The Australian National University

Computer Science Technical Report

June 1993 Technical Report TR-CS-93-02

Data Shader Language and Interface Specification

Brian Corrie
Paul Mackerras

Department of Computer Science
The Faculties

Computer Sciences Laboratory
Research School of Physical Sciences and Engineering

Joint Computer Science Technical Report Series

Data Shader Language and Interface Specification

Brian Corrie *Paul Mackerras*

Department of Computer Science
Australian National University

Abstract

The process of visualizing a scientific data set benefits from an extensive knowledge of the domain in which the data set is created. Because an in-depth knowledge of all scientific domains is not available to the creator of a visualization system, a flexible and extensible system is essential in providing a productive tool to the scientist. One approach to providing this flexibility is through a *shading language* that enables users to write programmable *data shaders* that determine how scientific data sets are rendered.

This paper describes the implementation of such a shading system. The system consists of two parts, a *shader library* and a *shader compiler*. The shader library does not provide a shading model directly, but instead provides a means of loading and binding externally created shaders to a rendering engine which uses the library. The shader compiler is used to compile a shader description, written in the shading language, into a form that the shader library can load. The shader library has been used in both a ray-tracing geometric renderer and a ray-casting volume renderer.

1 Introduction

Programmable shading models and shading languages are used to provide the user of a rendering system with a flexible and extensible means of describing how visible surfaces in a scene are shaded. This flexibility is necessary because of the vast number of subtle surface characteristics that are required to create a realistic image. Such flexibility is also important in a volume rendering system, as no single shading and classification model can display the myriad of subtle interactions that take place in a three-dimensional scientific data set. The system presented in this paper provides this flexibility through the use of a programmable shading system, enabling the user of a volume renderer to describe how data sets are to be rendered.

Volume rendering is one of several rendering techniques that can be used to visualize three-dimensional data sets. Volume rendering is the process of generating an image directly from such a data set without the generation of an intermediate geometric model. Ray-casting is a common technique for volume rendering [17, 11, 16]. It is an image space algorithm which determines the entire volume's contribution to a given pixel before processing the next pixel in

the image. A volume's contribution to a pixel is determined by casting a simulated light ray through that pixel and into the volume. As a ray traverses the volume, it is sampled at regularly spaced intervals. Each sample value is interpolated and mapped to a color and opacity. These colors and opacities are then integrated along the ray to determine the final color for the ray and therefore the pixel. The color and opacity at each sample point is determined by the shading model used by the volume renderer.

To date, most volume renderers apply a single, monolithic shading model to the volumes being rendered. Even with a highly parameterized shading model and flexible mappings of data to color and opacity, a single model can be overly restrictive. If the provided parameterization is not adequate for all volumes being rendered a renderer with a different shading model must be used or a new renderer that can produce the desired visualization must be developed. The creation or modification of such a renderer requires an in-depth knowledge of the rendering technique being used as well as a high level of skill in programming techniques. Clearly, this is an unreasonable requirement to place on the general user. A more flexible method is to allow the users of the system to describe how they want to visualize the data set, through the use of a programmable shading system.

The shading system implementation described in this paper provides a flexible framework for the exploration of three-dimensional data sets. Data shaders, a new class of shader developed in this research, are similar in functionality to RenderMan surface shaders [19], except that they are applied at a sample point within a volume, rather than at a point on the surface of a geometric primitive. The data shader attached to a volume is responsible for shading the subvolume around each sample point within that volume. The shader used for each volume need not be the same, and need only be as complex as the visualization requires. The flexibility provided by such a shading system allows users to apply their domain specific knowledge to the design of new and powerful shaders and to the creation of new ways of exploring and analyzing their data sets.

This paper is structured as follows. Section 2 briefly discusses the history of shading languages, focusing on the RenderMan shading language. Section 3 describes our extension to the RenderMan shading language to incorporate data shaders. Section 4 describes an implementation of a shader library that enables a rendering system to use the shading language and programmable shaders. Section 5 briefly describes our experiences using the shader library with two different rendering systems. Section 6 discusses areas that require further research.

2 Shading languages

Shading languages were developed as a powerful and flexible method of specifying a shading model for rendering geometric primitives. They were introduced to the graphics community by Cook's paper on shade trees, in which he described a flexible, tree-structured shading model capable of representing a wide range of shading characteristics [2]. Perlin [14] extended Cook's shade tree language to include general control flow structures. Both Perlin [14] and Peachey [13] introduced the concept of procedural solid textures at this time.

2.1 The RenderMan shading language

The RenderMan shading language [15, 19, 8] is an extension of Cook's shade trees, providing a small set of high level data types, a full set of flow control structures, and a rich set of mathematical and shading related functions. The RenderMan interface specifies that six classes of programmable shaders (light source, surface, volume, transformation, displacement, and imager shaders) can be used within the RenderMan framework. The six classes of shaders provided by the RenderMan shading language are distinguished by their class variables. The class variables define the environment in which a shader executes, and provide the key interaction mechanism between the rendering and shading domains.

The RenderMan shading language simplifies the task of writing shaders by providing a library of support functions for commonly used shading operations. Two high level data types (points and colors) are also provided. Points are defined as three-dimensional position or direction variables, while colors are defined as multi-channel variables with an arbitrary number of channels. Arithmetic operations work on points and colors, as well as floating point values. These operations operate on a coordinate by coordinate or channel by channel basis. If a scalar floating point value is used in an expression involving a point or color data type, it is promoted to that data type by duplicating the scalar value in each channel of the data type. Color and point variables can not be used in the same expression.

