Global Illumination: Efficient Renderer Design and Architecture

Michael Farnsworth and Robert F. Erbacher

Computer Science Department, Utah State University, Logan, UT
mafarnsworth@cc.usu.edu and Robert.Erbacher@usu.edu

1. Abstract

In this paper we show that developing a global illumination (GI) renderer can be made efficient through the creation of a multi-mode rendering system. Such a system should include a real-time target coupled with scene modification tools and using hierarchical scenes and bounding volumes to optimize the development cycle and run-time performance. Additional design considerations include using the ray-processor paradigm and efficient testing using reference images. We also provide examples of optimizations taking advantage of these techniques.

1. Introduction

The global illumination class of renderers differ from other renderers in that they account for all light paths, be they specular, diffuse, or any combination thereof, as they travel from a source to the eye or camera. Traditional raytracing, for example, will usually miss diffuse to diffuse light transport and miss out on accurate color bleeding effects.

GI renderer development brings with it a number of difficulties. The typical code-compile-test cycle can be the most time-consuming task for a developer, and the highly computational nature of GI renderers creates a very drawn-out testing phase. There can be long waits to determine if a scene is set up correctly, a particular issue was eliminated, or a feature implemented correctly. The classic problem of a developer losing time tweaking source code is a significant problem when the process of verification is much longer than the implementation itself, as it is in GI renderer development. A related issue is that testing itself presents a problem in that verification is somewhat subjective. After many cycles of testing, does the developer really know if the results are closer to the desired accuracy and correctness?

As the testing phase of the development cycle is much longer for GI renderers, care must be taken to accommodate optimizations affecting all aspects of the runtime efficiency of the renderer. A good first step is to create a number of test scenes that fully exercise the functionality of a robust implementation. A treatment of this subject containing a number of example test scenes was provided by Smits and Jensen [1] and the well-known variations on the Cornell Box scene might be employed as well. After some scenes are selected or created, a known-good renderer should be used to create reference images. For example, the Radiance package [2] might be used in a path-tracing mode to create accurate references for comparison during development. This was done by Jensen to show the robustness of his two-pass photon map algorithm [3], comparing final output to path-traced images.

In order to explicitly verify the accuracy of the results, a *difference image* should be computed by taking the absolute value of the difference between pixel components, pixel by pixel [6]. The less black the resulting pixels are, the more divergent the test image is from the reference image. Identified problem

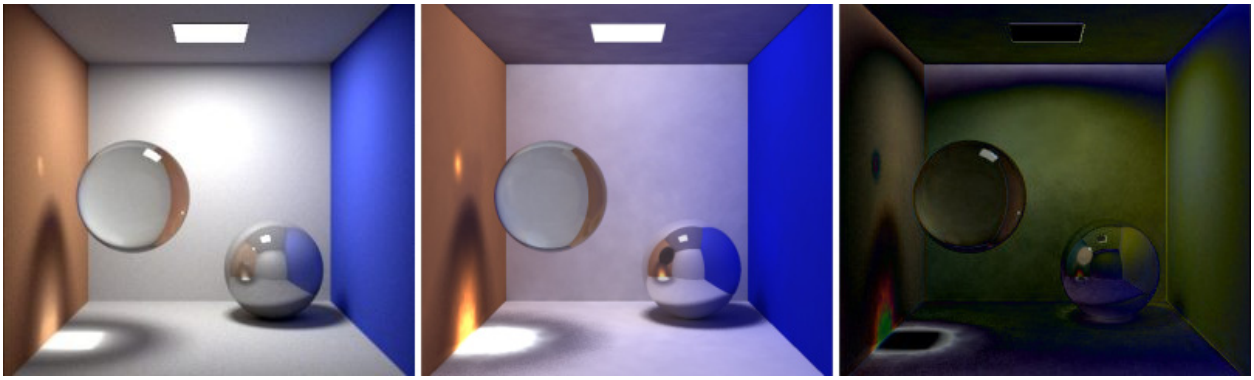

Figure 1: WinOSI-rendered reference image (left), rendered test image (middle), and difference image (right).

areas should be examined for faults in the rendering system. In figure 1 we show a comparison between a reference image from WinOSI, an equivalent image from our implementation renderer, and finally the difference image. Many of the differences are due to the fact that WinOSI is wavelength based [7] while our implementation is strictly RGB based. Note that the color bleeding is stronger in our implementation. These types of discrepancies are more easily detected in difference images.

