

Implementing RenderMan - Practice, Problems and Enhancements

Philipp Slusallek, Thomas Pflaum, Hans-Peter Seidel

Computer Graphics Group, University of Erlangen

Am Weichselgarten 9, D-91058 Erlangen, Germany

email: {slusallek,tspflaum,seidel}@informatik.uni-erlangen.de

Abstract

The RenderMan interface has been proposed as a general interface to rendering systems, yet only a few implementations of the interface exist. In this paper we describe the implementation of the RenderMan interface on a general rendering architecture that supports various rendering algorithms. Specifically we discuss the implementation of the RenderMan Shading Language and its integration into our rendering architecture. Special attention is focused on the problems that we have encountered and how they can be solved. Additionally, we suggest extensions and enhancements to the current interface definition, which would make RenderMan easier to implement and more flexible to use.

1 Introduction

The RenderMan Interface [Pix89, Ups90] was designed by Pixar as a standard interface between modeling and rendering systems. The idea was to create a high level description for rendering jobs and thus separate the modeling and the rendering programs that were (and often still are) tightly integrated.

Although the RenderMan interface was designed as an open specification that could be supported by most rendering systems, there are only a few commercial rendering systems available that actually support the RenderMan interface. The best known implementations of the interface certainly are “Photorealistic RenderMan” and “Quick-RenderMan” from Pixar itself. None of the large animation systems like Alias, Softimage or Wavefront support the interface for their renderer. If they support programmable shading at all, they have chosen a proprietary interface (like OpenRenderer for Alias).

In this paper we describe a new implementation of the complete RenderMan interface on top of an object-oriented rendering system that supports various rendering and lighting algorithms, including global illumination. The implementation includes the RenderMan API (RM-API), the RenderMan Interface Bytestream (RIB) protocol and the RenderMan Shading Language (SL). Besides describing the implementation of the RM-API and the RIB protocol, we focus on the shading language, because it is the most advanced and critical part of the RenderMan interface. The SL code is executed in the heart of the rendering program (probably millions of times) and must therefore be as efficient as possible. This is why we decided to compile the SL code to object code that is linked with the renderer, instead of using interpreted code.

We finally analyze the problems that we encountered in implementing the RenderMan interface and describe solutions. We also recommend extensions and enhancements to the RenderMan interface and the shading language in particular, that would make them easier to implement and more flexible to use.

1.1 The RenderMan Interface

The RenderMan Interface consists of three distinct components: The RenderMan Application Programming Interface (RM-API), the RenderMan Interface Bytestream (RIB) protocol and the Shading Language (SL) [Pix89, Ups90, HL90].

The RenderMan API is a collection of C functions that are used by an application to transfer geometry and rendering attributes to the renderer and to control the rendering process. These functions must be linked with an application and can either directly drive a specific renderer or can generate a description of the render job in a so called RIB file.

The RenderMan Interface Bytestream (RIB) protocol can basically be viewed as a binary or ASCII encoding of a sequence of RM-API calls. Most functions of the RenderMan API have a simple translation into a RIB command. A renderer can later be used to read and execute the commands in a RIB file. Thus modeling and rendering can be separated into independent programs.

The last part of the RenderMan interface is the RenderMan Shading Language (SL). It is a high level language designed to specify the interaction of light with various parts of a scene. It supports high level language constructs to manipulate points and colour and supports writing various types of shaders.

These shaders specify different aspects of the shading process. Light source shaders describe the distribution of light from a light source, the surface shaders calculate the interaction of light with a surface, while volume shaders specify the interaction with participating media. Additional shader types are provided to describe the imaging process, i.e. the mapping from calculated floating point light intensities to pixel values and to manipulate geometry through transformation and displacement shaders.

There are several technical reasons why the RenderMan interface has found little support in the industry (and certainly many political and marketing reasons too, which we will not discuss here). One shortcoming of the RenderMan interface is the incompleteness of the specification and its strong foundation on the REYES rendering architecture [CCC87]. Especially the last point makes an implementation on other rendering architectures difficult.

While designing our rendering system, we faced the problem of how to interface it to modeling systems. After analyzing the advantages and disadvantages of various interfaces, including the RenderMan interface, we decided to use RenderMan as a starting point. It was the only interface that did a reasonably good job in describing a rendering task and is still flexible and powerful enough to be extended for special purposes.