Shader definitions are similar to function definitions in the C programming language [10], except that the instance variables (the variables in the function's parameter list) are given default values. These instance variables are set when an instance of the shader is created, and can be different for each instantiation. A shader of a given class is declared by using the appropriate keyword (light, surface, volume, transformation, displacement, or imager) instead of a return type for the shader. Because shaders return the result of their computation in their class variables, no return value is given for a shader. Local variables in a shader are analogous to those in C.

Control structures in the RenderMan shading language are similar to those in C (while, for, if-then-else). It also provides special constructs (illuminate, solar, and illuminance) for spatial integration of light that is emitted from light sources and light that is incident on a surface. Functions can be declared in the RenderMan shading language, and are declared as a function is declared in C, except that the extended data types (points and colors) can be used as return types. Parameters to functions in the shading language are passed by reference.

2.2 Surface shaders

In the RenderMan specification, surface shaders are used to determine the color of the light reflected in a given direction from a point on the surface of a geometric primitive. It uses the location of the point, the orientation of the surface, the light sources in the scene, and other information to compute this color. With respect to visualization, this functionality can be very useful for mapping parameters, such as temperature or stress, to colors on the surface of an object described by geometric primitives.

Variable	Meaning	Variable	Meaning
`Cs, Os`	Input color and opacity	`P`	Surface position
`N, Ng`	Surface normal and geometric normal	`dPdu, dPdv`	Change in position with u, v
`I`	Incident light ray	`E`	Position of the camera
`L`	Direction from surface to light[1]	`Cl`	Light color[1]
`u, v`	Surface parameters	`du, dv`	Change in u, v across surface
`s, t`	Surface texture coordinates	`Ci, Oi`	Output color and opacity

Figure 1: Surface shader class variables

Variable	Meaning	Variable	Meaning
`Cs, Os`	Input color and opacity	`Vn`	Number of channels in the volume
`P`	Sample position	`Ds`	Distance to sample
`I`	Incident light ray	`Din`	Distance to volume entry
`E`	Source point of incident ray	`Dout`	Distance to volume exit
`Cl, L`	Light color and direction to light[1]	`Dunit`	Unit distance for opacities
`Ci, Oi`	Output color and opacity	`Dstep`	Distance between samples
`u, v, w`	Parameterized sample location	`Du, Dv, Dw`	Volume size in u, v, and w dimensions

Figure 2: Data shader class variables

3 Language description

The shading language described in this paper is a subset of the RenderMan shading language, with the extensions necessary to support data shaders. These include an extension to the basic data types, a new set of class variables for the data shader class, extensions to the built-in functions, as well as an extension to the method in which a shader's instance variables are used. These extensions are described in detail below.

3.1 Class variables

The class variables provided to a data shader define the state in which the shader executes. To effectively shade volumetric data, these variables must describe the volumetric state at the sample point. The class variables defined for data shaders are similar to those defined for RenderMan surface shaders, and include variables such as the input color and opacity (`Cs` and `Os`), the surface point (`P`), and the incident ray along which the computed light will be returned (`I`), as well as an assortment of others. To extend the set of variables to deal with volumetric data, it is necessary to specify volume sampling information, information about how the incident ray interacts with the volume data set, and three-dimensional sample location parameters. Complete lists of the class variables for surface and data shaders are given in Figures 1 and 2.

The extended class variables used for data shaders define the geometric and volumetric state of the sample point relative to the volume data set that is being sampled. The class variable `Vn`

[0]Usage is only valid within an illuminance statement

specifies the number of channels in the data set. For scalar data sets, there is only one channel (`Vn = 1`), while for vector data sets or data sets that have multiple scalar values for each sample point `Vn` specifies how many channels exist. Access to sample values of the volume at a given point is provided by the `sample` built-in function. This function is described in Section 3.3. `Ds` specifies the distance from the sample point `P` to the source point of the incident light ray. `Din` and `Dout` specify the distance from the source point of the incident light ray to the points at which the ray enters and leaves the volume being sampled. `Dstep` gives the distance between sample points in the volume. `Dunit` defines the distance over which opacities are defined. For example, a subvolume with an opacity of 0.5 attenuates fifty percent of the light that passes through the subvolume if the distance the light travels in the subvolume is `Dunit` units in length. If the light travels more or less than `Dunit` units in a subvolume of a specified opacity, then the computation of the attenuation is more complicated. The built-in `attenuation` function, described in Section 3.3, performs this computation. The class variables `u, v`, and `w` specify the three-dimensional parameterized location (between 0 and 1) of the sample point within the volume. The variables `Du, Dv`, and `Dw` specify the size (in the world coordinate system) of the volume in the `u, v`, and `w` dimensions.

3.2 Data types

To make the use of transfer functions relatively simple, a new data type is added to those provided by the RenderMan shading language. A *map* has one or more channels, each of which represents a function which is defined on the real interval [0,1]. The functions take values in the range [0,1]. Typically, a function would be represented by a set of knot points used for linear or cubic interpolation. The `colormap` built-in function is used to obtain the value of a mapping function at a given point.

3.3 Built-in functions

Several built-in functions have been added to the RenderMan shading language. The `sample` and `gradient` functions provide access to the multiple channels of a data set. The `sample` function returns the sample value at the specified point in the given channel of the data set. The `gradient` function returns the gradient at the specified point in the given channel. For example, the second component of a vector data set at the sample point `P` can be obtained through the shading language statement `float y = sample(P,1)`. Similarly the gradient at point `P` for the same channel can be obtained with the statement `point N = gradient(P,1)`. Note that these functions use 0 based indexing to identify a channel.

