

Using Procedural RenderMan Shaders for Global Illumination

Philipp Slusallek, Thomas Pflaum, and Hans-Peter Seidel

Universität Erlangen, IMMD IX, Graphische Datenverarbeitung
Am Weichselgarten 9, D-91058 Erlangen, Germany
Email: {slusallek, tspflaum, seidel}@informatik.uni-erlangen.de

Abstract

Global illumination techniques like radiosity or Monte-Carlo ray-tracing are becoming standard features of rendering systems. However, there is currently no accepted interface format which supports an appropriate physically-based scene description. In this paper we present extensions to the well-known RenderMan interface, which allow for a physically based scene description and support advanced global illumination techniques. Special emphasis has been laid on the support for procedural descriptions of reflection and emission by RenderMan surface shaders. So far, they could not be used with most global illumination algorithms. The extensions have been implemented in a physically-based rendering system and are illustrated with examples.

1. Introduction

In recent years physically-based rendering and global illumination have become major topics in computer graphics. While these techniques are on the edge of being used in commercial applications, it is getting more and more important to find ways for suitable descriptions of the physical properties of a scene including the reflectance properties of surfaces and the energy distribution of light sources.

Because the reflection and emission properties of real world surfaces are too complex to be described by a fixed reflection model, a physically-based scene description interface requires a procedural approach. But arbitrary procedural descriptions of reflection or emission properties are difficult to handle for Monte-Carlo or finite element style algorithms. Both techniques require some knowledge about the procedural description in order to be implemented efficiently.

In this paper we describe extensions to the well established RenderMan interface^{1, 2, 3}, which enable the use of this interface in combination with advanced global illumination algorithms. The RenderMan Shading Language, which is part of the RenderMan interface, allows for describing emission and reflection properties by writing procedural *shaders*. The proposed extensions make necessary information about these shaders available to the global illumination algorithms.

We have chosen the RenderMan interface as the base for our work because it is still the only widely used scene description interface which is powerful and flexible enough to describe complex scenes as well as arbitrary reflectance and light source models, due to its use of procedural shaders. Unfortunately, the interface has not been designed for physically-based scene descriptions and lacks the capabilities for supporting global illumination techniques. The proposed extensions allow for overcoming these weaknesses.

The paper is organized as follows: In Section 2 we give a short overview over the global illumination aspects of rendering and the techniques available to calculate the global illumination in a scene.

In Section 3 we give a brief introduction to the RenderMan interface and discuss its shortcomings in supporting global illumination.

The following sections discuss the syntactic and semantic extensions to the RenderMan interface for supporting global illumination algorithms. Overall changes to the interface for using quantities with well-defined units are described in Section 4. In Section 5 we describe the extensions to surface and light source shaders

which allow for a physically-based representation of the reflectance and light emission models and how these shaders can make the necessary information available for an efficient implementation of global illumination algorithms. In Section 6 we discuss changes to the RenderMan interface for post-processing of rendered images by tone reproduction operators and image filters.

In Section 7 we demonstrate the usefulness of the extensions with example shaders and images of RenderMan scenes which use Monte-Carlo ray-tracing and wavelet radiosity to compute the global illumination. We finally conclude in Section 8.

2. Global Illumination

For scenes consisting of surfaces, rendering can be described mathematically as the problem to find a solution to the rendering equation⁴:

$$\begin{aligned} L(\mathbf{x}, \vec{\omega}) &= L_e(\mathbf{x}, \vec{\omega}) + L_r(\mathbf{x}, \vec{\omega}) \\ L_r(\mathbf{x}, \vec{\omega}) &= \int_{\Omega} f_r(\vec{\omega}', \mathbf{x}, \vec{\omega}) L_i(\mathbf{x}, \vec{\omega}') \cos \omega' d\omega' \end{aligned} \quad (1)$$

This equation describes the *radiance* (the power of light radiation per projected area and solid angle [$Wm^{-2}sr^{-1}$]) $L(\mathbf{x}, \vec{\omega})$ leaving a point on a surface in direction $\vec{\omega}$ as the sum of the self-emitted radiance L_e and the reflected radiance L_r . L_r is the amount of incident radiance L_i from all directions which is reflected by the surface into the outgoing direction. The fraction of radiance reflected is determined by the *bidirectional reflectance distribution function* (BRDF) $f_r(\vec{\omega}', \mathbf{x}, \vec{\omega})$. The generated image is finally given as a function of the radiance arriving on the image plane of a virtual camera from all visible surfaces.

For viewing the result, the radiance values on the image plane must be converted to pixel values for display. In a real environment this is done by the exposure control of the camera, by the non-linear response of the film, and the development process (or a CCD-element and electronic devices). For computer graphics all this is simulated by a tone reproduction operator T ⁵

$$N_p = T(L_p), \quad (2)$$

which maps the radiance L_p on the image plane to pixel values N_p in the final image. T is usually a non-linear operator.

Between the image generation and the application of the tone reproduction operator optional filters can be applied to the radiance values of the image to enhance the image quality⁶. Common to both the filters and tone reproduction operators is that these algorithms often require access to the complete rendered image.

2.1. Global Illumination Algorithms

Because the rendering equation is too complex to be solved analytically, numerical approximations must be used instead. There are basically two different approaches to calculate approximations: Monte-Carlo integration and finite element methods.