1.2 Organization of the Paper

The following sections will discuss the implementation of the three parts of the RenderMan interface. The RM-API and its mapping onto our rendering architecture [SS94] is described in Section 2. The RIB protocol is implemented by directly mapping RIB commands to their equivalent RM-API calls. Details are given in Section 3.

The heart of our implementation is the Shading Language Compiler. Section 4 gives details on the implementation and its integration into our rendering architecture. It describes the shading language compiler that is used to compile the shading code to object code using C as an intermediate language. The compiled shaders can then be dynamically linked into the renderer at runtime. We also describe the optimizations that are possible in the interface between the renderer and the compiled shader code.

In Section 5 we discuss some shortcomings of the RenderMan interface definition and the problems we encountered by implementing it on a rendering architecture very different to the REYES architecture [CCC87]. We indicate possible solutions and directions how the RenderMan interface definition could be changed to ease supporting it on different rendering architectures.

Finally, we conclude with a discussion of the benefits and problems of using a standard interface to a rendering system.

2 The RenderMan API

2.1 Overview

The RenderMan-API (RM-API) is the procedural interface to RenderMan. The API allows programs to directly control a RenderMan renderer and to use RenderMan as a high-level, high-quality visualization tool. The possibilities of using the RenderMan interface as a programming tool is demonstrated by the “3D Graphics Kit” and Quick-RenderMan in the NextStep environment [Nex92]. Here, the RM-API is encapsulated into an object-oriented class hierarchy and is the standard for 3D graphics output.

The interface functions of the RenderMan API can be grouped into five categories:

- The *graphics state* functions manage the state of the interface. Functions for saving and restoring the current state on stacks allow a hierarchical scene description to be built.
- The functions from the *options* category mainly deal with specifying the viewing, imaging and general rendering parameters.
- The *attribute* functions specify shading, illumination and geometric transformations.
- The *geometric primitive* functions are used to define surface information ranging from polygons to procedurally defined surfaces.

2.2 Implementing the RenderMan API

Our interface to the RenderMan-API is mainly implemented using three stacks, one each for the options, the attributes and the geometric transformations. They are manipulated by the functions from the corresponding categories. Most other functions can be directly mapped to the creation or manipulation of equivalent objects in the rendering system. A more detailed description of the mapping between the functions of the RM-API and our renderer is given in the following subsections.

2.2.1 The Graphics State

The state of the RM-API is controlled through pairs of functions `Ri*Begin()/Ri*End()`, which start and end a new state. Whenever a new state is started, the previous state is suspended and restored upon the end of the new state. Thus a hierarchical structure can be imposed on the scene description.

The initial state is opened through `RiBegin()`, which initializes the RM-API as well as the rendering system, if that has not yet been done. The function `RiFrameBegin()` opens a new state for a single frame. Upon `RiFrameEnd()` all attributes, options, and primitives defined within this state could be removed. Static objects in an animation can be created by placing them outside of a `RiFrame` block, in contrast to objects changing from frame to frame. In our system each of these calls is mapped to the creation of a subgraph in the hierarchical scene database. All objects created within the new state will be added to this subgraph, which could be removed after rendering the frame.

The function `RiWorldBegin()` freezes all options and creates a virtual camera, a renderer, a lighting, an image shader, and an image frame object for the resulting picture, with the appropriate parameters given by the current options. They are added to the current scene subgraph and a view object is created. The active view objects control all information about how a certain image should be rendered. One view object is created for each new RenderMan frame.

Calls to `RiAttributeBegin/End()` or `RiTransformBegin/End()` manipulate the corresponding stack of current attributes or the current transformation from modelling to world coordinates. These stacks are used whenever primitives are to be created, and determine how these primitives will be grouped.

All created primitives are collected until the currently active attribute or transformation state is to be changed. Then these primitives, which are known to share common attributes, are put into a

special container object, that describes these attributes. Thus, the geometric objects do not need to know about their attributes individually, but rather this information is contained within the scene graph. The design to separately store geometry and other information allows for flexible extensions of the rendering system.