The `colormap` built-in function is similar to the traditional RenderMan `texture` statement, except the mapping that is performed is one dimensional rather than two dimensional. A value is retrieved from a map by specifying a map, a map channel, and a mapping value (between 0 and 1) to the `colormap` function. The function returns either a color or a floating point value. If a color is returned, the first channel of the color is obtained from the channel of the map supplied by the channel argument. Subsequent channels of the color are obtained from

```
float attenuation(float value)
{
  return (1 - pow(1-value, Dstep/Dunit));
}
```

Figure 3: The attenuation function

subsequent channels in the map. The behavior is undefined if there are not enough channels in the color map.

The `attenuation` function provides a simple way to compute a commonly used expression in volume rendering. It computes the amount light is attenuated as it travels a distance through a subvolume of a specified opacity. The function is called with a single parameter that specifies the opacity of the subvolume. It uses the class variables `Dstep` and `Dunit` to calculate this attenuation. The implementation of the attenuation function, as it would be written in the shading language is given in Figure 3. The user can implement a user-defined attenuation function if desired.

3.4 Instance variables

In the RenderMan specification, all variables have either a `uniform` or `varying` storage class. Uniform variables remain constant throughout the life of a shader, while varying variables can have different values each time the shader is invoked. In the RenderMan specification, all instance variables are assumed to be uniform variables [15]. This restricts a shader from keeping any global state between invocations. If this restriction is relaxed, and varying variables are allowed as instance variables, then global state can be maintained by assigning values to these variables. This can be important in implementing some shading techniques that require global knowledge about past calculations in a given volume. It should be noted that making use of such state information in a shader often requires a knowledge about the renderer's internal operation, and may make that shader less portable.

3.5 An example shader

A simple example of a data shader is given in Figure 4. The *threshold* data shader renders the data samples between the thresholds `mint` and `maxt` as opaque and renders all other data samples as transparent. The opaque data values are assigned the color given in the `c` instance variable. The threshold surface is shaded by using the built-in functions `ambient` and `diffuse`, which are scaled by the `ka` and `kd` instance variables respectively. On completion, the shader returns a color and an opacity to the graphics environment through the class variables `Ci` and `Oi`. To change the characteristics of the shader, new values can be supplied for the instance variables `mint, maxt, c, ka,` and `kd` when an instance is created. For more details on writing shaders in the RenderMan shading language the reader is referred to [15] or [19].

```
data threshold(float mint = 0.5, maxt = 0.6, ka = 0.5, kd = 0.5; color c = 1)
{
  float Vs = sample(P,0);
  /* If it is an opaque sample, shade it like an opaque surface. The color   */
  /* returned in Ci is computed by compositing our color behind the previous */
  /* samples along the incident ray.                                         */
  if (Vs >= mint && Vs <= maxt) {
    Oi = 1;
    Ci = Cs + c * (1-Os) * (ka*ambient() + kd*diffuse(gradient(P,0)));
  }
  else { /* Transparent sample, no change to color and opacity */
    Oi = Os;
    Ci = Cs;
  }
}
```

Figure 4: The *threshold* data shader

4 Implementation

The implementation goal of the shading system is not only to create a flexible and extensible volume rendering system, but also to provide a general purpose shading library that enables creators of other rendering systems to have access to the flexibility provided by programmable shaders. This functionality is provided through the implementation of a shader library and a shader compiler. The shader library does not provide a shading model directly, but instead provides the framework for binding externally created shaders to the geometric engine that the renderer provides. This work is based on an early implementation of a shader library, which is in turn is based on research into realistic image synthesis [3]. This section gives implementation details for the shader library and the shader compiler.

The shader library maintains a table of shaders that are available for use by a renderer. Shaders can be created and added to the system dynamically, with no changes to the renderer itself. The renderer binds a shader to a primitive at run time by asking the shader library to provide an implementation for the shader. Shaders are written in the shading language, and are compiled by the shader compiler into a form that can be loaded and executed by the library on behalf of the renderer. Once the shaders have been bound to their respective primitives, the renderer determines the points at which the shading calculations are to be performed. At each of those points the renderer invokes the appropriate shader. The shader applies its shading model and returns a result of that computation to the renderer.

Because our research focuses on the shading of objects for scientific visualization, only a subset of the RenderMan shading language is supported at this time. Priority has been placed on implementing features that are required for visualization, in particular data shaders. Currently, only the RenderMan surface shader class and our data shader class are supported. Despite this restriction, it is still necessary for our implementation to support most of the RenderMan environment and almost all of the programming functionality. It supports all of the RenderMan data types, all of the programming constructs except functions, 25% of the mapping functionality, and 85% of the built-in RenderMan function library. Of the RenderMan mapping function-

ality (texture, bump, shadow, and environment mapping), only texture mapping is currently supported in the implementation.

The remainder of this section describes the interactions between the renderer and the shader library. It describes in detail both how a renderer interacts with the shader library and how a renderer interacts with a shader. It also describes how shaders are created and compiled with the shader compiler. It is important to note that the typical user of the shader compiler is the end user of a visualization system, whereas the user of the shader library is typically the developer of a rendering system.

4.1 Library interaction

The shader library provides a framework for an extensible shading system. This framework provides the functionality that enables a rendering system to use programmable shaders. The shader library provides the renderer with a set of services. These services include: the initialization of the shader library, the binding of shader names to shader implementations, and the ability to provide the library with advanced capabilities that shaders can use. Examples of the type declarations and function interfaces to the shader library are given in Appendix A.

4.1.1 Initialization

To initialize the shading system, the renderer calls the `SLInit` function. This function takes one parameter, a null-terminated string which contains a colon-separated list of UNIX directories. These directories specify where the shader library should look to find the compiled form of any required shaders.