The Monte-Carlo methods (e.g. path tracing⁴) approximate the integral in the rendering equation directly, using stochastically distributed point samples in the integration domain^{7, 8}. An integral can be approximated with point samples according to the formula

$$I = \int_{\Omega} f(x) dx = \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{i=1}^N \frac{f(\xi_i)}{p(\xi_i)} \quad (3)$$

with samples ξ_i of any probability density function p which is non-zero over the integration domain. To obtain a fast approximation with low variance the probability density functions should closely resemble the shape of the integrand. This is known as *importance sampling*. For rendering, this technique results in tracing rays into the scene from the camera and recursively gathering the illumination at the surfaces of the scene by sending more stochastically distributed rays.

On the other hand, finite element methods (e.g. radiosity) represent the illumination in the scene over finite sized surface elements by functions from a certain finite dimensional function space (e.g. piecewise constant functions in traditional radiosity and wavelets in wavelet radiosity computations)^{9, 10, 11}. Finding an approximation can then be reduced to solving a large linear system describing the exchange of radiation between the elements. Finally, an image is generated by determining the value of these basis functions at all visible points.

A problem of all these global illumination techniques is that there is currently no widely used interface for describing scenes and the physical properties of the surfaces (e.g. reflection and emission). A scene description format that is widely used for classical rendering is defined by the RenderMan interface.

3. RenderMan

In this section we only give a brief overview of the features offered by the RenderMan interface as far as they are required for the remaining parts of this paper. More information is available elsewhere^{1, 2, 3}.

The RenderMan interface consists of three parts: the RenderMan interface byte-stream (RIB) protocol, which is used to describe scenes for still images or animations. A scene is described as a sequence of requests to a renderer encoded in an ASCII or a binary format. The application programmers interface (API) provides essentially the same functionality as a library of functions that can be called from an application program. Common to both, the RenderMan Shading Language offers the ability to describe certain aspects of a scene as procedural *shaders*.

3.1. Shading Language

Several different types of shaders may be used in a RenderMan scene description:

- *Surface shaders* allow for a procedural description of the reflectance properties of surfaces, i.e. the BRDF,
- *Light source shaders* describe the emission of light from point and surface light sources,
- *Image shaders* allow for image post-processing, operating on single pixels,
- *Transformation* and *displacement shaders* describe large scale and small scale surface distortions, and
- *volume shaders* are available for simple volume effects.

In this paper, we are only concerned with the first three shader types, because they have a major effect on global illumination computations. Also, we do not discuss volume effects and their integration into the RenderMan interface here, but see^{12, 13}.

The shading language is very similar to “C” but it is especially designed for shading calculations. It offers four basic data types (color, point, float, and string) and operations on them, but currently no higher level type like arrays or structures. It also includes many build-in functions which compute quantities commonly used for shading calculations (reflection vector, noise function, spline interpolation, etc.).

In the shading language, a shader is written as a C-like function, but the function arguments act more like instance variables in an object-oriented language. A shader may be instantiated multiple times with possibly different values for its arguments and is applied to surfaces in the scene. During rendering, a shader is called by the renderer and receives the rendering state (i.e. position, normal, tangent, base color, etc.) through a set of global state variables. For each shader class an appropriate set of these variables is accessible.

The shaders are then called by the renderer. Their task is to calculate one or more values and return them in other predefined global variables to the renderer. Surface shaders evaluate the BRDF and return the reflected light and the opacity of the surface. Light source shaders compute the amount of light illuminating a point in the scene from a given point on a light source. Finally, image shaders map between floating point pixel values e.g. to correct for non-linearities in the output medium.

3.2. The Imaging Pipeline

In the current definition of the RenderMan interface the pixel values computed by the renderer are passed through an imaging pipeline. This pipeline consists of four stages (see Figure 1): image filtering, exposure control, image shaders, and quantization. Except for the image shader stage, which applies an image shader

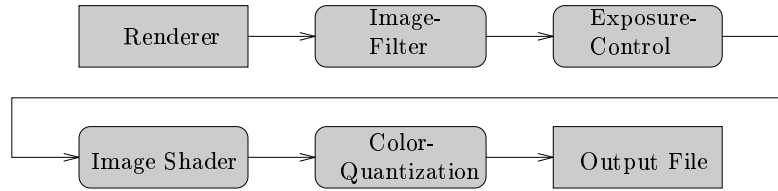


Figure 1: *The RenderMan imaging pipeline*

to each pixel, the processing model is fixed for each stages and only a few parameters can be changed (e.g. the gamma value for the exposure control).

The pixel filter stage receives as input the color and coverage information of pixels and outputs a color value. Its task is to perform filtering on the image.

The exposure control allows to scale and gamma-correct the resulting color values. Finally, after the application of the image shaders, the quantization stage applies a fixed and simple quantization algorithm to the color of each pixel, before the values are written to a file or a display device.

3.3. Previous Work

At the time the RenderMan interface was developed, global illumination was still in its early development. Furthermore, RenderMan was developed mainly for the animation market and for the Reyes rendering architecture¹⁴, which offers no support for global illumination.

The Blue Moon Rendering Tools¹⁵, the other published implementation of the RenderMan interface outside of Pixar, also supports radiosity computations. To overcome the limitations of the standard interface this system assumes special semantics for some shader variables. For instance, the variable “Kd”, which describes the diffuse reflectance factor for some predefined RenderMan shaders, is interpreted as the diffuse reflectance factor for radiosity.

While this approach works fine in most cases and is fully compatible to the current definition of the interface, it is severely limited. It is, for instance, impossible to describe a non-uniform reflectivity of a surface. The approach also fails if the shader uses a texture map. In that case the user must make sure that the supplied value for “Kd” matches the mean reflectivity of the texture.