2.2.2 Options

Options, as defined by the RenderMan interface definition, include a complete description of the virtual camera, and the imaging (or display) process, i.e. mapping calculated floating point light intensities to pixel values. The camera description includes aspect ratio, resolution, clipping planes, depth of field, and shutter opening times. The imaging options include parameters for filtering of pixel samples, exposure control, colour and depth quantization, and output specifications for the resulting image.

All these options are supported by the interface and map directly to corresponding objects in our system and their parameters. Many of the options can be supported by all rendering algorithms, but some of the options (like depth of field or motion blur) are ignored by certain renderers.

Options specific to our renderer can be accessed through the generic `RiOption()` function. They include the specification of the lighting method (e.g. direct lighting with shadows calculated using ray tracing, or lighting using radiosity or Monte Carlo integration), and of the rendering colour space.

2.2.3 Attributes

Attributes in the RenderMan interface are shading and geometry attributes. The shading attributes specify the current colour and opacity, the current surface and volume shader, shading quality, light sources, and texture coordinates. Geometry attributes include the current transformation, orientation and sidedness of surfaces, basis matrices and trimming curves for spline surfaces, a clipping box, the current displacement and transformation shaders and parameters for the geometric approximation of curved primitives.

Our renderer handles shading attributes in the RenderMan interface by carrying them along on the attribute stack until geometric primitives are being added to the scene graph. Then the shading attributes are used to instantiate an object of the `RenderManShader` class, our internal equivalent to a RenderMan surface shader, passing the attributes that relate to surface shading. The `RenderManShader` class is the interface between our rendering engine and the RenderMan surface shader code and are described in Section 4. The volume shader attributes (exterior and interior shader) are also passed to the `RenderManShader` object, which later uses it to inform the renderer about the volume shaders on each side of a surface.

The call `RiShadingRate()` needs special attention within our RenderMan interface. This call changes the way a surface is shaded, by adjusting the maximum screen size which may be covered by a single shading evaluation. Although the call applies directly to the render, it is implemented by the `RenderManShader` class. During rendering it supplies information to the renderer about the desired sampling frequency for this surface.

Area light sources are added to the scene graph like normal geometric primitives, with additional information that identifies them as light emitting surfaces. Again this information is not directly attached to the primitive, but rather applied indirectly by its position in the scene graph. Special interior nodes in the graph allow the emission properties for their subgraphs to be specified.

At rendering setup time, the Lighting object searches the database for light emitting surfaces and prepares the lighting calculations.

The function `RiIlluminate()` is used to switch light sources on and off. It is implemented by adding special interior nodes to the scene graph, which - during traversal for rendering - change the state of light sources.

The current transformation is maintained on the stack and used when adding geometric objects as described above. In our system, geometric primitives exclusively operate in their own coordinate

system and are positioned through special transformation objects in the scene graph. The same applies to their texture coordinates.

The RenderMan interface allows texture coordinates to be specified at each vertex of a surface or through the current attribute state. If an object does not have texture coordinates specified for its vertices, the current texture coordinates on the attribute stack are used.

The renderers in our system currently do not implement deformation shaders, but support is included in the scene database, should a new renderer support them. A special container object is inserted, encapsulating any geometry to which a deformation shader applies. At a later stage, deformation shaders could be implemented by querying the geometry objects for a finely subdivided mesh and applying the deformation to it. Displacement shaders are implemented, but they currently only affect the surface normal. Supporting true surface displacement would, for most renderers, need a similar technique as for deformation shaders.

All other geometric attributes are only maintained on the attribute stack and used when creating geometric objects. Since they are used at creation time no support is needed for them during traversal of the scene graph.

2.2.4 Geometric Primitives

The RenderMan interface definition defines the following geometric primitives: planar convex polygons (`RiPolygon()`), planar concave polygons possibly with holes (`RiGeneralPolygon()`) and the same for multiple polygons that share common vertices (`RiPointsPolygons()`, `RiPointsGeneralPolygons()`). It supports bilinear and bicubic patches (possibly rational) with arbitrary basis matrices, meshes of these patches, and NURBS with trimming curves. Additionally support is included for quadrics (sphere, cone, cylinder, single sheet hyperboloid, paraboloid, disk and torus).

All geometric primitives are supported by equivalent objects in our rendering system. Additionally we have added, through use of the generic `RiGeometry()` function, simple triangles and quadrilaterals, planar and arbitrary triangular meshes.