4.1.2 Binding shaders

To gain access to a shader, the renderer calls the `SLBindShader` function. Given a shader name (as a character string), the `SLBindShader` function returns a set of *shader interface* functions that provide the renderer with a means of interacting with that shader. These functions are described in detail in Section 4.2. When called, the `SLBindShader` function first checks the shader table to see if the desired shader already exists in the system. If the shader exists in the shader table, the function returns a success code and a set of shader interface functions to the renderer. If the shader does not exist in the shader table, `SLBindShader` searches for the shader implementation in the list of directories specified in the call to the `SLInit` function. If the shader implementation is found in one of the shader directories, the shader library loads the shader, enters it into the shader table, and returns the shader interface functions to the renderer. A shader implementation is created by the shader compiler, and must reside in a file named `shader.slo`, where `shader` is the name of the shader passed to the `SLBindShader` function. Creating a shader is described in detail in Section 4.3.

To find out which shaders have already been loaded, the shader library provides two functions: `SLNumShaders` and `SLShaderName`. The `SLNumShaders` function returns the number of shaders that are currently maintained in the shader table. The `SLShaderName` function returns

Capability Name	Token String	Capability Name	Token String
Ambient	"ambient"	Trace	"trace"
Illuminance	"illuminance"	Illuminance Init	"illuminanceinit"
Sample	"sample"	Gradient	"gradient"
Read Color Map	"readmap"	Write Color Map	"writemap"
Read Texture	"readtexture"	Write Texture	"writetexture"

Figure 5: Capabilities and their token strings

the name of a shader, given an index number between 0 and $N-1$, where N is the number of shaders returned by the `SLNumShaders` function. Once the user of the shader library has acquired the name of a shader, its shader functions can be obtained through the use of the `SLBindShader` function.

4.1.3 Capabilities

Shaders know nothing of the internal structure of the renderer which is using them. For a shader to complete a shading operation, it is necessary for the renderer to provide some geometric *capabilities*. For example, it is often necessary for a shader to know the positions and colors of the light sources in the scene. In this implementation, such information is supplied through a set of *call back functions* which are registered with the shader library after initialization. If a renderer cannot provide the required functionality, the shader library supplies a default. Capabilities are registered with the shader library through the `SLRegisterCapability` function. It is called with a textual name for the capability being registered and a pointer to a function that implements that capability. The function implementing a capability can be retrieved by using the `SLGetCapability` function. Capabilities provide a method for the renderer to extend and tailor the shader library. They also provide an easy way to extend the rendering system as more and more of the RenderMan interface is implemented.

The shader library capabilities that can be set by the renderer are: the ability to compute the ambient light falling on a point, the ability to compute the light falling on a point from a particular direction, the ability to compute the light falling on a point over a given illuminance cone, the ability to compute the sample value at a point in a volume, the ability to compute the gradient at a point in a volume, the ability to read and write color map files, and the ability to read and write texture map files. The names and the functionality of each of these capabilities is described below. The token strings that are passed to the `SLRegisterCapability` and `SLGetCapability` functions for each of the capabilities are given in Table 5. Example declarations for each capability are given in Appendix A.

Ambient

The `ambient` capability provides a shader with a means of determining the amount of ambient light that is falling on a given point. Unless a radiosity solution [20] is performed by the renderer to determine how the diffuse light interacts in the given environment, most rendering systems

will simply return a constant for the ambient light. The default action provided by the shader library for this capability is to return no ambient light (black). This capability is implemented as a function that takes a point and a color as parameters and returns the ambient color that falls on the given point in the second parameter.

Trace

The `trace` capability provides a shader with the ability to compute the color of the incident light reaching a point from a given direction. This light is usually light that is reflected from other objects in the graphics environment, and determining this color is usually implemented through ray tracing [7]. The default action provided by the shader library is to return black. The trace capability is implemented as a function that takes three parameters. The first two give the point that the incident light hits and the direction that the incident light is coming from. The third parameter is the returned color that falls on that point.

Illuminance

The `illuminance` capability provides a shader with a means of computing the amount of light that reaches a point in the illuminance cone. In the shading language, a shader uses the `illuminance` programming construct to determine this information by iterating through each light source that falls within this cone. For each light source in the the cone, two shader class variables, L and Cl are defined. These class variables are accessible for use within the block of code that is defined by the illuminance construct itself, and define the color and the direction of that light source. Thus, the illuminance construct behaves like a `while` loop that iterates over the light sources in the environment. The renderer provides this capability to the shader library by providing two call back functions, `illuminance` and `illuminanceinit`. The `illuminance` call back function takes five parameters. The first parameter defines the base of the cone over which the integration takes place. The second is a vector that gives the axis of the cone. The third parameter specifies the inclusive angle in radians. The next two parameters are the return values of the `illuminance` call back function, and give the location of the light source and the color of the light emitted from the light source. This function is called once for each light source in the defined cone, and it is responsible for keeping track of the state that it requires to process each light source defined within that cone. Each time that it is called, it either returns `TRUE` or `FALSE`. It returns `TRUE` while the information that it returns is for a light source within the cone, and returns `FALSE` when all light sources have been processed. The `illuminanceinit` function takes no parameters and returns no value. It is used to reset the state that the renderer is maintaining so that the renderer can process the light sources consistently. For example, if there are N light sources in the environment within the defined illuminance cone, a typical calling sequence for the illuminance capability would be; first a call to `illuminanceinit` to reset the renderer's internal state, followed by $N + 1$ calls to `illuminance` to obtain the light information for the N light sources. On call $N + 1$, `illuminance` returns `FALSE` indicating that all light sources have been processed. The default action provided by the shader library is to

have no light sources, so every call to `illuminance` returns `FALSE` and the block of code within the illuminance construct is never executed.