3.4. Problems

There are several issues that need to be addressed if the RenderMan interface should be used in the context of physically-based and global illumination algorithms.

Units The RenderMan interface does not define the units for quantities used in the interface. For instance, all color and illumination quantities are unit-less and are supposed to lie between zero and one.

This has several implications: First of all, it makes all computations context dependent and may introduce inconsistencies when combining shaders that assume a different environment. This can make it difficult to reuse shader code or use shader libraries for different rendering environments.

Secondly, all specifications are relative to an implicit unit system imposed by the creator of the scene. All quantities must be converted into this implicit system before being used in a scene description. For example, it is not possible to add a light source with a particular lighting effect to a scene without knowing the overall scale factor of the other light sources.

Information Hiding The computations performed by a shader are hidden from the renderer. While this is generally a good idea because it keeps the two concepts separate, it does not allow to use many of the modern global illumination techniques. For instance, the use of Monte-Carlo algorithms in the renderer for computing the incident illumination for a surfaces shader requires information about the importance of incident illumination, in order to efficiently sample the illumination from those directions that have the greatest influence on the results of the shader.

Flexibility for Image Processing In the current interface specification image shaders are limited to operate on the value of a single pixel. However, for many techniques it is necessary to have access to the complete image. Also the filter stage of the imaging pipeline is restricted to a few filters with a fixed filter kernel. New filter kernels can only be specified in the RenderMan API and not in the RIB interface. Also a kernel is always fixed for a complete image.

Using global illumination techniques requires the use of appropriate tone mappings on the resulting pixel values for generating images of good quality. Also adaptive filtering is important for Monte-Carlo style algorithms⁶.

These problems are addressed in the following sections.

4. Units

As described above, RenderMan currently only deals with unit-less color values to describe illumination. In order to make shaders compatible with each other and with any renderer we suggest that all quantities used in the RenderMan interface are assumed to be given in international standard SI units. This means that all geometric quantities are in meters and that radiance [$Wsr^{-1}m^{-2}$] or irradiance [Wm^{-2}] is used for all illumination quantities. For instance, the emitted and reflected light returned by a light source or surface shader and the amount of incident light passed to a surface shader by the renderer are then given as a radiance value.

Note, that this is fully compatible with the current standard, where no units are assumed. It also does not limit the usefulness of the RenderMan interface because it is simple to convert from other unit system to SI units, e.g. by a simple scaling transformation.

With this specification the results of shaders are well-defined. For example, a light source shader for a point light source must scale the light flux with the square of the distance to the receiver. The effect of the lightsource is now independent of the assumed scale of the scene model.

For illumination quantities the photometric units luminance and illuminance would have been an alternative. However, they are a subjective measure and are simply scaled versions of the radiometric units, weighted with the spectral response function of a standard observer¹⁶.

Two new functions have been added to the shading language which perform the transformation from radiometric to photometric units and vice versa, respectively.

```
color photometric(color c);
color radiometric(color c);
```

These functions are required because a shader cannot know about the internal color representation.

5. Procedural Shaders

Procedural shaders are a very powerful tool for the flexible description of rendering attributes like reflection, emission, and others. RenderMan shaders compute values for a single given point on a surface and a specified reflection direction. To allow for antialiasing in the shader the renderer also passes information about the size of the sampled region on the surface.

5.1. Surface Shaders

The task of the surface shader is the computation of the integral L_r in Equation (1). To perform this integration the shader needs access to the incident illumination at this point.

This illumination is made available to the shader through the `illuminate()` construct of the Shading Language. This construct is similar to a while loop in C. The arguments of the `illuminate` statement specify the directions from which the shader wants to receive illumination in the form of a single cone (axis and angle) and the position of its apex. The construct executes the block of code following the `illuminate` statement for each illumination sample received by the renderer. Within this code additional global state variables are available which give the direction and the value of the incident illumination, again as a unit-less color variable.

This concept of procedural shaders poses several problems to global illumination algorithms:

```

surface phong(float Kd = 0.4; /* diffuse */
             float Ks = 0.6; /* specular */
             float exp = 2 ) /* exponent */
{
    point Nn = normalize(N); /* normal */
    point V = normalize(-I); /* outgoing */
    point R = reflect(V, N); /* reflected */
    point Ln; /* to light */
    color C = 0;

    illuminance(P, 2, /* pos., 2 cones */
               N, 180, 0, Kd,
               R, 180, exp, Ks )
}
{
    Ln = normalize(L); /* dir to light */
    if ((Ln.Nn) < 0) /* Ignore samples */
        continue; /* below horizon */
    C += Ks*Cl*pow(R.Ln, exp); /* specular */
    C += Kd*Cl*(Ln.Nn); /* diffuse */
}
Ci = Cs*C; /* multiply with surf. color */
Oi = Os; /* set opacity to default */
}

```

Figure 2: A simple surface shader computing the Phong reflectance model to illustrate the use of the new `illuminance()` construct. Note, that the Phong illumination model is not energy conserving for grazing angles.

5.2. Monte-Carlo Algorithms

Algorithms using Monte-Carlo techniques require that shaders are able to inform them about directionally-dependent illumination importance. The more accurate this information the better the probability density function p in (3) can be adapted to the importance distribution. This leads to less variance in the result and consequently to better performance for the same level of noise in the resulting image⁷. With the current definition of the illuminance statement the only information is, whether or not the shader is interested in illumination from a certain direction.