An appropriate scene organization and management scheme is also indispensable. A simple starting platform should include a hierarchical scene organization built around recursive method calls. This will allow for simple, yet effective, scene elements to be implemented more easily. This lends itself well to the usual raytracing based GI renderer functionality. Creating a multi-mode rendering system capable of rendering global illumination, simple raytracing, and real-time previews greatly aids in the development process.

2. Multi-Mode Renderers

While seemingly counterintuitive, integrating a real-time preview into a GI renderer can provide tangible benefits in turnaround time during the development cycle. The principle benefit is that it allows real-time scene manipulation such as moving objects, changing material properties, etc. The camera can navigate at will and some settings for the GI render mode can also be previewed and applied easily in a responsive environment.

In our test implementation, while adding irradiance caching [5] to Jensen's two-pass photon mapping algorithm [3], the typical turnaround time per change/test cycle for a 400x400 pixel image of a cornell box scene with four rays per pixel was approximately 20 minutes. The extensive time requirement was largely due to difficulty in arranging a scene where deficiencies could be demonstrated easily and then waiting for the algorithm to scan to the critical image region for verification. We implemented a preview renderer for managing instance transforms and material properties in OpenGL with a GUI interface using Gtk+ 2.0/Gtkmm 2.0. Once the preview renderer and GUI were integrated the turnaround time was reduced to approximately 5 minutes for the code-compile-test cycle. While anecdotal and highly dependent on the developer, the difference in time still demonstrates the value of the investment. Scene manipulation tools with a real-time renderer can and will reduce turnaround time. It also has the benefit of being highly reusable across renderers.

Preview renderers as part of a multi-mode rendering system are only useful, however, if their output properly corresponds to what will be produced during the full GI render. Environment mapping might be used to quickly simulate reflection and a combination of transparency and reflection used to simulate refractive surfaces and Fresnel reflection. Shadows can be simulated using a volumetric or shadow-map algorithm, depending on how easily it can be integrated from the immediate-mode toolkit used. Lighting must also be relatively accurate but need not be exact. For example, we used one point light per corner of an area light source and a single point light for a spherical source as approximations. Our implementation also included a lower limit on translucency so objects that normally were totally transparent but still refractive would not disappear during the preview. As the preview renderer is meant to be an aid to the developer, these types of approximations help developers quickly see the material properties of scene objects and understand what light interactions will result when the full GI renderer is applied.

2.1. Rendering System

Each renderer in the multi-mode set should provide the same methods to objects to be rendered. Our implementation includes a module for each renderer and in particular an interface to them consisting of 'renderables', i.e., simple items such as lines, points, polygons, and so on which contain only the necessary information for immediate-mode rendering. The renderer is asked to pre-render, render each renderable, and finally post-render. Pre-rendering includes clearing buffers, setting up necessary rendering states, etc. Post rendering includes the final processing and flip-to-screen. In the case of a real-time preview, rendering is performed as renderables are submitted or are deferred onto a sorted list for optimal rendering speed and rendered in one final step during post-render. In contrast, a GI render target ignores submitted renderables but in post-render emits photons from lights, scans out the scene, or otherwise does as dictated by each particular algorithm.

To support planned optimizations, new basic renderables should be easy to create and new scene objects should be simple to implement to aid in creation of interesting test scenes. The last part of the rendering system, in line with this principle of extensibility, is that each scene object must implement a simple interface to interact with the rendering system. Scene objects must be capable of submitting the renderables they are composed of (e.g., for the preview renderer), computing self intersecting rays, and attaching typical object property values. Computation of self-intersecting rays is critical for determining if they themselves are the nearest

intersection to said ray. Typical object property values include: material properties, positions, ray distances, and surface normals.