Sample

The `sample` capability provides a shader with the ability to compute the value of a sample point in a given channel of the volume data set to which the shader is bound. This capability can only be used by a data shader. The default action provided by the shader library is to return 0. The sample capability is implemented as a function that returns a floating point value. The function takes two parameters, the point at which to perform the sample and the channel of the volume data set to use. The function should return a sample value between 0 and 1. It is the responsibility of the renderer's implementation of the sample capability to map the data values within a volume into this range. It is also the responsibility of the renderer to maintain enough internal state so that the sample call back function can determine which volume it is sampling. If the sample point is outside of the volume data set or the channel given is not valid then the sample value returned is undefined.

Gradient

The `gradient` capability provides a shader with the ability to compute the local gradient at a point in a given channel within a volume data set. This capability can only be used by a data shader. The default action provided by the shader library for this capability is to return a gradient of $(0, 0, 0)$. The gradient capability is implemented as a function that takes three parameters. The first parameter gives the point at which the gradient is to be computed, while the second parameter gives the channel of the volume data set to use. The third parameter returns the calculated gradient. If the sample point falls outside the volume or the channel number is invalid then the gradient vector returned is undefined. It is the responsibility of the shader writer to insure that the sample point is inside the volume to obtain well defined behavior.

Color map

The color map capabilities provide the ability to read and write color maps to and from external files. The default actions provided by the shader library are to read and write files in a simple file format. Such a file should contain two integers, c and n, followed by $c \times n$ double precision floating point values in the interval [0,1]. The first integer in the file (c) gives the number of channels in the color map, while the second integer (n) gives the number of values in each channel. The first n floating point values represent the first channel of the color map, followed by n floating point values for the second channel, and so on. Values are stored in binary using the host machine's byte ordering, so these files may not be portable.

The `readmap` and `writemap` functions use a map data type (defined by the shader library) which contains the information required by the shader library to implement color maps. The `readmap` function takes two parameters, a file name and a pointer to a map data structure. The

file name gives the function the path name of the file to be opened. The pointer to the map data structure points to an allocated map data structure that is empty (all fields of the data structure must contain zeros). The `readmap` function opens the file and fills in the map data structure. The `writemap` function takes a pointer to a map data structure, and writes that data structure to an external file. The map data structure contains the name of the file that the color map is written to.

Texture map

The texture map capabilities provide input and output operations for the two-dimensional texture maps used by surface and data shaders. The default actions provided by the shader library read and write files in a simple file format, which contains a header that describes the texture map, followed by the texture map data itself. The header contains an integer giving the number of channels in the texture map (c), followed by two integers (s and t) giving the dimensions of each channel of the two-dimensional texture map. This data is followed by two bytes (*swrap* and *twrap*) which define how the texture map is extended outside the unit square in the s and t dimensions. There are three possible wrap modes supported by the shader library, *black*, *clamp*, and *periodic*. *Black* specifies that any reference to the texture map outside of the unit square results in the color black being returned. *Clamp* specifies that all texture coordinates outside of the unit square will be clamped to the range [0,1] and the appropriate texture value for the clamped texture coordinates is returned. *Periodic* specifies that texture coordinates outside of the unit square will be mapped into the unit square by subtracting the largest integer less than or equal to the texture coordinate. The header information in the file is followed by the texture map data, which consists of $c \times s \times t$ bytes of data, each in the range 0 to 255. The texture data is stored a channel at a time, with the first $s \times t$ bytes giving the first channel. Each channel is stored with the s dimension varying fastest in the data file.

The `readtexture` function has six parameters. The first gives the file name to read the texture map from. The next three parameters are pointers to integers, in which the function returns the number of channels and the s and t dimensions of each channel. The next two parameters are pointers to unsigned bytes in which the function returns the wrap modes for the s and t dimensions of the texture map. The function should allocate a block of memory to contain the texture map data stored as $c \times s \times t$ double precision floating point values. The data in this block of memory should be laid out in the same manner as the texture map data was laid out in the file. The function should return a pointer to this block of memory. The `writetexture` function has seven parameters, the file name to write the texture map to, the number of channels, the s and t dimensions of each channel, the wrap modes in s and t, and a pointer to a block of memory that contains the texture map data. The texture map data should be in the same format as that returned by the `readtexture` function.

4.1.4 Closing the library

To free up the resources maintained by the shader library, the `SLClose` function is used. Calling this function empties the shader table, frees up the resources required by the shader library, and resets the capabilities to their default actions. After a call to this function, the shader library may be reinitialized by calling `SLInit`.

4.2 Shader interaction

The interface between a shader and a renderer is provided through a set of five interface functions: the *constructor*, *instance_set*, *instance_get*, *print*, and *shader* functions. The *constructor* creates an instance of a shader, and returns it to the renderer. The shader instance is the object that the renderer uses to perform shading, and it is the responsibility of the renderer to keep track of which shader is to be applied to each rendering primitive. When the constructor is called, the renderer can supply values for the shader's instance variables by passing an array of instance variable *type-token-value* triples to the constructor. The variable *type* should be one of the following: float, point, color, string, or map. A *token* is a string giving the name of the variable. The *value* contains the value of the variable. A float value is passed as a pointer to a double precision floating point value. A point value is passed as an array of 3 double precision floating point values. A color is passed as an array of n double precision floating point values, where n is the number of channels for colors in the current environment (the number of channels is three by default). A string is passed as a pointer to a null-terminated string of bytes. A map is passed as a pointer to a map data structure (the map data structure is defined by the shader library). If a type-token pair matches a type-token pair of an instance variable of the shader being created, then the value is copied into an internal representation for that instance variable. If any type-token pairs do not match an instance variable, the instance of the shader is not created and an error code is returned.

