Reference: Gritz, Larry and James K. Hahn, "BMRT: A Global Illumination Implementation of the RenderMan Standard," *Journal of Graphics Tools*, Vol. 1, No. 3, pp. 29-47 (1996).

BMRT: A Global Illumination Implementation of the RenderMan Standard

Larry Gritz[†] *and James K. Hahn*

The George Washington University[*]

ABSTRACT

The RenderMan Interface specification proposed by Pixar is a standard for communication between modeling software and rendering software or devices. This standard has proven very powerful and is extremely popular in production work. Although the standard itself claims not to specify a rendering algorithm, people have speculated RenderMan and global illumination are mutually incompatible.

We have implemented a rendering system which fully adheres to the RenderMan Interface and uses global illumination algorithms. Specifically, this implementation supports progressive refinement radiosity and distribution ray tracing in a two-pass approach. This rendering system is widely distributed, very popular, and has been used in production (three properties usually not found in global illumination renderers). We discuss how we overcame problems in mating global illumination algorithms with the RenderMan standard, and make recommendations for future versions of the standard to better accommodate such algorithms. We also present a summary of important lessons we learned by creating and distributing this tool.

1. Introduction

The RenderMan® Interface was designed by Pixar to be a standard communication protocol between modeling/animation software and rendering software or devices [14][10]. RenderMan gives a means of describing a photorealistic 3-D scene in terms of what should be rendered, but without dictating which algorithms or rendering methods should be used.

The RenderMan Interface has two bindings: a procedural interface consisting of C language function calls, and an ASCII or binary metafile binding known as RenderMan Interface Bytestream (RIB). There exists a nearly one-to-one correspondence between RIB requests and procedural API calls. Individual RenderMan calls either (1) set the options, attributes, and transformations which comprise the graphics state; or (2) declare geometric primitives, which are bound to the current graphics state at the time of their declaration. The exact nature of the API calls and RIB protocol are well documented elsewhere [14][1], and will not be described further in this paper.

RenderMan allows arbitrarily complex descriptions of surface appearances (including geometric displacements), illumination distributions from light sources, and attenuation by volumes in RenderMan Shading Language (SL) [9][2]. This powerful ability allows the user to extend the surface and light descriptions in a way which is more flexible than generally allowed by rendering systems.

The RenderMan standard does not dictate which rendering algorithms a particular implementation should use. Minimally, an implementation must support all of the geometric primitives (including polygons and polyhedra, bilinear and bicubic patches and patch meshes, NURBS, and quadrics), perform hidden surface removal, provide antialiasing and filtering, and support the hierarchical graphics state. In addition, optional capabilities may be supported in a particular implementation, including CSG, motion blur, depth of field, ray tracing, radiosity, displacements, texture and environment mapping, area light sources, and programmable shading in Shading Language. Implementations are not expected to support optional capabilities which are not possible or are impractical given their rendering algorithms.

Until recently the only available implementation of the RI standard has been Pixar's PhotoRealistic RenderMan (*PRMan*). This implementation has become very popular in the production community, arguably becoming the *de facto* standard in rendering software for feature film work. Several other proprietary implementations of the RI standard no doubt exist, but remain undocumented and are of unknown levels of compliance to the standard (an exception is [13]). *PRMan* is based on the REYES algorithm [6]. Indeed, it is apparent that this implementation and the standard itself were developed hand-in-hand. Though it claims to be independent of any particular rendering algorithm, many aspects of the standard seem to rely

[†] Author's current address: Pixar, 1001 W. Cutting Blvd., Richmond CA 94804
[*] Dept. of EE&CS, 801 22nd St. NW, Washington DC 20052.
Email: lg@pixar.com, hahn@seas.gwu.edu

on the REYES or similar micropolygon-based scanline algorithms, perhaps contributing to the lack of other available implementations. This led people to speculate that the RenderMan standard and global illumination were mutually incompatible.

This paper describes a full implementation of the RenderMan standard, publicly available and known as the Blue Moon Rendering Tools (BMRT). BMRT uses global illumination algorithms, supporting both ray tracing and radiosity in a two-pass technique similar to [15]. This design presented many challenges, both in attempting to support all of the functionality of RenderMan without using REYES, and in trying to use the RenderMan interface as the basis for a global illumination renderer. We will describe our system, discussing these challenges and our solutions.