We propose an extension to the `illuminance` construct, by which a surface shader can provide more detailed information to the renderer about the importance of illumination. In the extended version the shader can supply several cones with a well-defined importance distribution in their interior to the renderer. This allows a Monte-Carlo renderer to choose the distribution of illumination rays accordingly. This construct can also be used for many other purposes.

The representation of the importance should be simple to compute by the shader and should allow for fast sample generation in the renderer. One description of importance which fulfills this criteria is, if we specify a set of cones with a power cosine distribution within each cone, and their relative importance.

$$p_i(\vec{\omega}) = \begin{cases} C_i \left(\left(\frac{\vec{\omega} \cdot \vec{\omega}_i}{\cos \alpha_i} \right)^{e_i} - 1 \right) & \vec{\omega} \cdot \vec{\omega}_i > \cos \alpha_i \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

where $\vec{\omega}_i$ and α_i describe the center axis and spread angle of a cone, e_i is the exponent of the power cosine distribution within cone i , and $C_i = C_i(\alpha_i, e_i)$ is a normalization factor.

This density function has a peak in the direction $\vec{\omega}_i$ and falls to zero at the boundary of the cone. The complete distribution is then given as the sum over all cones weighted with their relative importance r_i

$$p(\vec{\omega}) = \frac{1}{\sum r_i} \sum r_i p_i(\vec{\omega}). \quad (5)$$

Because the shading language allows for overloading of names, we can still use the same `illuminance` construct, which now specifies the position, the number of cones, and takes a variable number of arguments for the parameter of each cone.

```

illuminance(position, num_cones, direction1, angle1,
            exponent1, rel_importance1, direction2, angle2, ...)
{ /* calculate BRDF and integrate */ }

```

The use of a cosine distribution is motivated by the common use of cosine distributions in computer graphics, its simple calculation using a dot product, and the availability of a suitable mapping from uniform random dis-

tributions to power of cosine distributions⁸. A sample surface shader using this modified illuminance construct to implement the well known Phong reflectance model is given in Figure 2.

Note, that the importance distribution p of a shader need not exactly match the real BRDF of the surface, because the importance distribution is never explicitly used to calculate illuminance values. It simply indicates to the renderer where it is worth to calculate illumination. An inaccurate importance distribution, therefore, only results in slower convergence and thus more noise in the resulting image.

5.2.1. Using the Importance Information

The illuminance construct can then be used by the renderer to determine the direction of spawned reflection rays in a Monte-Carlo algorithm for a given point on the surface. In this case, the only task of the shader is to specify the importance distribution. The execution of the shader can be aborted, when it has executed the illuminance statement and the renderer has received the importance information. This information is then used to choose an importance weighted sample direction for the reflected ray.

A similar technique can also be used to determine the importance density for a particular reflection or transmission configuration, given an incoming and an outgoing direction. This kind of information is required by algorithms which link stochastic samples on surfaces with each other, e.g. if using bidirectional estimators¹⁷.

5.3. Finite Element Algorithms

Finite element algorithms use a set of basis functions on surface patches to represent quantities for their computations. For example, classical radiosity uses constant basis functions for the diffuse reflectance factor and the radiosity on a patch¹⁸. More advanced Galerkin radiosity algorithms use higher order basis functions like polynomials or wavelets^{19, 10, 11}.

With a set of basis functions N_i , a function f can be approximated by the representation $\tilde{f}(x) = \sum b_i N_i$. The coefficients b_i of projecting the function f on the function space spanned by ortho-normal basis set $N_i(x)$ are given by the integral over the support of f and the basis function.

$$b_i = \langle f, N_i \rangle = \int f(x) N_i(x) dx, \quad (6)$$

The problem with the combination of this approach and procedural shaders is that the shader has no knowledge about the particular basis functions used by the renderer. Therefore, it cannot compute appropriate values, e.g. mean reflectance over a surface patch for a piecewise constant basis function. Generally, a shader should not have this knowledge, because it is supposed to be a generic description of the BRDF of the surface and should not depend on the chosen set of basis functions. Another problem is that a procedural shader computes values only at a single point while a finite element patch might be an arbitrary piece of a surface, i.e. usually a triangle or quadrilateral, but more general geometries could also be used.

It is hardly possible to let a shader compute the representation of a value in a basis set defined by the renderer, because of the difficulty to write such general shaders and the problem to specify the basis functions and their domain to the shader. A more general solution is to stay with the concept of procedural shaders which compute values only at a single point, and to use numerical methods in the renderer to derive the coefficients b_i of the basis functions.

Assigning the task to compute the coefficients of the basis functions to the renderer is more suitable, because the renderer knows the characteristics of the chosen basis functions and can use appropriate numerical integration techniques. However, this technique has the disadvantage that the renderer has no knowledge about the function being integrated. This makes it difficult to decide about a set of suitable sample points for the numerical integration. An inappropriate choice of these samples can result in aliasing artifacts if the sampling is too coarse or in poor performance if it is too fine.

5.3.1. Bounding the Shader

This problem can be solved by an extension to the current definition of shaders in the RenderMan interface. In addition to the outgoing radiance, a surface shader supplies the partial derivatives and a tolerance value

to bound nearby values of the shader. These bounds can then be used by the renderer to determine or adjust the sampling for the numerical integration.

Although, it might be difficult for a shader to provide this information, we see this method as a clean and general way to allow the use of procedural shaders with finite element techniques. An alternative would be to specify a suggested sampling rate for the shader near the point for which the shader has been called. However, the presented approach is much more flexible and general.