The *instance_set* function sets the value of an instance variable. Its parameters are a shader and a type-token-value triple that represents the instance variable to be set. If the type-token pair matches one of the instance variable type-token pairs, then the value provided is copied into the internal representation of that instance variable. Values are provided to this function in the same form as they are provided to the *constructor* function. If a match is not made, then no change is made to the shader and an error code is returned.

The *instance_get* function gets the value of an instance variable from a shader. It is passed a shader and a type-token-value triple also. The type-token pair designates the variable to be accessed. The value is passed as a pointer to a pointer, in which the value of the instance variable is read. A pointer to a pointer is passed to the function so that it can allocate the required memory for the instance variable's value and return a copy of the variable, rather than the actual data itself. A copy of the data is returned in order to maintain the integrity of the shader library. It is the caller's responsibility to free this memory when it is no longer needed.

The *print* function is passed a shader, and prints out a textual representation of that shader to the standard output stream. The textual representation includes the name of the shader and

the type, name, and value for each of the shader's instance variables.

The *shader* function performs the shading calculation which a shader implements. The function is passed an instance of a shader and the class variables for that shader class. The shader class variables must be initialized by the renderer to define the environment in which the shader executes. The class variables provided to the shader depend on the class of the shader being invoked, and it is up to the renderer to provide the correct class variables for the given shader type. Class variables are passed to a shader through a data structure provided by the shader library. The class variables for surface and data shaders are given in Figure 1 and 2.

4.3 Creating shaders

A shader written in the shading language is compiled by the shader compiler into a form that can be used by the shader library. In this implementation, the compiler converts the shading language description of a surface or data shader into an implementation of the shader's interface functions, written in the C programming language [10]. C was chosen for the language of implementation because of its widespread use and its portability across a wide range of hardware platforms. This intermediate form is then compiled with a standard C compiler into a relocatable object module. This relocatable object module is the file that is loaded by the shader library. The shader object file is given the name of the shader being compiled with a .slo filename extension.

The shader library uses the DLD package [9] to load a shader. This package provides a means of adding and removing compiled object code to and from an executing process. The strategy of compiling a shader to native object code, executed directly by the renderer, was chosen over an interpretive approach because its execution speed is much higher. It is important to note that it is necessary to call the DLD initialization function (`dld_init`) before any calls are made to the `SLBindShader` function. If this is not done, DLD will not be able to load the shader object files successfully. The DLD initialization is done outside of the shader library so that the renderer can use the DLD library for other purposes if desired. The DLD system is part of the *GNU* software distribution under the Free Software Foundation's General Public License. Given the wide range of machines for which C compilers are available, this implementation strategy is very flexible and portable, with the major limiting factor being the number of platforms that have a DLD implementation.

4.4 Using the shader library

To use the shader library system with a renderer is quite simple. There are four major components of the system, the shader library, the shader compiler, the DLD library, and a suite of shaders. The shader compiler is a stand alone system, and converts a shader description, written in the shading language, into an object file suitable for loading by the shader library. The suite of shaders typically includes the standard RenderMan surface shaders, as well as a set of standard data shaders.

To convert a renderer from using a single shading model to using the shader library, two

steps must be taken. First, the renderer must determine a way of binding shaders to primitives. That is, the renderer must be able to identify which shaders are attached to which rendering primitives. This can be done by changing the scene description format that is accepted by the renderer to (1) describe each shader instance and (2) describe which shader instances are attached to which rendering primitives. Note that it is not necessary to create a new instance of a shader for each primitive unless the instance variables change. A new instance of a shader is created with the shader's constructor function, which is obtained with the `SLBindShader` function.

The second step is to determine when a shader should be invoked. In a typical geometric rendering system, this would be immediately after a point on the surface of a geometric primitive has been identified as requiring a shading operation. For the rendering of volumetric primitives, this would be when a sample position has been identified as requiring classification and shading. At these points in the renderer code, the renderer initializes the class variables for the shader that is attached to the rendering primitive, and invokes that shader. The shader returns the result of the shading operation in the class variables. The renderer must initialize both the shader library (`SLInit`) and the DLD library (`dld_init`), and it must be linked with both.

5 Results

To date, the shading language, the shader library, and the shader compiler have been used by two separate renderers. The initial use of the shader library was in a traditional ray-tracing renderer for geometric primitives, using surface shaders only [3]. The RenderMan standard shaders described in the RenderMan Companion (constant, matte, metal, and plastic) have been implemented [19], as well as shaders similar to the granite, marble, and wood shaders described in that text.

Data shaders were created to circumvent the limitations of having a single, monolithic shading system in a volume renderer. The volume renderer used to test data shaders was initially developed to investigate the issues involved in rendering very large data sets on massively parallel architectures [4, 12]. Data shaders were first used to reimplement the original renderer's shading model, which includes features similar to those presented by Levoy [11] and Upson and Keeler [17]. Other volume shading techniques, such as Sabella's HSV color mapping [16], Foley and Lane's value probes [6], and several experimental techniques have also been implemented using data shaders. Each of these data shaders is implemented in less than 75 lines of shading language code. Although the number of shaders that have been written thus far is small, the flexibility and power of data shaders is evident in the relative ease with which new shading and classification techniques can be implemented.

6 Future work

The driving force behind the development of the shader library is its use as a visualization tool. For that reason, most of the work invested in the shader library in the short term will be in

exploring the library's functionality in areas that are pertinent to visualization. These include the mapping of arbitrary parameters, such as heat or stress, across the surface of an object defined by geometric primitives, the exploration of the potential of data shaders for visualizing complex data sets in new and innovative ways, and the investigation of the model defined for data shaders to determine if more, less, or different information should be provided to data shaders.