RenderMan supports solid modeling with the two calls `RiSolidBegin(operation)` and `RiSolidEnd()`, where `operation` can be “primitive”, “union”, “difference” or “intersection”. These calls may be nested and generate appropriate primitive objects. Currently evaluation of solids in our implementation is only supported through ray tracing renderers.

The use of retained geometry is very restricted in the RenderMan interface definition. The `RiObjectBegin()` and `RiObjectEnd()` calls only allow objects of a single type to occur within this block. The enclosed objects are retained and may later be placed into the scene graph through the use of `RiObjectInstance()`. Because our rendering system was designed with the experience that multiple instantiations of scene subgraphs can dramatically reduce the amount of storage for nontrivial scenes and, if well implemented, can also reduce rendering time, it fully supports retained geometry and no restrictions apply. For compatibility reasons the unrestricted use of retained geometry is controlled through an implementation specific option.

2.2.5 Creation Versus Traversal Time Support

Options and attributes can either be completely bound to the database at scene creation time or might need some support during scene traversal. Because support during scene traversal can be expensive in terms of rendering time, our interface design tries to minimize this traversal time overhead by applying most attributes to database objects at scene creation time. Table 1 gives an overview about those RenderMan attributes that need traversal time support.

Evaluation of attributes that need traversal time support is generally delayed until after traversal of the scene graph. During traversal, nodes of the scene graph that would change the state, register themselves with the traversal object. The current status of the traversal object is recorded in a path object. It can then be queried for a certain attribute value, which is obtained by asking the registered

Transformations	Since our rendering architecture allows multiple instantiations of a scene subgraphs in the database, the traversal time support for transformations is required anyway. This also allows for a simpler implementation of operations on geometric objects in their intrinsic coordinate system.
Texture coordinates	The transformation of texture coordinates could be directly applied at the creation time of geometric primitives, but our architecture already allows the remapping of shaders for instantiations of a subgraph. Thus it makes sense to also allow the remapping or transformation of texture coordinates. In our rendering system we make heavy use of lazy evaluation strategies, so that the texture transformation overhead is only required when the texture coordinates are requested by a shader.
State of light sources	RenderMan allows to switch light sources on and off for scene subgraphs. Thus traversal time support is unavoidable.
Surface shader	Unavoidable, if we want to allow for remapping of shaders for scene subgraphs, instead of binding the shader directly to a surface.
Surface orientation	The current surface orientation could be implemented for RenderMan at primitive creation time with a flag for each geometric primitive. But to allow for multiple instantiations of a scene subgraph with different orientations, we decided to bind this attribute at traversal time. In contrast to RenderMan our system defaults to a right handed coordinate system.

Table 1: Features in the API that need traversal time support

internal nodes for its value. Using this technique, the overhead for traversal time support of attributes can be minimized.

3 The RIB protocol

3.1 Overview

Using the RenderMan-API still couples the modeling and rendering programs very tightly. Although the modeling program can now use a well defined interface to the renderer, it must still be linked to it. The RenderMan Interface Bytestream protocol (RIB) allows a separation of these two tasks with the exchange of rendering data through an external representation. With RIB, rendering jobs can be archived in files (e.g. for later batch rendering) or the RIB stream can be sent across a network for simple distributed processing.

The RIB protocol comes in two flavors: a binary and an ASCII encoding. With a few exceptions all RM-API functions can also be used in the RIB stream. The general format of a RIB stream consists of a sequence of RenderMan requests with arguments, much like a shell command language. Because many arguments are often optional, the use of “token - value” pairs is used in these cases. The token identifies the type and semantics of the following value. The same technique is also used in the RM-API together with variable length argument lists.

3.2 Implementing the RIB Protocol

Because the mapping of the the RIB protocol to the RenderMan API is straightforward, we decided to implement the RIB parser on top of the RM-API. The RIB stream is parsed by collecting a RIB request with all its arguments and calling an interface function for that request. The interface function checks the arguments and eventually calls the corresponding RM-API function. Testing the arguments

of the RIB request is straightforward but tedious to implement and is supported by the extensive use of macros in the implementation.