A RenderMan surface shader already receives information about the region of the surface being sampled through the global state variables `Du` and `Dv`. Together with the parameters of the sample point (u, v) they specify an axis-aligned box in the parameter space of the surface that is being sampled. This information is currently used to perform anti-aliasing in shaders.

We extend the current interface by introducing new state variables, which can be written by the shader. Each pair of variables specifies a bound on the value of the shader over the sample domain. The shader supplies the partial derivative of the shader value and a tolerance around this approximation bounding the value of the shader. For surface shaders the variables `DCiDu/TCiDu` and `DCiDv/TCiDv` specify this bound for the value of reflected radiance `Ci` in the u and v direction of the surface, and `DCiDw/TCiDw` specifies this bound for changes over the outgoing solid angle, specified to the shader by `Dw`.

These bounds restrict the value of the shader based on the information that is available to the shader. Thus, they do not estimate changes due to changes in the incident illumination. A shader that cannot estimate the bounds over the indicated area of the surface would set the tolerance to a large value. It thereby indicates to the renderer that this surface needs fine sampling to determine an accurate approximation of the shader. This behavior is also default for any older shaders that are not writing to these new variables at all.

5.3.2. Irradiance

Many of the finite element algorithms used in global illumination today compute outgoing values (radiosity¹⁸ or radiance^{20, 11, 21}). However, these values cannot be used with a procedural surface shader, which requires information about the incoming illumination. Because these finite element algorithms have generally lost all information about the directional distribution of the incident illumination, they cannot make incident radiance values available to the shader.

Furthermore, the values computed by these algorithms are generally approximations over larger surface patches. This is accurate enough for computing the global illumination in the scene, but unacceptable for the generation of the final image, where much more detailed information about the BRDF of a surface is required (e.g. high resolution textures).

Both problems can again be solved by a small modification to the definition of surface shaders and the values returned by the finite element algorithms.

In addition to the radiance values which are received by the shader through the `illuminate()` construct, the renderer makes an irradiance value E [Wm^{-2}] available to the shader. The irradiance $E(\mathbf{x}, \vec{\omega})$ may also depend on the *outgoing* direction $\vec{\omega}$ to allow for algorithms which compute directional dependent outgoing values²⁰. Thus, the definition of E as supplied by the renderer is “the irradiance at \mathbf{x} resulting in the proper outgoing radiance in direction ω under diffuse reflection”.

For instance, this irradiance value can easily be obtained for standard radiosity by dividing by the reflectance coefficient of the surface which has been used to derive this value. The shader can then derive the radiance reflected into the outgoing direction due to this irradiance term and can take into account more detailed information (e.g. textures).

In essence, our model of the interaction of light with surfaces as used in the RenderMan surface shaders computes the reflected radiance according to the formula (see Figure 3)

$$L_r(\mathbf{x}, \vec{\omega}) = f_{ir}(\mathbf{x}, \vec{\omega})E(\mathbf{x}, \vec{\omega}) + \int_{\Omega} f_r(\vec{\omega}', \mathbf{x}, \vec{\omega}) \hat{L}_i(\mathbf{x}, \vec{\omega}') \cos \omega' d\vec{\omega}'. \quad (7)$$

Where L_r is the reflected radiance, f_{ir} describes the reflectance of the surface in respect to the irradiance term and the outgoing direction $\vec{\omega}_i$ in units of $[sr^{-1}]$, and $E(\mathbf{x}, \vec{\omega})$ is the “directional” irradiance at \mathbf{x} . $\hat{L}_i(\mathbf{x}, \vec{\omega}')$ is the incident radiance without the illumination already accounted for by $E(\mathbf{x}, \vec{\omega})$.

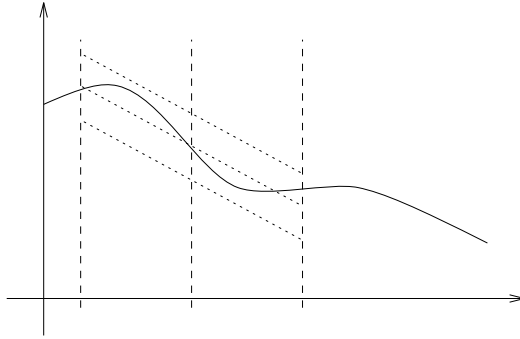


Figure 3: *Bounding the value of a shader near the evaluation point by specifying the partial derivatives and a tolerance value.*

If, for example, traditional radiosity is used for the illumination calculation, $\hat{L}_i(\mathbf{x}, \vec{\omega}')$ is zero and $E(\mathbf{x}, \vec{\omega})$ does not depend on $\vec{\omega}$. E would depend on $\vec{\omega}$ when outgoing radiance is computed by a finite element algorithm¹¹.

The irradiance E is made available to the surface shader through the new function

```
color irradiance(point x, point omega);
```

This function should be used instead of the current `ambient()` function, which does not allow for a directional dependence of the returned value.

The above extensions are fully backward compatible with the current use of surface shaders in the RenderMan interface.

5.4. Light Source Shaders

In a classical rendering system there are three different classes of light sources: The ambient light source which describes the amount of overall illumination in the scene, local light sources which have a specific location in the scene (point and area light sources) and directional light sources which have no position and send light only in certain directions (e.g. parallel light from the sun). Each of these light source types can be described in the shading language.

The task of light source shaders is to describe the incident radiance distribution at a receiving point. The class of the light source shader is determined by its usage of the `illuminate()` construct to define a local light source, of the `solar()` construct to define a distant light source, or non of the two for an ambient light source.

