which is using a suitable file format. COLLADA with RenderMan Shading Language support will be a good choice for this task.

8 Acknowledgements

I'd like to thank Bill Mark for advising me throughout this project. I'd also like to thank Don Fussell and Vijaya Ramachandran for providing valuable feedback.

Appendix A

This is an extension of the COLLADA 1.4.0 specification to include support for the RenderMan Shading Language. The framework for including a RenderMan shader in a COLLADA file is similar to that for including a shader currently supported by COLLADA. For more information, see "COLLADA – Digital Asset Schema Release 1.4.0."

Elements

This section describes the elements related to including a RenderMan shader in a COLLADA file and how to use them.

<library_effects>

Declares a module of effect elements.

<effect>

Description of a COLLADA effect. This element has the attribute "id," which is used by other elements to reference the effect. Children include <newparam> elements specifying the parameters needed by the effect, and <profile_*> elements specifying how to compute the effect.

<newparam>

Creates a new parameter in the FX Runtime and assigns it a type and initial value. It must contain exactly one of four children: <float>, <point>, <color>, or <string>. The <float> element should contain a single floating point number. The <point> element should contain three floating point numbers separated by spaces. The <color> element should contain any number of floating point numbers, depending on how the system specifies the color type, separated by spaces. The <string> element should contain a string of characters.

<profile_RSL>

Opens a block of platform-specific data types and <technique> declarations. This should have one <technique> element as a child.

<technique>

Holds a description of the shaders, parameters, and passes used to perform this effect. Must contain the "sid" attribute, which describes the element. Must have at least one <code> or <include> child, and one <pass> child.

<code>

Contains an embedded block of source code. This is where the code for the actual RenderMan shader goes.

<include>

Imports source code or precompiled binary shaders into the FX Runtime by referencing an external resource. The "url" attribute specifies where the resource is located.

<pass>

Contains information about the different shaders in this technique. Contains <shader> elements as children.

<shader>

Declares a shader and specifies how it interacts with the rest of the COLLADA file. The "stage" attribute must be one of the following values: LIGHT, VOLUME, TRANSFORMATION, DISPLACEMENT, SURFACE, or IMAGER, depending on the type of shader being declared. Contains <name> and <bind> elements as children.

<name>

Contains the entry symbol for the shader function. The symbol is given by a string of character in between the <name> and </name> tags.

<bind>

Binds a symbol in the COLLADA file to a parameter in the shader. The “symbol” attribute specifies the name of the parameter in the shader. Contains either a <param> element or one of <float>, <point>, <color>, or <string>. If it is one of the latter four, a constant value is bound to the parameter.

<param>

References a predefined parameter in the COLLADA file. The “ref” attribute contains the name of the parameter.

Behavior

The FX Runtime must follow certain behaviors when interacting with a RenderMan shader.

Certain pieces of information are communicated to a RenderMan shader through the use of global variables. When a shader is called, the FX Runtime must make sure that the global variables that the shader might use are set to the correct values. Also, certain pieces of information are returned from a shader through the use of global variables. The FX Runtime must also make sure to read these values after the shader is done to get the information it needs. For a complete list of which global variables apply to which types of shaders, see “The RenderMan Companion.”

Surface shaders must take into account other objects in the scene, even if those other objects do not have RenderMan shaders associated with them. For example, a surface shader must be able to get and use information about a light defined within the COLLADA file. Thus, the runtime system must be able to simulate the effects of a RenderMan shader for lights and other objects based on their normal behavior in a COLLADA file.

Both COLLADA and RenderMan shader require a default value for parameters, so two default values must be specified. However, the COLLADA default value will override the shader default value, because the COLLADA default value will be passed in as an argument to the shader.

Example

This example demonstrates a matte surface shader.

```
<library_effects>
  <effect id="Matte">
    <newparam sid="ambient">
      <float> 1.0 </float>
    </newparam>
    <newparam sid="diffuse">
      <float> 1.0 </float>
    </newparam>
    <profile_RSL>
      <technique sid="default">
        <code>
          surface
          matte(float Ka = 1,
                      Kd = 1)
          {
            point Nf = faceforward(normalize(N), I);
            Oi = Os;
            Ci = Os * Cs * (Ka*ambient() + Kd*diffuse(Nf));
          }
        </code>
        <pass>
          <shader stage="SURFACE">
            <name>matte</name>
            <bind symbol="Ka">
              <param ref="ambient" />
            </bind>
            <bind symbol="Kd">
              <param ref="diffuse" />
            </bind>
          </shader>
        </pass>
      </technique>
    </profile_RSL>
  </effect>
</library_effects>
```

References

Barnes, M. "COLLADA – Digital Asset Schema Release 1.4.0," The Khronos Group Inc., January 2006.

Macromedia, Inc., "Macromedia Flash (SWF) File Format Specification, Version 7," 2005.

Pharr, M., and Humphreys, G. *Physically Based Rendering*. Morgan Kaufmann, 2004.

Upstill, S. *The RenderMan Companion*. Addison-Wesley, 1992.

Web3D Consortium. "X3D Specification," April, 2006. http://www.web3d.org/x3d/specifications.

Whitted, T. An Improved Illumination Model for Shaded Display. *Communications of the ACM*, 23(6), pp. 343-349, June 1980.

World Wide Web Consortium. "Extensible Markup Language," February 5, 2006. http://www.w3.org/XML.