The RM-API can be used in immediate or in retained mode. Retained mode is used together with RIB files, so that request are not immediately executed, but are rather added to the scene graph. This allows the RIB scene to be edited before it is rendered, or use of the system as a RIB editor (although the resulting RIB file will have a different structure than the input file, because of the rearrangements in the system).

4 The Shading Language Compiler

In comparison to the RM-API and the RIB protocol parser, the SL compiler is more difficult to fit into a rendering system. Because the SL operates in the heart of the renderer, a well designed and optimized interface is required. The interface between the renderer and the SL must introduce little overhead, so that a RenderMan shader can operate with the same performance as a build-in shader. Additionally, the renderer and the interface must support special features of the SL, like calculating derivatives of arbitrary expressions.

We will first give a short overview of the SL before we describe the SL compiler and the interface to the renderer.

4.1 The RenderMan Shading Language

The RenderMan interface definition as described so far allows for a flexible description of a rendering job. This part of the interface was not too difficult to design, because there is a common understanding of how surface geometry, projections and general surface attributes can be described. But for the description of light sources or volume and surface shading no such common understanding exists. There are many different shading algorithms ranging from simple heuristic models to elaborate, physically based shading models.

Because the shading of surfaces and volumes, the description of light sources, and the imaging process need a very flexible interface, the designers of the RenderMan interface added a programmable interface – the RenderMan Shading Language (SL) [Pix89, Ups90, HL90]. The SL allows the procedural specification of the local interaction of light with matter (surface and volume shaders) and light source distribution functions (light shaders). Support is also included for general, procedural transformation of geometry (transformation shader), small surface changes (displacement shaders) and imaging functions (image shader). A good overview of the SL and its design is given in [HL90].

The SL is modeled after the C language, but has some high level constructs for dealing with colour and points and special control structures for lighting calculations. The SL supports writing procedural shaders through special shader functions. These functions are very much like classes in an object-oriented language: They have parameters that act like instance variables and the same shaders can be instantiated with different parameters for each surface. On the other hand, they offer only one single external method - the shading function. This function is called by the renderer whenever it needs to determine, e.g., the colour of a surface, or the intensity of a light source in a certain direction. The information about the environment in which the evaluation takes place (coordinates of a surface point, normal vector, texture coordinates and so on) is passed to the shader in external state variables.

The arguments passed to a shader can either be specified when the shader is instantiated, or they can be supplied by attributes of the surface that is to be shaded. The RenderMan interface allows arbitrary data to be specified for surface vertices, which is interpolated across the surface and can then be used by a shader.

The shading language specifies a set of special purpose functions, which offer support for many common shader tasks. These functions include the calculation of derivatives and normals, reflection and refraction directions, the phong shading model, the noise function, and allow spline interpolation

of points and colours. They also support access to texture maps with optional filtering. The SL allows specification of points and colours in any user defined coordinate system or colour space.

In the following section we describe the implementation of the shading language and the integration of RenderMan shaders into our rendering system.

4.2 The Shading Language Compiler

The SL code is executed in the heart of the rendering program (probably millions of times) and must therefore be as efficient as possible. This is why we decided to compile the SL code to object code using C as an intermediate language. For each shader the compiler generates three C functions: the core shader code, an instantiation and a destructor function. The generated C code is passed to a C compiler for generating dynamically loadable PIC (Position Independent Code). At runtime the shader object code can be dynamically linked directly with the renderer, instead of using interpreted code. Using this approach, the compiled shader code is platform specific and does not allow precompiled shaders to be shared between platforms. As a result, and unlike Pixar's implementation, shaders must be distributed in source code to be useful.

4.2.1 Special Compiler Features

Although the SL and C are very similar languages, there are enough syntactic differences and other reasons which prohibit a simple translation of the shader code to C. Instead a complete syntactic and semantic analysis is required. We give a short overview of these features and how they affect the shading language compiler.

The state of the renderer is passed to the shader in external variables, but depending on the shader, only certain variables are accessible and others may be read-only. Also, certain language constructs (illuminate, solar and illuminance) may only be used in certain shaders, and within these constructs, additional external variables are accessible.

The SL distinguishes between varying and uniform variables. Uniform variables do not vary across a surface, while varying parameters must be interpolated by the renderer. The compiler can optimize a shader by identifying expressions that only contain uniform or constant terms. These expressions can then be moved out of the core shader code and need only be evaluated once when the shader is instantiated.