In a physically-based rendering process there is usually no ambient light source because the ambient term is calculated as part of the global illumination process. In addition area light sources are often used to provide a more realistic simulation of real extended light sources. For our purposes area light sources can be handled similar to point light sources.

The `illuminate` and the `solar` construct in light source shaders are very similar to the `illuminate` construct of surface shaders. They also specify a cone of directions, into which light is emitted by the light source. The cone may be used to cull points outside the cone from the illumination calculations.

The problems encountered when implementing common global illumination algorithms are very similar to those for surface shaders. The same solution techniques can be used by extending these language constructs and providing the ability to specify an importance distribution for emission by a set of cones and the associated power cosine distribution from Equations (5) and (4). This distribution can then be used to determine the direction of outgoing sample rays or to calculate the probability of choosing a given direction.

Light source shaders make similar bounding information about the emitted radiance available to the renderer.

These bounds on the illumination are available for the changes on the light source surface and in the outgoing direction through additional global state variables, similar to surface shaders.

6. The Imaging Pipeline

The current definition of the imaging pipeline of RenderMan must also be modified in order to perform post-processing on the image in physically well-defined units. These extensions are necessary to support the current state-of-the-art in physically-based image synthesis. This includes tone-mapping operators^{5, 22} and image filtering to reduce artifacts of the particular global illumination algorithm⁶.

In the current definition of the RenderMan interface the imaging pipeline (see Figure 1) can only execute a set of standard filter algorithms. Also, image shaders currently operate on the already filtered and gamma corrected pixel values and image shaders operate on the value of a single pixel at a time. These features are too restrictive for an implementation of the mentioned algorithms.

6.1. Filtering

Filtering is a major step for calculating the final value of the pixels, especially in the case of Monte-Carlo sampling. A good interface for global illumination algorithms should therefore support the flexible specification of such filters.

This filtering process is usually a multi stage process: First, the energy distribution may be enhanced by reducing sampling artifacts through an adaptive filtering strategy⁶. While this first step is most useful for Monte-Carlo style algorithms, a tone reproduction step which converts the radiance values into pixel values for display is unavoidable. For rendering scenes with physically-based light sources descriptions and reflection functions, the large dynamic range of the input distribution must be matched to the rather small dynamic range displayable by a CRT or a color printer. There are different approaches to solve this problem. Some include simple linear scaling²², others use more complex non-linear models⁵.

These filters receive the radiance distribution at the pixels on the image plane as their input and their task is to convert the radiance values to pixel values suitable for display. By making use of the Shading Language we can introduce new *filter shaders* in order to flexibly support image post-processing, such as filtering and tone-mapping algorithms, in the RenderMan interface.

To support multiple layers of filtering we allow for a stack of filter shaders to perform the filtering. Any number of filter shaders can be pushed on a filter stack. The filter first pushed receives the output of the renderer and its output is the input of the next filter in the stack. A shader is pushed on the stack through the new request

```
PushFilter "filter_name" ...
```

The output of the last shader should be values in the unit range. It is further processed by the standard RenderMan imaging pipeline (i.e. gamma correction, standard image shaders, and quantization), before it is sent to the image file or device. Care must be taken to match the units of the output of one shader to the required input of the next shader. To allow for a large flexibility in this area, no restrictions are imposed by the interface.

6.2. Extensions to the Shading Language

To allow for the specification of new filter functions in the Shading Language, a new RenderMan shader class, *filter shaders*, is introduced together with the concept of *maps*. In contrast to the old image shaders, which are called once for each pixel in the image, a filter shader is called once for the complete image. The image is made available to the shader through a predefined map, instead of through global state variables. This is necessary because a filter shader may require access to other pixel values or even the complete image to accomplish its task (e.g. it must compute the mean radiance in the image for computing an adaptive tone-mapping operator⁵).

The size of the image is passed to the shader in the global state variables `xsize` and `ysize`. Access to the samples of the image is available through a named map, which hold the pixel values and new map access functions.

```
color image(string map; float x, y, ...);  
void setimage(string map; float x, y; color c, ...);
```

The function `image()` returns the pixel value of a map and the function `setimage()` allows to change pixels in the map. Image shaders receive their input in the predefined map “Cs” and store their result in the map “Ci”.

Although a renderer often works in radiometric units, it is common for a image shader to perform some computations in respect to luminance values. In addition to the functions `photometric()` and `radiometric()` efficient transformations are made available through new functions, which integrate over the spectrum and return the total luminance or radiance values.

```
float totalluminance(color c);  
float totalradiance(color c);
```

Other channels which are available to the filter shader are “Oi”, which contains opacity information, and “Z”, which stores the depth at a pixel location. Implementation dependent channels might also be made available to the shader.

This concept of filter shaders allows for a very flexible handling of image post-processing for global illumination algorithms. It takes advantage of the available Shading Language and extends the usage of shaders for this task.

7. Examples

In this section we present results of implementing the proposed extensions to procedural shaders. The extensions have been integrated into the Vision system, a physically-based rendering architecture build upon an object-oriented design, which integrates both traditional and most global illumination techniques into a common rendering architecture^{12, 23}. The Vision system uses RenderMan as its main interface and implements the complete standard including an efficient native shading language compiler²⁴. The problems of supporting procedural shaders with global illumination techniques in the Vision architecture have motivated the work presented in this paper.

7.1. Monte-Carlo Ray-Tracing

Figure 4 (see color section) shows a scene where a desk lamp illuminates the desk and the back of a blue mug. The rough specular metal plate on the table reflects the illumination on the mug and the desk. The image has been calculated using standard Monte-Carlo path tracing⁴.