2. System Overview

Our renderer implements a two-pass radiosity / distribution ray tracing scheme. The basic algorithm may be outlined as follows:

1. The input RIB stream, which specifies the scene to be rendered, is parsed. Each RIB directive corresponds to a particular RenderMan procedural API call. These calls change options or attributes, alter transformations, or declare geometry (which binds to the current attribute list).

2. When the **WorldEnd** directive is reached, indicating that all information necessary to render a particular frame has been transmitted, an automatic bounding hierarchy is constructed to speed up the ray casting, and the rendering calculations begin.

3. If radiosity calculations are desired, the geometry list is meshed. Our system implements a progressive refinement radiosity solution which uses ray casting to calculate form factors and distribute energy directly to element vertices [5][16]. High radiosity gradients may cause geometry to be progressively diced into finer elements.

4. The first energy distributed by the progressive refinement radiosity is the energy which comes from the explicit light sources (i.e. those defined with **RiLightSource** and **RiAreaLightSource**). For each element vertex, the appropriate light source shader is invoked in order to calculate how much energy reaches that element vertex. Several samples may be taken from randomly selected points on the geometry comprising each area light source to minimize the error.

5. After the light source list is exhausted, redistribution of reflected energy is performed, including geometry which had been designated as emissive. At each step, the patch with the highest unshot energy has its energy redistributed into the environment. The progressive refinement stops when either a specified maximum number of progressive refinement steps have been reached, or the remaining unshot energy in the entire environment is below a certain threshold. We employ a separate accounting of radiosity coming from direct sources (lights) and indirect sources (reflection from other geometry). This allows us to recalculate the direct component again on the second pass, which affords many advantages [12][8].

6. Once the radiosity pass is completed, distribution ray tracing [7] of the scene is performed. Prior to rendering time, the shader compiler parses the Shading Language code and generates an assembly-like code for a simulated stack-based computer. This allows for fast interpretation of compiled and optimized shader code in a machine-independent manner. When the closest intersection to a ray is found, the displacement shader (if any) and the surface shader for that object are interpreted, resulting in the surface color and opacity of the visible object. Though there is some overhead associated with interpreting the shader (as opposed to having it compiled into the renderer itself), this is minor compared to time spent calculating ray-object intersections, calculating noise values, etc., and affords extra flexibility when designing and implementing shaders.

   When performing shading calculations on the ray tracing pass, calls to the SL function *ambient*() return the indirect illumination component of the radiosity calculations performed earlier. The values are returned from the radiosity mesh corresponding to the parametric position at which the ray intersected the object.

   The renderer also attempts to handle specular-to-diffuse illumination, such as light bouncing off of a mirror. The mechanism for this calculation is documented in [8]. The SL built-in functions which integrate the contributions of light sources[1], automatically compute and account for this mode of energy transfer.

3. Ray Tracing / Radiosity Implementation of RenderMan

Certain optional features of the RenderMan interface are easy to perform automatically using distribution ray tracing [7]. These include solid modeling (CSG), bump mapping, motion blur, and depth of field. Other features of RenderMan were obviously put in specifically to accommodate scanline renderers which were not capable of ray tracing, including environment mapping and shadow depth mapping. While there is no reason why these cannot be used by a ray tracer, the functionality of these features (reflection/refraction and shadowing) are generally implicit in ray tracing. Therefore we support both explicit calls to these "fake" functions, and also automatic calculation of these effects.

Several RenderMan features do not map in obvious ways to our radiosity/ray tracing paradigm. This section will cover some of these and explain how we approached these problems.

3.1. Interpretation of shaders

When it is determined which geometry intersects a ray at the closest point, a "ShaderVariables" record is prepared to

[1] The semantics of SL mechanisms for integrating light source contributions, including the built-in functions *diffuse*() and *specular*(), is beyond the scope of this paper, but further information may be found in [10][14].

contain the variables accessible to the shaders which determine what the point looks like (surface position, normal, derivatives, shading coordinates, etc.). The object class definitions for each geometric primitive type include methods for computing these quantities. When each shader is compiled, a list is automatically made that specifies exactly which shader variables are needed by the shader (for example, maybe it needs **N**, **s**, and **t**, but not **dPdu**). To avoid unnecessary computation, only the shader variables which are needed will be computed for a particular shading calculation. The shader interpreter is invoked with these variables.