The SL supports overloaded functions that perform different operations depending either on the number or types of parameters, or even on the return type (e.g. `noise()` and `texture()`). To implement this, the compiler must have detailed knowledge about the surrounding expression of the function call.

Special code needs to be generated for special language constructs. This includes the derivative (see Section 5.1) and the solar, illuminate and illuminance constructs. The latter three language features are similar to loops that execute the following statement or block multiple times, depending on the rendering environment.

In the following subsections we describe the various stages [HL90] of a RenderMan shader in our rendering system.

4.2.2 Compilation

The SL compiler starts by piping the shader source code through the standard C preprocessor, which is responsible for file inclusion, conditional compilation and macro expansion. Then the compiler performs lexical analysis and parsing to generate a parse tree of the shaders. It also maintains an extensive symbol table, including a list of shaders and functions, and a list of colour spaces and coordinate systems used by the shaders.

For each variable the symbol table contains the name, type, class (uniform or varying) of the variable, the point of definition (external or local variable, or formal parameter), and a parse tree,

which contains the initialization code for that variable. The list of shaders, colour spaces and coordinate systems is used in the interface to the renderer (see below).

The next step is code optimization: We only perform optimizations that are specific to shader code and which cannot be detected by a back-end C compiler, due to missing informations. Remaining (and platform dependent) optimizations can then be performed by the back-end C compiler.

The main optimization performed by the SL compiler is moving uniform expressions out of the shader's body. There are more options to perform optimizations, but these are sometimes difficult to detect. As an example a varying formal parameter of a shader normally receives its actual value from attribute values that were interpolated across a surface. If, for a given surface that attribute is actually uniform, certain subexpressions of a shader could become uniform and in this case could be moved out of the core shader code. But in contrast to Pixar's RenderMan implementation, we do not bind a shader to a specific surface, but rather to a scene subgraph. This technique requires less instantiations of a specific shader, but does not allow us to take advantage of this kind of optimization (which would be difficult to implement anyway, if there is more than one such parameter used in the same expression).

For each shader the compiler generates three C functions: The core shader code, an instantiation and a destructor function. The instantiation function is used to create and bind a shader to a scene subgraph. It receives the instantiation values for formal parameters (they can later be overwritten for a specific call to the shader) and the shader coordinate system. The function creates a shader instance structure that contains information about the usage of external parameters in the shader, as well as initialized local variables, including the shader coordinate system.

For each shader source file the compiler also generates a function returning the list of shaders in this file and a list of colour spaces and coordinate systems used by the shaders. The colour spaces and coordinate systems need only be initialized once when the shader code is loaded into the renderer, except for the special coordinate systems "shader", "current", and "object".

Finally the generated C code is passed to a C compiler for generating dynamically loadable PIC code. This code is then placed in a predefined directory, where the renderer can find it later.

4.2.3 Loading and Instantiating Shaders

When a rendering job is started, the system searches predefined directories for shader code. Each shader file found is then dynamically loaded and the containing shaders are registered with the scene database. Then the RIB file is read or RM-API calls are received. Whenever a set of surfaces sharing common shading attributes is to be added to the scene graph, we check to see if we already have instantiated a matching shader. If none is found, a new one is created using the current shading attributes and is added to the list of instantiated shaders. This shader cache severely reduces the number of shader instantiations in nontrivial scenes, where the same shader is used for many surfaces.

4.2.4 Calling the Shader

The flexible architecture of our renderer does not allow us to assume any order for calls to a shader. Thus we must make provisions for a single shader instance to be called for any surface at any time during rendering. This again forbids certain optimizations as already mentioned.

As a consequence, we have taken care to perform the least amount of work when calling a shader: We use lazy evaluation for all external state variables of the renderer (like normal, tangent, texture coordinates etc.). Because the shader contains information about which state variables are actually used in the shader, often many values do not need to be calculated at all. This is especially useful for varying surface attributes, which would need to be interpolated across the surface.