The scene is described by an extended RIB file with all surface and light source shaders given in the Shading Language using the proposed extensions. The desk lamp consists of a small area light source placed in the back of an opaque rectangular lampshield. The shader for the area light source emits light only on one side of the surface with a cosine distribution centered around the surface normal. The surface of the metal plate indicates a high importance for illumination from the mirror direction using a small cone with large relative probability in addition to a cone indicating low importance for illumination from the entire hemisphere for a diffuse reflection term.

7.2. Energy Preserving Filters

While Monte-Carlo methods are known to generate statistically correct energy distributions they are also known for being slow. The resulting images often still contain a high level of variance or visible noise. One way to reduce the remaining noise was proposed in⁶, where a form of adaptive filtering of the radiance distribution of the image is suggested. The filter is designed to reduce the noise by first identifying noisy pixels and then spreading the energy of this pixel to its neighbors.

An example of the effect of an energy preserving filter is given in Figure 5. The image on the left is the un-filtered Monte-Carlo image, while the image on the right uses the energy-preserving filter from⁶.

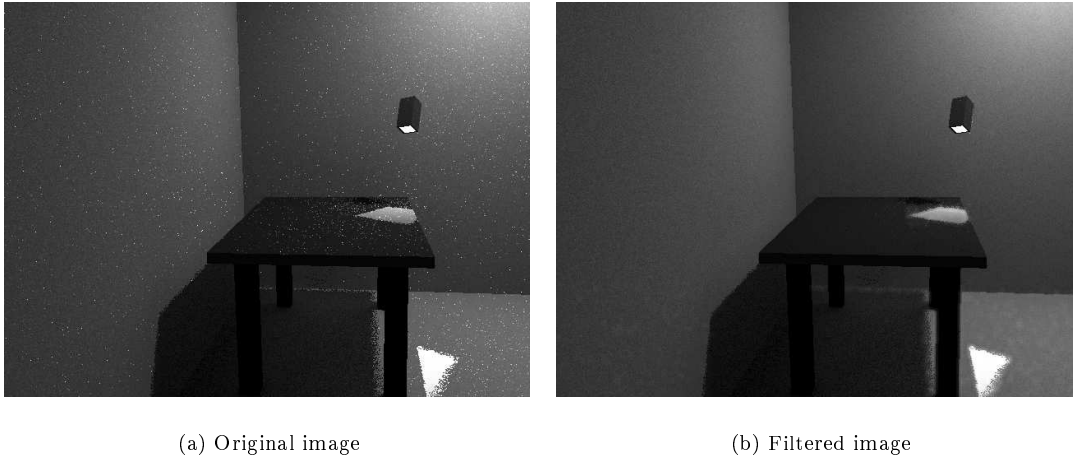


Figure 5: *Two images showing the effects of applying an energy preserving filter to the result of a Monte-Carlo solution to global illumination.*

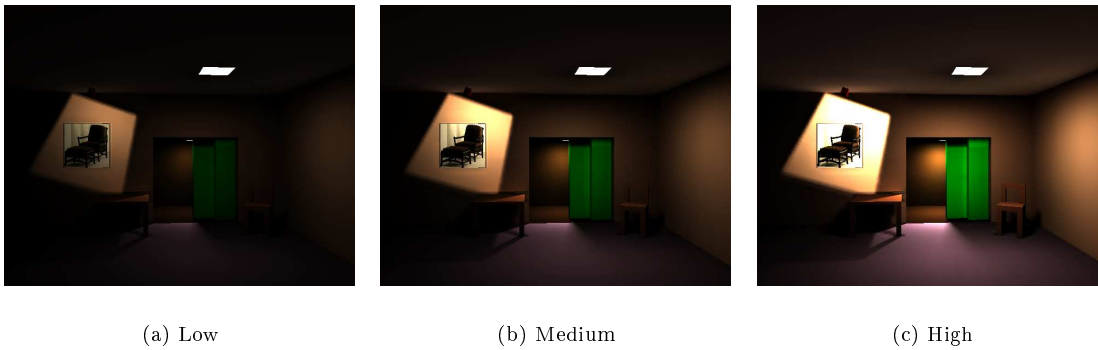


Figure 6: *The tone reproduction operator from Tumblin and Rushmeier⁵ for the same scene with a low, medium, and high illumination.*

7.3. Tone Reproduction

The simple tone reproduction operator²² is given as a RenderMan filter shader in Figure 7 to outline the basic structure of a filter shader. Figure 6 shows the effects of applying the tone mapping operator from Tumblin and Rushmeier⁵ to three different illuminations of the same scene. The illumination in the images increases by a factor of two from left to right.

8. Conclusion

In this paper we have discussed the problems of using procedural shaders together with advanced global illumination techniques. To overcome these problems, we have suggested extensions to the RenderMan interface, especially to the RenderMan Shading Language. These extensions allow for using procedural shaders both in Monte-Carlo as well as in finite element based global illumination techniques.

