Benjamin R. Landon

Using Graphics Hardware to Accelerate Progressive Refinement Radiosity

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-04-02
February 2004

Using Graphics Hardware to Accelerate Progressive Refinement Radiosity

Benjamin R. Landon
Department of Computer Science
Brown University
Providence, Rhode Island 02912
brl@cs.brown.edu
brl@acm.org

1 Introduction

Progressive refinement radiosity is an iterative approach to generating a global illumination solution for a purely diffuse scene. Progressive refinement radiosity was introduced by Cohen et al. [Cohen et al. 1988] and it is very similar to Southwell relaxation, which is described in detail in Sillion and Puech [Sillion and Puech 1994].

In progressive refinement, a scene is divided into patches, each of which has radiosity and delta radiosity attributes. Additionally, each patch has a diffuse reflectance ρ and area A. One iteration of progressive refinement radiosity has two steps: shooting and reconstruction. Iteration is done either a fixed number of times or until the convergence criteria are met.

In the shooting step, the scene patch with the greatest delta radiosity is treated as an emitter. Finding the patch with the greatest delta radiosity is not treated as a separate step since this information can be cached during scene initialization and reconstruction. The shooting step consists of looking up the form factors F_{ij} for the emitting patch i and for each receiving patch j and computing the amount of radiosity shot to each receiver. This is given by the expression from Cohen et al.:

$$\Delta Rad_{ij} = F_{ij}\rho_j\Delta B_i A_i/A_j$$

where ΔB_i is the delta radiosity of the emitting patch i and ρ_j is the diffuse reflectance of the receiving patch j [Cohen et al. 1988].

The reconstruction step in progressive refinement radiosity consists of iterating over the scene patches j and incrementing the patch's radiosity and delta radiosity attributes by the amount ΔRad_{ij} computed in the shooting step. Finally, the delta radiosity of the emitting patch i, ΔB_i, is set to zero.

I have implemented a system that computes the contribution to ΔRad_{ij} per hemicube cell. Using features available on modern graphics processors (GPUs), this contribution can be computed with one rendering pass per hemicube face. The reconstruction then consists of iterating over the hemicube and adding the ΔRad_{ij} contribution for each hemicube cell to the appropriate target patch j.

The reconstruction step is implemented in C++ code and executes on the CPU. Limitations of the floating point render target formats on modern graphics hardware make the reconstruction phase difficult to implement on the GPU.

For the purposes of this report I will refer to this system as *PRRoG*, which is a mnemonic for Progressive Refinement Radiosity on Graphics hardware. PRRoG is approximately five thousand lines of C++ code written for Microsoft Windows. It uses the GPU features available on the ATI RADEON 9700 Pro, particularly the DirectX 9 pixel shader version 2.0 instruction set and the floating point render target and texture formats. A sample rendering of the Cornell Box is given in the results section. The primary contribution of this report is that the shooting step in progresive refinement can compute each hemicube cell's contribution to ΔRad_{ij}. Furtherore, this per-hemicube-cell contribution to ΔRad_{ij} can be computed using the features on modern commodity graphics hardware.

The progressive refinement technique presented by Cohen et al. [Cohen et al. 1988] uses *substructuring*, where patches are further subdivided into elements. The patches are treated as the emitters of radiosity and the elements are the receivers of radiosity. PRRoG does not use substructuring, and the emitters and receivers are treated symmetrically. In the implementation presented in this report there are no elements, only patches.

2 Previous Work

There is a very large body of literature on radiosity techniques in computer graphics. Summarizing the body of radiosity literature is far beyond the scope of this report. For a good introduction to radiosity, see the book by Glassner [Glassner 1995]. More comprehensive surveys are available in the books by Sillion and Puech [Sillion and Puech 1994] and by Cohen and Wallace [Cohen and Wallace 1993].

Progressive refinement radiosity was introduced by Michael F. Cohen, Shenchang Eric Chen, John R. Wallace and Donald P. Greenberg [Cohen et al. 1988]. The hemicube technique for computing form factors between scene patches was introduced by Cohen and Greenberg [Cohen and Greenberg 1985].

Mark Harris proposed using the lighting features in fixed-function graphics hardware to perform progressive refinement radiosity calculations [Harris 2001]. Harris's ideas were extended and implemented for programmable graphics hardware by Greg Coombe, Mark Harris, and Anselmo Lastra [Coombe et al. 2003]. As far as I know , the implementation of Coombe et al. is currently the most

successful attempt at implementing a shooting radiosity solution on a modern GPU.

3 Computing Form Factors in Traditional Radiosity Implementations

The computation of the form factors deserves further comment since this is important to the contribution of PRRoG. The form factor F_{ij} is often computed with the hemicube technique [Cohen and Greenberg 1985]. A hemicube is placed over the patch i, and scene patch identifiers instead of colors are rendered to a frame buffer corresponding to each hemicube face. This image of patch ids is called the *item buffer*.

Each pixel of the item buffer has a corresponding hemicube cell, and therefore a corresponding delta form factor. The phrases "item buffer pixel" and "hemicube cell" are often used synonymously.