In the longer term, applying data shaders to other volume rendering techniques, such as projection [5] and splatting [21], should be investigated. Another important area of research is in the production of an easy to use graphical user interface (GUI) for the system. The GUI should not only provide control over the instance variables for the shaders, but it should also present the user with an intuitive means of programming shaders. A system similar to that created by Abram and Whitted [1] would be very useful. And finally, integrating a rendering system that uses the shader library into a modular visualization environment, such as AVS [18], would enhance the functionality and flexibility of both the visualization environment and the rendering system itself.

References

[1] Abram, G., and Whitted, T., "Building Block Shaders," *ACM Computer Graphics*, **24**(4), August 1990.

[2] Cook, R., "Shade Trees," *ACM Computer Graphics*, **18**(3), July 1984.

[3] Corrie, B. "A Workbench for Realistic Image Synthesis," MSc. Thesis, Department of Computer Science, University of Victoria, Victoria, B.C., Canada, 1990.

[4] B. Corrie and P. Mackerras, "Parallel Volume Rendering and Data Coherence on the Fujitsu AP1000," Technical Report TR-CS-92-11, Department of Computer Science, Australian National University, August 1992.

[5] Drebin, R., Carpenter, L., and Hanrahan, P., "Volume Rendering," *ACM Computer Graphics*, **22**(4), July 1988.

[6] Foley, T., and Lane, D., "Multi-Valued Volumetric Visualization," *Proceedings of Visualization '91*, pp. 218-225, October 1991.

[7] Glassner, A., ed., *An Introduction to Ray Tracing*, Academic Press, Toronto, 1989.

[8] Hanrahan, P., and Lawson, J., "A Language for Shading and Lighting Calculations," *ACM Computer Graphics*, **24**(4), August 1990.

[9] Ho, W., and Olsson, R., "An Approach to Genuine Dynamic Linking," *Software—Practice and Experience*, **21**(4), April 1991.

[10] Kernighan, B., and Ritchie, D., *The C Programming Language*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[11] Levoy, M., "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, **8**(3), May 1988.

[12] Mackerras, P., and Corrie, B., "Visualizing 3-Dimensional Data on the AP1000," *Fujitsu Scientific and Technical Journal*, **29**(1), March 1993.

[13] Peachey, D., "Solid Texturing of Complex Surfaces," *ACM Computer Graphics*, **19**(3), July 1985.

[14] Perlin, K., "An Image Synthesizer," *ACM Computer Graphics*, **19**(3), July 1985.

[15] Pixar, *The RenderMan Interface — Version 3.1*, September 1989.

[16] Sabella, P., "A Rendering Algorithm for Visualizing 3D Scalar Fields," *ACM Computer Graphics*, **22**(4), July 1988.

[17] Upson, C., and Keeler, M., "V-BUFFER: Visible Volume Rendering," *ACM Computer Graphics*, **22**(4), July 1988.

[18] Upson, C., Faulhaber, T., Kamins, D., Laidlaw, D., Schlegel, D., Vroom, J., Gurwitz, R., and vanDam, A., "The Application Visualization System: A Computational Environment for Scientific Visualization," *IEEE Computer Graphics and Applications*, **9**(4), July 1989.

[19] Upstill, S., *The RenderMan Companion*, Addison Wesley, 1990.

[20] J. Wallace, M. Cohen, and D. Greenberg, "A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods," *ACM Computer Graphics*, **21**(4), July 1987.

[21] Westover, L., "Footprint Evaluation for Volume Rendering," *ACM Computer Graphics*, **24**(4), August 1990.

A Shader library interface

This appendix contains a list of the definitions provided by the shader library to a renderer. It gives the C type definitions and the function prototypes for the shader library functions and the shader interface functions. It also presents example declarations of shader library capability functions.