Non-real-time renderers must include the ability to provide continuous feedback on the rendering process. An efficient implementation is to invoke a progress update method for every row of pixels rendered. This is an ideal place to draw the newest pixels to the screen, allow the GUI to process events, and perform any other necessary activity required to show meaningful feedback. Our implementation includes an option to cancel the rendering, in which case it will switch back to the real-time renderer and allow for further scene modification.

3. Enabling Optimization

Facilitating optimization largely involves ensuring that the renderer's architecture is designed to easily add improvements such as ray-intersection speedups, irradiance caching, etc. We describe an architecture that meets these goals, give examples of optimizations, and describe associated benefits.

3.1. Centralized Ray Processor

GI renderers are typically founded on one or two pass implementations in which the first pass handles light source emissions and the second pass gathers results via a form of raytracing. There are three main components to a raytracing engine: the emitter, the retrace processor, and the ray-intersection handler.

In global illumination there are generally two separate ray processors that share the same ray-intersection handler. For example, in two-pass photon mapping there is a photon emitter and a photon retrace processor for light sources. There is also a regular raytrace emitter and a ray processor for the camera. Each emitter sends rays into the scene to determine nearest intersections and recursively process them using the corresponding ray processor. However, in each case the ray-intersection calculations are performed using the same handler. Figure 2 provides an overview of a second-pass ray processor. We note that the second pass, where rays are 'emitted' from the camera, is a potentially inaccurate way to describe the process. More formally, the rays emitted in this case are meant to gather radiance. So this second pass in a two-pass system is more appropriately termed a "scatter-gather" process with the first pass scattering light and the second pass gathering it.

Certain algorithms, such as radiosity/raytracing hybrids, may not have two emitter/processor pairs but rather just use the gather stage after distributing

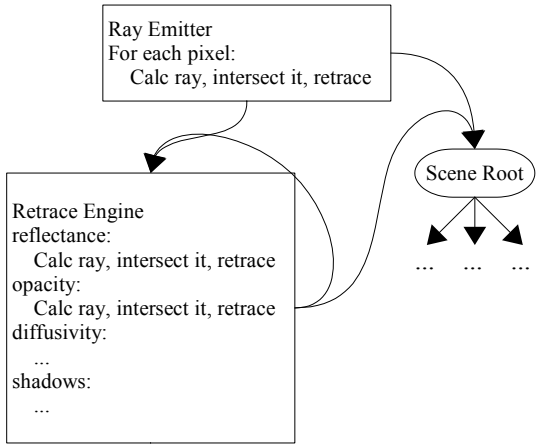

Figure 2: Example ray processor for a gather pass.

radiance via some other method. In any case, every GI renderer should have at least the second-pass raytracing processor at its disposal. We found that this module must have the ability to run on its own, without the GI first pass, to verify that the raytracing engine itself works properly. When in a gather-only mode it will lack some features, such as caustics and color bleeding, which is what the full GI renderer is meant to incorporate. A raytrace only capability can be useful in testing the basic feature set of the renderer and to ensure that any visual artifacts resulting later are likely localized to GI-only code. Another benefit is that a raytracing render will be much faster than a full GI render and will still incorporate accurate basic effects such as refraction, reflection, and (soft) shadows. This makes for a quick (but not real-time) preview for a GI render and is easy to incorporate into a GI renderer that already contains a raytracing pass. A diagram of the code reuse possible among renderer modules is exemplified in figure 3.

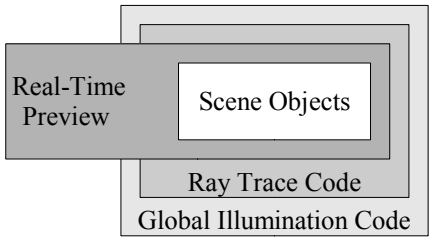

Figure 3: Code reuse among renderer and scene modules.

3.2. Hierarchical Scenes

As discussed earlier, GI renderers, as well as raytracers and real-time renderers, benefit from a hierarchical scene organization due to the nature of

the algorithms involved. It is imperative that intersections with the nearest scene object for a ray be found as quickly as possible. There exist many methods for doing this: hierarchical bounding volumes, octrees, etc. We found that employing hierarchical bounding volumes to manage scenes and speed up ray-intersection testing to be the most effective solution.