3.2. SL Differential and Area Operators

Among the variables accessible by the shaders are the **du** and **dv** values. These are defined as the distance in u-v parametric space of the geometric primitive between shading samples and are the basis for differential and area operators in SL. Table 1 lists some differential quantities commonly used in SL. For a REYES implementation, these are trivial to compute, since they are simply the differences of the quantities evaluated at adjacent micropolygons. Ray tracing is fundamentally a point sampling process in screen space and does not use a regular discretization of the geometric primitives, so it is not immediately obvious how these values should be computed.

One may reasonably expect that a ray tracer should not need to compute these quantities. But since these variables are essential for analytical antialiasing of textures, we deemed it important to support this facility. We approached this problem as follows: Suppose **P** represents the point being sampled. Let $\mathbf{P_z}$ represent the depth of **P** in eye (camera) space. Then we can estimate the world space distance between screen samples for this depth as:

$$d_{est} = d_s \mathbf{P}_z$$

where d_s is the distance between screen space samples. This geometric relationship is shown in Figure 1. We are simply computing the distance at that depth which should map to adjacent screen space samples. This is really an estimate for |**dPdu***__du__| and/or |**dPdv***__dv__|, which are the changes in surface position between adjacent screen samples (**dPdu** and **dPdv** are the partial derivatives of the surface). By inverting, we can solve for estimates of **du** and **dv**:

$$\mathbf{du} = \frac{d_{est}}{|\mathbf{dPdu}|} k_u \qquad \mathbf{dv} = \frac{d_{est}}{|\mathbf{dPdv}|} k_v$$

Finally, the orientation of the surface is taken into consideration with the additional k_u and k_v terms. Comparing the view direction to the partial surface derivatives serves to enlarge our estimates of **du** and/or **dv** when we are viewing the surface at a glancing angle. The orientation terms are:

$$k_u = \frac{1}{\sqrt{1 - \left(\frac{\mathbf{I} \cdot \mathbf{dPdu}}{|\mathbf{I}||\mathbf{dPdu}|}\right)^2}} \qquad k_v = \frac{1}{\sqrt{1 - \left(\frac{\mathbf{I} \cdot \mathbf{dPdv}}{|\mathbf{I}||\mathbf{dPdv}|}\right)^2}}$$

Each geometric primitive class has methods for calculating its partial surface position derivatives, **dPdu** and **dPdv**, for a particular surface point.

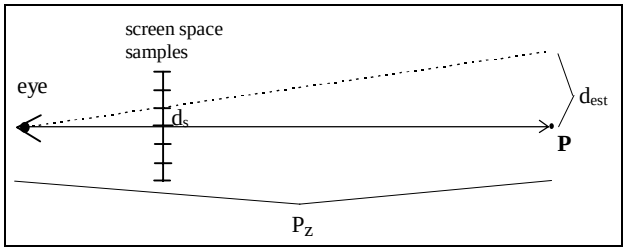

Figure 1: Calculating distance at intersection which corresponds to adjacent screen samples.

Many of the functions in Table 1 take arbitrary expressions as arguments. It is impractical to attempt to solve these analytically at compile time, so we generate ShaderVariables records for two *auxiliary points* on surface *S*, $P_{+u} = S(\mathbf{u}+\mathbf{du},\mathbf{v})$ and $P_{+v} = S(\mathbf{u},\mathbf{v}+\mathbf{dv})$, which approximate the step sizes between screen space samples. The shader is evaluated at

Table 1: Some differential quantities. SL built-in variables are in boldface.