A.1 Type declarations

This section lists the basic type declarations that a renderer requires to use the shader library. It defines the basic data types (floats, points, colors, maps, and strings), the tokens that are passed to the shader interface functions to denote these data types, and the data types that contain the class variables for the data and surface shader classes. These type definitions are provided by the shader library in the include file `SLType.h`.

```
#define MAXCHANNELS (3)

/* Basic type definitions */

typedef double    SLT_Float;
typedef SLT_Float SLT_Point[3];
typedef SLT_Float SLT_Color[MAXCHANNELS];
typedef char*     SLT_String;
typedef struct slt_map
{
  char*      name;
  int        channels;
  int        size;
  SLT_Float* map;
} SLT_Map;

typedef void      (*SLT_VoidFunc)();
typedef SLT_Float (*SLT_FloatFunc)();
typedef int       (*SLT_IntFunc)();

/* Types of variables allowed to be passed to a shader */

typedef enum {SLFLOAT,SLCOLOR,SLPOINT,SLSTRING,SLMAP,SLINVALIDTYPE} SLType;

/* Structure that is returned by the constructor of the shader. It contains */
/* the relevant information that one might need to change and describe the  */
/* shader to the outside world.                                             */

typedef struct sl_shader_t
{
    char*   name;        /* The name of this shader             */
    char*   description; /* A description of the shader         */
    int     numvars;     /* The number of instance variables    */
    char**  names;       /* The names of the instance variables */
    SLType* types;       /* The types of the instance variables */
    void*   instance;    /* Private data for this shader        */
```

```
} SLShader;

/* The Surface Shader Variables defined by the interface */

typedef struct sl_surface_vars_t
{
    /* Colors needed for shading                                        */

    SLT_Color SLCs ;         /* Surface color                           */
    SLT_Color SLOs ;         /* Surface opacity                         */

    /* Geometry information                                             */

    SLT_Point SLP ;         /* Surface position                         */
    SLT_Point SLdPdu ;      /* Derivative of P along u                  */
    SLT_Point SLdPdv ;      /* Derivative of P along v                  */
    SLT_Point SLN ;         /* Surface normal                           */
    SLT_Point SLNg ;        /* Geometric surface normal                 */

    /* Surface parameterization information                             */

    SLT_Float SLu,SLv;      /* Surface parameters                       */
    SLT_Float SLdu,SLdv;    /* Change in the surface parameters         */
    SLT_Float SLs,SLt ;     /* Surface texture coordinates              */

    /* Incoming light information. The following are only allowed/make */
    /* sense inside the illuminance statement                           */

    SLT_Point SLL ;         /* Incoming light ray direction             */
    SLT_Color SLCl ;        /* Color of incoming light ray              */

    /* Information about the incident light ray that we are computing  */
    /* the color for.                                                   */

    SLT_Point SLE ;         /* Position of the eye                      */
    SLT_Point SLI ;         /* Incident ray direction                   */

    /* Output color and opacity for the shaded ray                      */

    SLT_Color SLCi ;        /* Ray color (output)                       */
    SLT_Color SLOi ;        /* Ray opacity (output)                     */
} SLSurfaceShaderVars;

/* The Data Shader Variables defined by the interface */

typedef struct sl_data_vars_t
{
    /* Colors needed for shading                                        */

    SLT_Color SLCs ;         /* Input color                             */
    SLT_Color SLOs ;         /* Input opacity                           */

    /* Volume geometry information                                      */

    SLT_Float SLVn ;        /* Number of sample values at location SLP */
```

```
    SLT_Float SLDs ;        /* Value of t at sample location SLP       */
    SLT_Float SLDunit;      /* Unit distance within the volume         */
    SLT_Float SLDstep;      /* Step size within the volume             */
    SLT_Float SLDin;        /* T value where the ray enters the volume */
    SLT_Float SLDout;       /* T value where the ray leaves the volume */
    SLT_Point SLP ;         /* Sample position                         */

    /* Surface parameterization information                           */

    SLT_Float SLu,SLv,SLw;     /* Parameterized sample location        */
    SLT_Float SLDu,SLDv,SLDw;  /* Volume size in u, v, and w           */

    /* Incoming light information. The following are only allowed/make */
    /* sense inside the illuminance statement                         */

    SLT_Point SLL ;         /* Incoming light ray direction            */
    SLT_Color SLCl ;        /* Color of incoming light ray             */

    /* Information about the incident light ray that we are computing  */
    /* the color for.                                                 */

    SLT_Point SLE ;         /* Origin of the incident ray              */
    SLT_Point SLI ;         /* Incident ray direction                  */

    /* Output color and opacity for the shaded ray                    */

    SLT_Color SLCi ;        /* Ray color (output)                      */
    SLT_Color SLOi ;        /* Ray opacity (output)                    */
} SLDataShaderVars;
```

A.2 Function declarations

This section contains the declarations for the functions that the shader library provides to the renderer. They are described in more detail in Section 3.3. These function prototypes are provided by the shader library in the file `SLFunctions.h`.

```
/* Initialization and shader binding functions */

extern void          SLInit(char *shaderdirs);
extern void          SLClose();
extern int           SLRegisterCapability(char *name, void (*func)());
extern SLT_VoidFunc  SLGetCapability(char *name);
extern int           SLBindShader(char *name,
                                  SLShader *(**constructor)(),
                                  int (**instanceget)(),
                                  int (**instanceset)(),
                                  void (**shade)(),
                                  void (**print)());

/* Access functions to the state of the shader table */

extern int   SLNumShaders();
extern char* SLShaderName(int index);
```

```
/* Access functions to read and write SLT_Map data structures */

#define SL_WRAP_BLACK 0
#define SL_WRAP_CLAMP 1
#define SL_WRAP_PERIODIC 2

extern int        SLWriteMap(SLT_Map *map);
extern int        SLReadMap(char *name, SLT_Map *map);
extern SLT_Float* SLReadTexture(char *file, int *channels,
                                int *ssize, int *tsize,
                                char *swrap,*twrap);
extern int        SLWriteTexture(char *file, int channels,
                                 int ssize, int tsize,
                                 char swrap, char twrap,
                                 SLT_Float *tex);
```

A.3 Shader interface function declarations

This section contains an example declaration of the shader interface functions for the `matte` surface shader. These functions would be returned to the renderer when `SLBindShader` is invoked to bind the `matte` shader. The declarations would be similar for a data shader, except that the `SLSurfaceShaderVars` passed to the shading function would be replaced by `SLDataShaderVars`. The data type `SLmatte` is an opaque data type created by the shader compiler and is not meant for use by the user of the shader library.

```
extern void      SLmatteInstancePrint(SLShader *p);
extern SLShader *SLmatteInit(int n, SLType tylist,
                             char **names,**values, int *err);
extern int       SLmatteInstanceGet(SLShader *p, char *name,
                                    SLType *type, char **value);
extern int       SLmatteInstanceSet(SLShader *p, char *name,
                                    SLType type, char *value);
extern void      SLmatteShade(SLmatte *SLp, SLSurfaceShaderVars *SLVars);
```

A.4 Capability declarations

This section provides example declaration of call back functions that the render might provide to the shader library.

```
extern void      AmbientCallBack(SLT_Color ambient);
extern int       TraceCallBack(SLT_Point point, SLT_Point direction,
                               SLT_Color color);
extern int       IlluminanceInitCallBack();
extern int       IlluminanceCallBack(SLT_Point point, SLT_Point direction,
                                     SLT_Point angle, SLT_Point L, SLT_Color Cl);
extern SLT_Float SampleCallBack(SLT_Point point, int channel);
extern void      GradientCallBack(SLT_Point point, int channel,
                                  SLT_Point gradient);
```