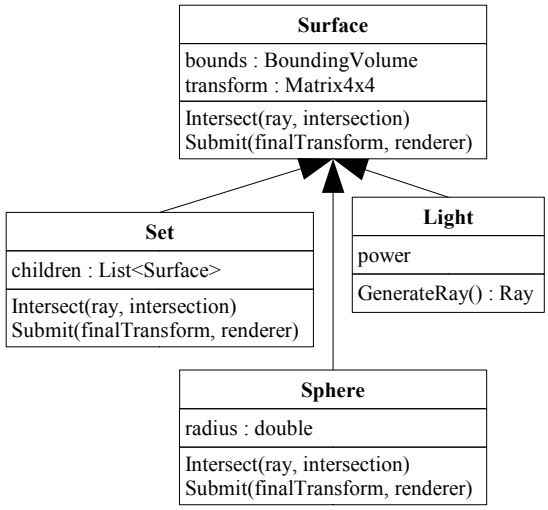

Figure 4: Example classes enabling simple hierarchical- and bounding-volume-based scenes.

Shirley and Morley use this method in their *Galileo* renderer [4]. The concept is simple. Every scene object is descended from the 'surface' class and implements a ray-intersection method; more than one if each method is specialized for a given ray type. A set or group subclass is included that simply serves as a container of other scene objects. Each object has a bounding volume. We employed bounding spheres. However, Shirley and Morley described how to effectively use axis-aligned bounding boxes. This allows the ray to be tested against a bounding volume and if it fails the detailed tests are not performed but rather the test moves on to the remaining objects within the scene. It is important to maintain bounding volumes for sets that contain other scene objects and the bounding volume for the scene must contain the bounding volumes of all of its children as tightly constrained as possible. This allows the intersection test to skip whole groups of objects quickly and minimize costly calculations. Our implementation used spheres to keep calculations as fast as possible, with a flag to denote those volumes enclosing infinite sized objects. It is important to note that the parent-child relationships need not be explicitly specified by the developer or scene designer. Instead, all of the leaf (non-group) objects can be divided up evenly to reduce the amount of open space within the bounding regions and maintain more consistent trace speeds. Shirley and Morley provide a treatment on axis-aligned bounding boxes in *Realistic RayTracing* and for balanced hierarchies we recommend their implementation [4]. Our own implementation had sets and subsets specified explicitly by the developer as the scene was constructed, as most of our test scenes were simple. We provide an overview of some classes employing bounding volumes and hierarchical organization in figure 4.

Rays that are sent into the scene only need tell the root of the scene hierarchy to test the ray against itself and it will in turn test the ray against its children within the hierarchy until the closest intersection is found. It now becomes simplistic to employ other ray-intersection speedups such as octrees to optimize rendering as they are easily integrated with a straightforward hierarchical scene system. Before rendering the scene hierarchy can be traversed, adding leaf scene objects to an octree. Then the root scene object, when asked to test ray intersections, can send the ray through the octree which will do its own recursive testing of octnodes to find the nearest intersection. Again, once an intersection is found and no closer instances arise, the latest intersection information (e.g., location, surface normal, material properties, etc.) is returned to the ray processor. The ray processor then employs the retrace engine to trace additional child rays for reflection, refraction, shadow tests, and any other necessary sampling. This retrace then triggers additional retraces and so on until the maximum ray depth is achieved or the contributed radiance falls below a specified threshold and becomes insignificant.