RenderMan Quantity	*Interpretation*
du, **dv**	Change in ***u*** and ***v*** geometric parameters between adjacent shading samples.
dPdu, dPdv	The partial derivatives of the surface at the point being shaded, i.e. ∂**P**/∂***u*** and ∂**P**/∂***v***.
Du(*x*), Dv(*x*)	Derivative of expression *x* with respect to parametric ***u*** and ***v***.
Du(*x*) * **du**	Estimated change in expression *x* between adjacent shading samples (along *u* direction).
area(*p*)	Estimated differential surface area ≡ length(Du(*p*)***du** ^ Dv(*p*)***dv**) (note: ^ is the SL operator indicating cross product)
sqrt(area(*p*))	Estimated change in point value *p* between adjacent shading samples.
texture(*name*, s, t)	Filtered texture area lookup at coordinates (s,t), implicitly takes into account the change of s,t between adjacent shading samples.
calculatenormal(*p*)	≡ Du(*p*)^Dv(*p*). If *p* is the displaced surface position, this returns the new normal.

these two points first, caching values passed to Du(), Dv(), area(), and calculatenormal(). Then when the shader is called for **P** (the point actually being shaded), these functions return the differences between the expression at **P** and its value at the auxiliary points. For example,

$$Du(P)\big|_{u,v} = (P\big|_{u+du,v} - P\big|_{u,v}) / \mathbf{du}.$$

In order to prevent evaluating the entire shader three times for each surface point, we utilize several shortcuts. First, area operators on some simple expressions can be determined analytically at compile time. For example, the common construct Du(s) is the rate of change of shading coordinate s with respect to geometric parameter u. This value is fixed for a particular primitive, so the SL compiler recognizes this and can substitute a reference to this value instead of invoking the shader again at (u+du,v). Second, the compiler recognizes which parts of the code contribute to the expressions sent to area operators. When invoking the shader for the auxiliary points P_{+u} and P_{+v}, the shader terminates immediately after the last area operator is executed. Thus, code which is after the last area statement (and therefore cannot contribute to any differential quantities at the main shading location) will not be executed. Finally, certain operators which are expensive and unlikely to have their results used in area expressions are short circuited. For example, *diffuse*() simply returns 0 for auxiliary points without walking the light list or computing shadows. These shortcuts cause the area operators to be evaluated correctly with very little overhead.

3.3. Displacements

An important feature of RenderMan is the ability to use *displacement shaders*. Similar to bump mapping [3], displacement mapping actually moves surface points as well as changing normals for shading. This allows for very realistic detailing on surfaces and an alternate way to model fine geometric detail while keeping the underlying geometric primitives simple.

For a REYES or similar system, displacement shaders can be used to alter the micropolygon vertices prior to hidden surface elimination. Unfortunately, there does not exist a simple method for handling displacements for a ray tracer (short of dicing the primitives and ray tracing the micropolygons themselves, which is very expensive in both time and memory). For our implementation, we make the assumption that all displacements will be small, evaluate the displacement shader at points P_{+u} and P_{+v} in a manner similar to the way we handle area operators. The displaced **P** variable is compared at all three points and a new normal is computed:

$$\mathbf{N}' = (\mathbf{P}_{+u} - \mathbf{P}) \times (\mathbf{P}_{+v} - \mathbf{P})$$

This performs bump mapping based on the requested displacements. Admittedly, this is an approximation which suffers from several problems: the bumps do not self-occlude (either from the point of view camera or the light sources), object silhouettes are smooth, and other shading artifacts sometimes occur from backwards-facing normals. Even so, this approach is often quite adequate for relatively small displacements (see Figure 2). Our implementation retains the advantage of notational convenience; it is more intuitive to describe surface detailing in terms of surface displacement than in terms of perturbed normals. It is also important to retain compatibility with other implementations which do support true displacements. We do not see any inexpensive alternative strategy for using ray tracing and true surface displacements together.[2]

3.4. Using *trace*

The Shading Language defines a function *trace*(**P**,**D**) which returns the light color impinging on point **P** from direction **D**. Scanline renderers normally would not support the *trace* function (for example, *PRMan* always returns 0 from *trace*). Renderers which lack this facility must simulate reflective and refractive effects using environment maps. We continue to support environment maps, since this is still useful as a speedup method, and invaluable when combining CG foreground objects with live action background plates. However, we also support correct implementation of the *trace* function.