The next optimization when calling a shader is caching of formal parameters: Each shader remembers the last surface it has been called for. If the shader is called for the same surface again only the actually varying surface attributes must be changed before calling the shader. If the shader is being called for a different surface we must set all varying surface attributes. If for any varying formal

parameter there is no surface attribute, we have to set it back to the value at instantiation time. Each formal variable has a dirty flag indicating whether we actually need to reset this parameter.

Using the described strategies, we have implemented a compromise between the flexible use of RenderMan shaders by our renderer and keeping the cost of calling a shader at a minimum.

5 Enhancing RenderMan

The RenderMan interface definition is a complex document. During our implementation we have encountered some problems, ambiguities, and incomplete specifications which we want to discuss in this section. We also suggest enhancements to the Version 3.1 of the RenderMan interface definition, that we found or believe to be useful.

For instance some unnecessary restrictions are that the SL does not allow recursion or that retained geometry can include a single object. These restrictions should therefore be removed.

5.1 Handling Derivatives

The definition of the shading language includes special functions to obtain derivatives of arbitrary expressions in a shader. This functionality is a very powerful tool for programming shaders, but the current definition of this feature makes a conforming and portable implementation for various rendering algorithms very difficult.

The functions that compute derivatives can all be implemented using the two basic functions `Du(expr)` and `Dv(expr)`. Each function computes the derivative of its argument with respect to the underlying surface parameters u and v . The problems start with the exact definition of these functions. The RenderMan reference [Pix89] is not clear about whether the derivation extends to variables in the expression, which themselves depend on u or v . It turns out, that the derivation includes these expressions [Apo93].

The benefit of having the derivative extend to these variables is that more complicated expressions are possible. The expressions can now depend on values computed in conditionals or loops. On the other hand, this is a nightmare for the implementation: Not only can the computed values be non-continuous (due to conditionals) and thus the derivative be infinite, but special attention must be taken if a call to `Du()` is itself inside a loop or conditional. Because the derivation can include arbitrary code, a symbolic derivation cannot be done and the renderer must therefore resort to numerical methods.

This requires multiple evaluations of the expression (including the code that leads to it) near the original point on the surface and estimating the true derivative. It could, however, happen that evaluation of the code at nearby point does not compute values at all (because the call to `Du()` is inside a conditional), thus making the whole derivative undefined.

We suggest, that the use of any function that relies on computing derivatives is not allowed inside of loops and conditionals by the SL definition. This would remove undefined results and allow for an easier implementation without losing much functionality. We have not found examples, where this functionality was really needed.

In our implementation of the SL each call to `Du()` or `Dv()` is translated to a conditional that either calls an internal function or computes the value of the expression and returns. The internal function recursively calls the shader for nearby points with a parameter set, so that it will now compute the value of the expression and return immediately. From these values an estimate of the derivative is computed and returned to the shader.

If the shader processes large pieces of a surface at once (like the REYES algorithm does) nearby values are normally available at no extra cost. A ray tracing renderer, on the other hand, normally considers surfaces with point sampling and has no knowledge about the surrounding surface. Thus it must recompute nearby values for each shader call. Using a cache for these values seems like a good idea, but is difficult to implement due to the irregular sampling pattern.

5.2 Layered Shaders

The current definition of the RenderMan interface does not allow a shader to call another shader. We believe that it would add substantial functionality to support multiple, layered shaders for primitives. Currently, a completely new shader must be written, even if only additional detail should be added to a shader. This leads to extensive duplication of code with all associated problems.

Layered shaders would, for instance, allow to add dirt and scratches to any surface, just by adding a “dirt” shader on top of another shader. Other examples include adding a label or punching holes in other surfaces with already complicated shaders. Many existent shaders could benefit from this feature.

Layered shaders can be implemented in a RenderMan shader through stacks for each shader class. The RM-API could be extended to allow shaders to be added to or removed from the top or bottom of a shader stack. The renderer would then call the shaders in the correct order. The order depends on the type of shader: surface colour is best calculated starting with the topmost shader on a surface, while displacement shaders must be called starting from the bottommost shader to produce meaningful results.

As an alternative the SL could allow shaders to call other shaders and require the shader programmer to call the shaders in the correct order.

5.3 RenderMan and Global Illumination

The RenderMan interface was designed with global illumination in mind [Ups90, HL90]. The support offered by the SL is through the illuminance construct for surface shaders. As defined, illuminance is called with a point and the center axis and spread angle of a cone. The statement or block following the illuminance statement is then executed once for each direction inside the cone, from which light arrives at this point. Inside the illuminance block additional external state variables L and Cl are defined, which provide the incoming direction and the amount of light, respectively.