Furthermore, we have presented extensions, which allow the flexible specification of image post-processing algorithms using the available support of the Shading Language. This image post-processing is required for obtaining high quality images from modern global illumination algorithms.

```

filter ward94(float L_dmax = 100)
{
    float x, y, scale, L_wa;
    float sum_luminance;
    color rad;

    /* Compute the average luminance */
    for(x = 0; x < xsize; x += 1)
        for(y = 0; y < ysize; y += 1) {
            rad = image("Cs", x, y);
            sum_luminance += totalluminance(rad);
        }
    L_wa = sum_luminance / (xsize * ysize);

    /* Calculate the scaling factor*/
    scale = 1.219 + pow(L_dmax/2, 0.4);
    scale /= 1.219 + pow(L_wa, 0.4);
    scale = 1/L_dmax * pow(scale, 2.5);

    /* scale the pixel values */
    for(x = 0; x < xsize; x += 1)
        for(y = 0; y < ysize; y += 1) {
            rad = image("Cs", x, y);
            setimage("Ci", x, y, rad*scale);
        }
}

```

Figure 7: A global image shader implementing the tone-reproduction operator by Ward²². The adaptation level of the viewer is derived from the average luminance of the image. The parameter `L_dmax` describes the maximum display luminance measured in $[cd\ m^{-2}]$.

The extensions presented in this paper are mostly backward compatible changes to the existing RenderMan standard and can be implemented without major changes to the Shading Language and its compiler. With these changes, the RenderMan interface can now be used to describe scenes for rendering with traditional as well as more advanced physically-based rendering techniques, based on the same scene description.

References

1. Pixar, *The RenderMan Interface, Version 3.1*. Pixar, San Rafael, CA, (September 1989).
2. S. Upstill, *The RenderMan Companion*. Addison Wesley, (1990).
3. P. Hanrahan and J. Lawson, "A language for shading and lighting calculation", *Computer Graphics (SIGGRAPH '90 Proceedings)*, **24**(4), pp. 289–298 (1990).
4. J. T. Kajiya, "The rendering equation", *Computer Graphics (SIGGRAPH '86 Proceedings)*, **20**(4), pp. 143–150 (1986).
5. J. Tumblin and H. E. Rushmeier, "Tone reproduction for realistic computer generated images", *IEEE Computer Graphics & Applications*, **13**(6), pp. 42–48 (1993).
6. H. E. Rushmeier and G. J. Ward, "Energy preserving non-linear filter", *Computer Graphics (SIGGRAPH '94 Proceedings)*, pp. 131–138 (1994).
7. J. Arvo and D. Kirk, "Particle transport and image synthesis", *Computer Graphics (SIGGRAPH '90 Proceedings)*, **24**(4), pp. 63–66 (1990).
8. P. Shirley, "Monte carlo simulation and integration", in *Global Illumination (SIGGRAPH '93 Course Notes 42)* (P. Heckbert, ed.), pp. 9.1–9.23, (1993).
9. M. F. Cohen and J. R. Wallace, *Radiosity and Realistic Image Synthesis*. Academic Press, (1993).
10. S. J. Gortler, P. Schröder, M. Cohen, and P. M. Hanrahan, "Wavelet radiosity", *Computer Graphics (SIGGRAPH '93 Proceedings)*, **27**, pp. 221–230 (1993).
11. P. Schröder, *Wavelet Algorithms for Illumination Computations*. PhD thesis, Princeton University, (November 1994).
12. P. Slusallek, *Vision – An Architecture for Physically Based Rendering*. PhD thesis, University of Erlangen, IMMD IX, Computer Graphics Group, (April 1995).
13. B. Corrie and P. Mackerras, "Data shader language and interface specification", Tech. Rep. TR-CS-93-02, The Australian National University, Computer Science Department, (1993).

14. R. Cook, L. Carpenter, and E. Catmull, "The Reyes image rendering architecture", *Computer Graphics (SIGGRAPH '87 Proceedings)*, **21**(4), pp. 95–102 (1987).
15. L. I. Gritz, *Blue Moon Rendering Tools: User Guide*. Blue Moon Systems, (September 1994).
16. R. Hall, *Illumination and Color in Computer Generated Imagery*. Monographs in Visual Computing, Springer-Verlag, (1988).
17. E. Veach and L. Guibas, "Bidirectional estimators for light transport", in *Fifth EUROGRAPHICS Workshop on Rendering*, (Darmstadt), pp. 147–162, (June 1994).
18. M. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg, "A progressive refinement approach to fast radiosity image generation", *Computer Graphics (SIGGRAPH '88 Proceedings)*, **22**(4), pp. 75–84 (1988).
19. H. R. Zatz, "Galerkin radiosity: A higher order solution method for global illumination", *Computer Graphics (SIGGRAPH '93 Proceedings)*, pp. 213–220 (1993).
20. F. X. Sillion, J. R. Arvo, S. H. Westin, and D. P. Greenberg, "A global illumination solution for general reflectance distributions", *Computer Graphics (SIGGRAPH '91 Proceedings)*, **25**(4), pp. 187–196 (1991).
21. P. H. Christensen, E. J. Stollnitz, D. Salesin, and T. D. DeRose, "Wavelet radiance", in *Fifth EUROGRAPHICS Workshop on Rendering*, (Darmstadt), pp. 287–301, (June 1994).
22. G. Ward, "A contrast-based scalefactor for luminance display", in *Graphic Gems IV* (P. S. Heckbert, ed.), ch. 7.2, pp. 415–421, Academic Press, (1994).
23. P. Slusallek and H.-P. Seidel, "Vision: An architecture for global illumination calculations", *IEEE Transactions on Visualization and Computer Graphics*, **1**(1), pp. 77–96 (1995).
24. P. Slusallek, T. Pflaum, and H.-P. Seidel, "Implementing RenderMan - practice, problems, and enhancements", *Computer Graphics Forum (EUROGRAPHICS '94 Proceedings)*, **13**(3), pp. 443–454 (1994).