We implement the Shading Language *trace*() function in the obvious way—it directly invokes the ray tracing engine, which may recursively invoke other shaders. In addition, rays traced using this function may implicitly call the interior or exterior volume shaders associated with the object. A sample shader for a reflective metallic surface illustrates how the *trace*() function can be used (see also Figure 4):

```
surface
shiny ( float Ka=.25, Ks=1, Kr = 0.5, roughness = 0.005)
{
   point Nf = faceforward (normalize(N), I);
   point IN = normalize (I);
   color env = Kr * trace (P, reflect (IN, Nf));
   Ci = Cs * (Ka*ambient() + env +
              Ks*specular(Nf,-IN,roughness)));
}
```

Since *trace* may spawn rays which cause other objects' shaders to be evaluated, it is necessary for the ray tracing engine to support recursive rays, and for the SL interpreter to be fully reentrant. This would not need to be the case for scanline implementations. Indeed, the RI specification refers to the variables available to the shaders (such as P, N, s, etc.) as global.

3.5. Lights and Shadows

During interpretation of the surface shader, the *ambient*(), *diffuse*(), *specular*(), or *phong*() routines may be called to evaluate the light striking the surface (or an *illuminance* loop may appear in the shader). In any of these cases, the light source list is traversed, integrating the energy received from all of the nonphysical (**LightSource**) and physical (**AreaLightSource**) lights. When automatic shadows are requested (explained in the next section), ray casting is used to determine light visibility.

A small change is made to accommodate the radiosity algorithm: Ambient light is really just a trick for simulating interreflective effects which are explicitly calculated by the radiosity pass. When only the ray tracing is performed,

[2] Note: since acceptance of this paper, BMRT has been modified to support true displacements, albeit at large memory and time expense.

ambient() adds the contributions from all of the ambient light sources in the scene. However, in the cases where radiosity is performed, *ambient*() does not evaluate the ambient light sources at all, but rather returns the indirect illumination which was calculated during the radiosity pass. Remember that we separated direct and indirect lighting on the first pass. Our *ambient*() trick only adds the indirect component, since the direct component will be recalculated by the *diffuse*() function. Separating the two also allows for both higher fidelity in shadow calculation and for additional shading cues due to bump and displacement shading (since the light direction is not lost as it generally is for radiosity computations). This accounting of indirect illumination happens automatically and is completely transparent to the user and the shader author.

4. RenderMan binding for radiosity / ray tracing

Approaching the problem from the other side, we note that not all features of radiosity/ray tracing have obvious hooks to the RenderMan Interface. This section will outline our binding, i.e. how we used RenderMan to specify information needed by the radiosity and ray tracing engines. These are implemented using the **Attribute** and **Option** directives, the approved methods of specifying implementation-dependent data to the renderer.

4.1. Light Sources and Shadows

The standard RenderMan Interface allows specification of light sources through both the **LightSource** and **AreaLightSource** directives. Note that *PRMan* interprets an **AreaLightSource** call by instantiating a point light source, not an actual area light source. Both our ray tracing and radiosity support both of these types of light sources. Area light sources declared using the **AreaLightSource** directive are correctly handled in the obvious way. When they are evaluated, *Monte Carlo* integration over the surface geometry comprising the area light are used to estimate the integral that represents the true light arriving from the surface of the emitter.

We allow additional flexibility in declaring area light sources for radiosity by adding an implementation-dependent "emissioncolor" attribute which can be applied to any piece of geometry in the scene using the **Attribute** directive. For example,

```
AttributeBegin
  AreaLightSource "arealight" 1 "intensity" [0.75]
  Cylinder .5 1 2 360      # declare an area light cylinder
AttributeEnd
Illuminate 1 1
AttributeBegin
  Attribute "radiosity" "emissioncolor" [ .5 .5 .7 ]
  Sphere 1 -1 1 360        # this sphere is an bluish emitter
AttributeEnd
Polygon "P" [ ... ] # this polygon does not emit
```

This RIB example shows the declaration of three pieces of geometry: an area light source, an emitter, and a non-emitter, respectively. The semantic difference between an area light source and an emitter lies in how they are evaluated on the second pass. Both are treated similarly for the radiosity pass. However, as we mentioned earlier, area light sources are reevaluated on the second (ray tracing) pass. Emitters are not. This distinction allows us to specify that only certain area lights should be recalculated on the second pass, which may save time.