3.3. Optimization Examples

This section presents specific examples of the application of optimization techniques, providing insight into the achieved benefits in the development process from the previously described enhancements. These examples are given in the context of a photon map implementation, using Jensen's two-pass method. The first optimization is a necessary one for all raytracing-based renderers. This is the incorporation of stratified random sampling. When a ray from the camera intersects a diffuse surface, the diffuse transport is gathered by tracing additional rays from the intersection location into the scene and adding in irradiance from the other surfaces. Just randomly sampling rays in the hemisphere above the surface will yield highly varying irradiance and the resulting image will therefore have a fair amount of noise. To reduce this noise and be able to use fewer samples per irradiance calculation improved sampling must be used. Stratification is one such sampling technique and means that each random sample is taken from

within a uniformly subdivided hemisphere above the intersection point so that no two random samples fall within the same subdivision. The ray processor will retrace those sample rays as needed and the logic to initiate the samples for the irradiance calculation should occur in the retrace code as well. In this manner, every ray that is traced runs through the ray processor for retracing when necessary.

An additional optimization for calculating irradiance is *irradiance caching*. Based on the property that diffuse irradiance nearly always changes slowly across surfaces, it is possible to limit the need to compute irradiance by interpolating across the surface from a select subset of points. Ward, Rubinstein, and Clear pioneered the technique [5] and Jensen recommends it for photon mapping [3] as a needed optimization. Indeed, it can be used in nearly any raytracer to improve speed when incorporating irradiance. The relevant portion of the technique requires that each time a detailed irradiance computation is performed, a maximum radius is employed to be used in the interpolation. A simple octree is built that stores spherical objects and each sphere has as an attribute the irradiance computed at its center. When deciding if there are sufficient nearby cached irradiance points, a ray intersection point is tested against the octree contents to find the nearest irradiance cache spheres intersecting with the test point. In the retrace code the ray processor evaluates whether it needs to do a detailed irradiance calculation or merely interpolate the cached calculations.

The architecture outlined previously aids immensely in implementing these optimizations. It must be noted, however, that during implementation many development cycles were needed to tune each feature. For example, the stratification of rays needed work to ensure that we were properly using a cosine-distributed sample of rays in the hemisphere above the intersection point. Irradiance caching has several parameters used for fine tuning, testing and calibration of which is time consuming. Additionally, the output can often be misleading. To remedy this, efficient modification of scenes and properties using a real-time renderer and using a raytrace only mode helped to locate many of the issues during implementation more quickly. To efficiently verify the correctness of the output, reference images were used to convincingly show that the output was indeed correct.

4. Conclusion

Developing a global illumination renderer can be completed successfully and efficiently using multi-target rendering by integrating a real-time renderer along with scene manipulation controls and retaining one-pass raytracing. Additionally, scene organization is pivotal to the renderer and a hierarchical system will yield simple and scalable interaction between scene and rendering engines. The ray processor paradigm for individual passes applies well to any stage that utilizes ray-object intersections.

Finally, efficient testing requires that image quality be quantitatively measured. The use of difference images has been found to be effective and efficient at providing this testing need. These techniques combine to provide a clear picture for how to proceed with GI renderer development.

5. Acknowledgements

We would like to thank Dr. Joel Duffin of Utah State University for his input and perspective on both the implementation and the resulting research. We also thank Michael Granz, author of WinOSI, for the reference image and renderer for comparison.

6. References

[1] B. Smits, and H. W. Jensen, "Global Illumination Test Scenes," Tech Rep. UUCS-00-013, Computer Science Department, University of Utah, June 2000.

[2] G. W. Larson, and R. Shakespeare, "Rendering with Radiance," Morgan Kaufmann, Los Altos, California, 1998.

[3] H. W. Jensen, "Realistic Image Synthesis using Photon Mapping," AK Peters, Natick, Massachusetts, 2001.

[4] P. Shirley, and R. K. Morley, "Realistic Ray Tracing, Second Edition," AK Peters, Natick, Massachusetts, 2003.

[5] G. J. Ward, F. M. Rubinstein, and R. D. Clear, "A Ray Tracing Solution for Diffuse Interreflection," *Computer Graphics*, SIGGRAPH 1988 Proceedings, 22 (4), August 1988, pp. 85-92.

[6] Cornell University Program of Computer Graphics, "Cornell Box Comparison," http://www.graphics.cornell.edu/online/box/compare.html

[7] Michael Granz, WinOSI Optical Simulation Renderer, http://www.winosi.onlinehome.de/