This information is not enough for global illumination algorithms, since the illuminance construct actually performs an integration over the incoming light. To properly weight the samples, additional information about the solid angle that is covered by this sample must be provided. To make meaningful computations possible the units of the light values must also be provided: For global illumination light is generally specified as radiance [$W/m^2 sr$] [CW93]. This information would make it possible to use the shading language to describe arbitrary bidirectional reflectance distribution functions (BRDF) of advanced reflectance models for global illumination computations.

The image shader could also be used to compute tone reproduction operators [TR93]. The problem here is that an image shader operates on only one single pixel and has no knowledge about the complete image, which would be required, to derive the tone reproduction operator.

6 Discussion

Our implementation shows, that the RenderMan interface can be implemented on other rendering architectures and that it provides a good basis for an interface to a rendering system. Nearly all geometric data can be described by the standard functions, and generic functions make it possible, to extend the interface for special purposes. Also for the shading attributes and the control of the rendering process RenderMan offers a flexible interface with the RM-API and the RIB protocol.

The RenderMan shading language is a flexible tool to describe the interaction of light with its environment and to add geometric features to a scene. Although the SL can be implemented on other rendering architectures, it was difficult to provide the required support for many standard rendering algorithms. Additionally, some awkward details in the language make the implementation unnecessary difficult and we have suggested alternatives in these cases.

The object-oriented structure of shaders provides a flexible framework for programming shaders, but the RenderMan interface does not yet allow the flexibility to use shaders as building blocks for creating composite shaders.

We have used our implementation with example RIB files generated by the ALIAS PowerAnimator [Ali93], the Geomview program [Phi93], the Next 3D Graphics Kit [Nex92] and examples collected from the Internet (see Figure 1 and 2). Our experience was, that most examples did not use the available functionality of RenderMan to transfer all the available information. The examples normally included little more than the surface geometry with only a few simple shaders applied to them. As a result, we often had to manually edit the RIB files to apply interesting shaders to the models.

The RenderMan interface in its current state is certainly not the final word for a general interface to a rendering system, although we believe it to be currently the most flexible and comprehensive interface. Still many possible options are not or not fully implemented, like support for advanced rendering and shading algorithms. Also the object-oriented ideas in RenderMan are not fully exploited.

Further work is planned on our RenderMan implementation, mainly in the direction of better supporting advanced rendering and shading algorithms and to further extend the RenderMan interface for global illumination computations.

7 Acknowledgement

This work would have been impossible without the continuous support by Wolfgang Heidrich (who also created some of the models using ALIAS PowerAnimator), Sven Campagna and many others from the computer graphics group at Erlangen University. All members of the lab participated with their suggestions and discussions of various problems that appeared while developing our rendering architecture and its RenderMan interface.

References

- [Ali93] Alias Research Inc., Toronto, Canada. *Making Models in ALIAS*, version 4.0 edition, 1993.
- [Apo93] Tony Apodaca. private communications. 1993.
- [CCC87] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. *Computer Graphics*, 21(4):95–102, July 1987.
- [CW93] Michael F. Cohen and John R Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press Professional, 1993.
- [HL90] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculation. *Computer Graphics*, 24(4):289–298, August 1990.
- [Nex92] Next. *3D Graphics Kit, Version 3.0*. Next Computer Inc., 1992.
- [Phi93] Mark Phillips. *Geomview Manual, Version 1.4*. The Geometry Center, University of Minnesota, Minneapolis, 1993.
- [Pix89] Pixar. *The Renderman Interface*. Pixar, San Rafael, California, September 1989.
- [SS94] Philipp Slusallek and Hans-Peter Seidel. An architecture for global illumination calculation. Technical report TR-94-3, Universität Erlangen, IMMD IX, April 1994.
- [TR93] Jack Tumblin and Holly Rushmeier. Tone reproduction for realistic images. *IEEE Computer Graphics & Applications*, 13(6):42–48, November 1993.
- [Ups90] Steve Upstill. *The RenderMan Companion*. Addison-Wesley, 1990.