The RenderMan interface does not specify how (or whether) lights should cast shadows. With *PRMan*, if you want a light to be shadowed, you need to use a light source shader which explicitly references a shadow depth map to determine if the light is shadowed from a particular point [11]. The user needs to compute this shadow map ahead of time by rendering the scene as a depth image from the point of view of each light source which casts shadows.

Since a ray tracer can easily figure out if a light source is shadowed, we needed a way to specify this on a light-by-light basis. The Shading Language has no way to inquire about whether an unoccluded path exists between two points. We decided to solve this problem by adding an implementation-dependent attribute to light sources. The following RIB fragment illustrates our syntax:

```
Attribute "light" "shadows" ["on"]
LightSource "pointlight" 1       # this light casts shadows
Attribute "light" "shadows" ["off"]
LightSource "pointlight" 2       # this one does not
```

In addition, we also provided a mechanism for specifying which geometry can be considered as an occluder on an object-by-object basis. We did this using an implementation-dependent attribute:

```
Attribute "render" "casts_shadows" option
```

In order of increasing computational cost, *option* is one of: "none", indicating that the object does not occlude; "opaque" indicating that it blocks all light; "Os", indicating that the object has uniform opacity given by the **Opacity** directive (regardless of what the shader says); or "shader", which indicates that the surface shader should be evaluated for each shadow ray which intersects the object. This allows shaders which specify varying opacities for an object to determine how the object interacts with light as an occluder (Figure 3). This effect, particularly for partial transparency, would be difficult to achieve using shadow maps, which compare only depth.

4.2. Radiosity Information

Radiosity is notorious for not being very automatic; often a considerable amount of hand-tweaking is necessary to get an efficient radiosity solution. It is essential to be able to specify meshing rates, reflective and emissive characteristics, and other data on an object-by-object basis. Unfortunately, no standard mechanisms for these data exist in the RenderMan interface. Again, we make liberal use of the **Option** and **Attribute** directives. We have defined an attribute to specify the meshing rate for the radiosity solution:

```
Attribute “radiosity” “patchsize” [ps] “elementsize” [es]
    “minsize” [ms]
```

Where *ps*, *es*, and *ms* are floating point values representing the approximate sizes (in world space) of emitting patches and the maximum and minimum sizes of receiving elements, respectively, using a hierarchical radiosity scheme as in [4].

The radiosity engine needs to know the emissivity and reflectivity of each element in the scene. The element may be part of an object bound to a surface shader of arbitrary complexity, and there is no mechanism in the RenderMan Interface to query geometry for its average color, nor is there a predefined way to specify an object's reflectivity. We provide a mechanisms for this as follows. In the absence of any other information, the color given with the **Color** directive is assumed to be the average reflectivity of the object. This can be overridden by using the nonstandard "averagecolor" attribute:

```
Attribute "radiosity" "averagecolor" [ .4 .4 .7 ]
```

Generally, object reflectivity is implicitly defined as the surface color times the surface's Kd parameter, the renderer checks the shader bound to an object for a variable named "Kd", and uses the value to premultiply the average color. This allows more flexibility and makes changes to the shader parameters (but not the color) accurately change the reflective color that the radiosity calculations use. Emissivity of a patch (different from an area light source, as explained in section 3.5) is conveyed similarly:

```
Attribute "radiosity" "emissivity" [ .4 .4 .7 ]
```

In order to accommodate our renderer's ability to handle specular-to-diffuse reflections, similar attributes exist for specifying "specularcolor", "transmitcolor", and "refractionindex" on an object-by-object basis. Note that these refer to indirect specular reflective/refractive properties, *not* to the way the surface appears to the viewer (this is handled in the surface shader).

Other miscellaneous parameters, such as the number of progressive refinement steps to perform or the error tolerances, can be specified either as nonstandard options using the **Option** directive, or may be specified as command-line options when invoking the renderer.

5. Discussion

Though not often admitted by researchers, it is true that for most rendering tasks, global illumination is unnecessary. Particularly for large production tasks, the speed of scanline methods is often more desirable than the accuracy of global illumination methods. Ray tracing is much slower than scanline methods and cannot cheaply utilize true displacements; its nature as a point sampling method makes it more expensive to compute area operations. However, many applications or effects do need these capabilities (Figures 4-6), which include radiosity and diffuse interreflection, volumetric and participating media effects, area light sources, and accurate automatic shadows using ray casting.

Compliance with the RI standard has been established by extensive comparisons to *PRMan* (see figure 7). When rendering scenes which use RI features common to both renderers, the resulting output is nearly indistinguishable (in fact, the authors of both BMRT and *PRMan* are sometimes unable to correctly guess which is which)[3]. *PRMan* is typically around 5 times faster than our software, largely due to the greater efficiency of REYES compared to ray tracing (as well as the skillful optimizations by *PRMan*'s authors). However, it is not hard to create pathological examples that either render slightly faster with BMRT or render 10 or more times faster with *PRMan*.

In the process of creating and distributing this software, we have learned a number of important lessons about rendering systems and users' expectations of them. We feel many of these are of general use to creators of rendering tools. First, some observations about global illumination in general:

- The popular view of radiosity being prohibitively expensive does not ring true to us. A fairly straightforward radiosity calculation can greatly increase the visual richness of the illumination of a scene. One must watch out for the law of diminishing returns -- typically 90% of the quality is generated by the first 10% of radiosity computation. Careful pruning of the calculations can yield good results for the radiosity pass in a fraction of the time required for the actual scan conversion and shading of the image.
- Nevertheless, our experience with users shows that choosing good meshing rates, convergence criteria, and other radiosity parameters is not intuitive to users not previously familiar with radiosity. We have found no straightforward heuristics, nor any foolproof way to automate choice of the parameters.

And regarding RenderMan and other high-end standards:

- Practically no noncommercial rendering packages (and very few commercial packages) support the variety of geometric primitives, spatial and temporal antialiasing, programmable shading, and other advanced features that are required by the RenderMan Interface. This explains why so few public renderers see production-style use, and implies that standards such as RenderMan "raise the bar" by requiring advanced features that would be of use to professionals.
- We cannot overemphasize the importance of programmable shading, particularly with support of antialiasing and area operators. We are aware of no other public domain or shareware rendering system that has as flexible a shading system as is required by RI. This makes the task of writing renderers more difficult by requiring understanding of languages and compilers, but it is well worth the price.
- Our experience has been that many of our users are the same studios who use *PRMan*. This implies that (1) investment in software and tools that support RIB and SL influence subsequent tool usage (i.e. they're more likely to use BMRT because it is compatible with their primary renderer); and (2) even studios successfully using *PRMan* for feature film work are interested in at least experimenting with, and possibly actually using, global illumination.

[3] Of course, it is trivial to tell the difference when using features not common to both packages; for example, large displacements only work with *PRMan* and radiosity only works with BMRT.

- It is rarely possible to directly compare different rendering algorithms on identical input. Too frequently, we see papers comparing hand-crafted algorithm A to hacked-together algorithm B. Results of such tests are rarely meaningful. It would serve the entire graphics community if algorithms could be compared to optimized, proven packages on the same input. High-end standards such as RI help to facilitate this.
- Comparisons between BMRT and *PRMan*, simple to perform because they take the same input, have revealed bugs in both packages. This further argues for common formats and standards in rendering. If it were easier to make such direct comparisons of packages, we expect that all of the packages would see great improvements.

We would certainly like to see continued popularity and development of the RenderMan Interface. Future revisions of the standard could do much to accommodate non-REYES implementations and the global illumination community. It would be very helpful to have standard ways to communicate meshing rates, emissivity, and reflectivity of geometry.

A facility should be added to the Shading Language to ask for light visibility information in the absence of shadow maps, as well as to inquire at run time which of the optional capabilities of the standard are available. Surface and light shaders could be modified to allow for specification of physical units, and to give general BRDF's. It may also be desirable to have a means of enforcing physically realistic behaviors of light and surface shaders (for example, ensuring energy conservation).

Both access to global illumination renderers and common formats for rendering systems are sorely lacking. The global illumination community would do well to establish such standards. This would serve both to widen use of such renderers and to provide a basis to compare renderers to each other for research purposes. We have shown that the RenderMan Interface Standard is quite capable for global illumination applications, and conversely that ray tracing and radiosity are viable implementation methods for the RenderMan Interface.

Software Availability

A home page for the software described in this paper is available from http://www.seas.gwu.edu/student/gritz/bmrt.html. Binaries are available for a variety of Unix platforms and may be found at ftp://ftp.seas.gwu.edu/pub/graphics/BMRT.

Acknowledgments

The authors wish to thank the many users of BMRT, especially the students at GWU who were among the first guinea pigs. The first author would like to single out Michael B. Johnson and Tony Apodaca for being particularly helpful in our effort to make BMRT available to the world. Rudy Poat and Steve May graciously allowed us to use their images. *RenderMan* and *PhotoRealistic RenderMan* are registered trademarks of Pixar.

References

[1] Apodaca, Anthony A., ed. SIGGRAPH '90 course notes #18 (The RenderMan Interface and Shading Language), 1990.

[2] Apodaca, Anthony A., ed. SIGGRAPH '92 course notes #21 (Writing RenderMan Shaders), 1992.

[3] Blinn, Jim. Simulation of Wrinkled Surfaces. *Computer Graphics* **12**(3):286-292, 1978.

[4] Cohen, Michael F., Donald P. Greenberg, Dave S. Immel, and Philip J. Brock. An efficient radiosity approach for realistic image synthesis. *IEEE Computer Graphics and Applications*, **6**(3):75-84, March 1986.

[5] Cohen, Michael F., Shenchang Eric Chen, John R. Wallace, and Donald P. Greenberg. A progressive refinement approach to fast radiosity image generation. *Computer Graphics*, **22**(3):75-84, 1988.

[6] Cook, Robert L., Loren Carpenter, and Edwin Catmull. The Reyes Image Rendering Architecture. *Computer Graphics*, **21**(4):95-102, 1987.

[7] Cook, Robert L., Thomas Porter, and Loren Carpenter. Distributed ray tracing. *Computer Graphics*, **18**(3):137-145, 1984.

[8] Gritz, Larry. *Computing Specular-to-Diffuse Illumination for Two-Pass Rendering*. Master's Thesis, Dept. of EE&CS, The George Washington University, Washington DC 20052, May 1993.

[9] Hanrahan, Pat and Jim Lawson. A language for shading and lighting calculations. *Computer Graphics*, **24**(4):289-298, August 1990.

[10] Pixar. *The RenderMan Interface*, version 3.1 specification, September 1989.

[11] Reeves, William T., David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. *Computer Graphics*, **21**(4):283-291, July 1987.

[12] Shirley, Peter S. *Physically Based Lighting Calculations for Computer Graphics*. Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1991.

[13] Slusallek, Philipp, Thomas Pflaum, and Hans-Peter Seidel. Implementing RenderMan—Practice, Problems and Enhancements. *Computer Graphics Forum* **13**(3):443-454 (Proceedings of Eurographics '94, Oslo, Norway, Sep. 12-16, 1994), Blackwell Publishers, Oxford (1994).

[14] Upstill, Steve. *The RenderMan Companion*, Addison-Wesley, 1989.

[15] Wallace, John R. and Michael F. Cohen. A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods. *Computer Graphics*, **21**(4):311-320, 1987.

[16] Wallace, John R., K. A. Elmquist, and Eric A. Haines. A ray tracing algorithm for progressive radiosity. *Computer Graphics*, **23**(3):315-324, July 1989.

Captions for the color plates:

Figure 2: "Fake" displacement shading, use of the *trace* function, and automatic shadows.

Figure 3: Ray tracing allows for partially transparent shadowing that would be difficult to achieve with shadow depth maps. Image courtesy of Steve May, Ohio State University ACCAD.

Figure 4: "Kitchen Scene" combines radiosity and ray tracing.

Figure 5: "Dresser" shows radiosity, procedural textures, specular-to-diffuse illumination.

Figure 6: Volumetric and participating media effects using volume shaders and global illumination.

Figure 7: Comparison of images from PRMan (left) and BMRT (right) using identical input. The PRMan image used shadow depth and reflection maps, while the BMRT image used automatic shadows and reflections.