Each item buffer pixel is then examined and for each patch j found in the item buffer, the form factor F_{ij} is incremented by a small amount, called the *delta form factor*. The delta form factor that corresponds to a particular pixel in the item buffer depends only on the hemicube cell coordinates (x, y, z). These hemicube cell coordinates are in hemicube units. z is usually in the range [0,1] and x and y are in the range [-1, 1].

The form factors are a function of scene geometry and they do not depend on lighting or material parameters. The delta form factors are not a function of the scene, so they can be encoded into a texture or table.

The delta form factors for the top of the hemicube are given by:

$$\Delta F_{ij}(x, y, z) = \Delta A/(\pi(1 + x^2 + y^2)^2)$$

where ΔA is the area of the hemicube cell at (x, y, z). The delta form factors for the sides of the hemicube can be computed with a similar expression, which is discussed in the article by Cohen and Greenberg [Cohen and Greenberg 1985], and in the books already mentioned.

Computation of the form factor F_{ij} can be done eagerly or lazily. The approach described by Cohen et al. [Cohen et al. 1988] uses a lazy approach, computing F_{ij} only when needed.

4 Shooting Radiosity Per Hemicube Cell

The PRRoG system does not explicitly compute form factors. Instead, PRRoG takes advantage of the fact that ΔRad_{ij} can be computed with the delta form factors, and that the delta form factors are defined per hemicube cell. This

means that the contribution of each hemicube cell to ΔRad_{ij} can be easily computed. Avoiding the explicit storage of the form factors results in a large memory savings. This is the central point of this report, and it is the main contribution of PRRoG.

Consider the equation for the amount of delta radiosity shot during one iteration of progressive refinement.

$$\Delta Rad_{ij} = F_{ij}\rho_j \Delta B_i A_i / A_j$$

Suppose that form factors are not pre-computed, and instead this computation is done during the shooting phase. If the form factor is given by the sum over an item buffer of the delta form factors:

$$F_{ij} = \sum \Delta F_{ij}(x, y, z)$$

then the amount of radiosity shot from the emitting patch i is:

$$\Delta Rad_{ij} = \left(\sum \Delta F_{ij}(x, y, z) \right) \rho_j \Delta B_i A_i / A_j$$

Let δRad_{ij} be the amount ΔRad_{ij} accumulated per hemicube cell. δRad_{ij} is given by:

$$\delta Rad_{ij} = \Delta Rad_{ij}(x, y, z) = \Delta F_{ij}(x, y, z)\rho_j \Delta B_i A_i / A_j \quad (1)$$

This expression can be computed in a single rendering pass per hemicube face, which is the heart of what is interesting about PRRoG. The reflectance and receiving patch area can be set as patch properties. The delta radiosity of the shooting patch times its area, $\Delta B_i A_i$, can be set as a light parameter. The multiplication by the delta form functions $\Delta F_{ij}(x, y, z)$ can be done by modulating with a projective texture. The evaluation of the Equation 1 is done in a DirectX pixel shader.

I propose that Equation 1 looks more like an ambient lighting calculation than a diffuse one. If we think of ρ_j / A_j as an ambient material for the receiving patch, and $\Delta B_i A_i$ as the ambient light term for the emitting patch, then we have some amount of radiosity which can be accumulated per hemicube cell. This amount of radiosity shot per hemicube cell is denoted δRad_{ij}.

One of the major obstacles is one of precision. Often the framebuffer is limited to 8 bits per color channel, so 1/255 is the smallest radiosity value that can be shot. Values for δRad_{ij} are typically smaller than 1/255, so floating point render targets must be used in this application.

Like the values for δRad_{ij}, the delta form factors are also less than 1/255. Therefore, floating point texture format are used to encode the delta form factors as well.

5 Initialization in PRRoG

The scene is loaded and the patches are subdivided to the desired level. The patch radiosities are not stored as textures in PRRoG due to the difficulty in performing the reconstruction step on current GPUs. The scene patches are then copied to DirectX vertex buffers for fast rendering. The scene geometry is copied to a vertex buffer, and quotient ρ_j/A_j is stored per vertex in the vertex buffer. Five 128-bit floating point RGBA render targets are created, one for each face of the hemicube. I also create two floating point textures corresponding to the delta form factors for the top and the side of the hemicube. Note that the delta form factors are the same for each of the four side faces of the hemicube.

The pixel shader is loaded and compiled. The pixel shader is what performs the evaluation of Equation 1 on the GPU.

The shooting phase and reconstruction phase are performed for a fixed number of iterations. This will eventually be changed to have a convergence metric. The shooting phase and the reconstruction phase are described in the following sections.

6 The Shooting Phase In PRRoG

The brightest emitting patch is found, and then the hemicube is positioned over it, and the rendering target buffers are cleared. The product $\Delta B_i A_i$ is set as a constant to be used by the pixel shader. The scene is then rendered onto each face of the hemicube, with the RGB components of the render targets being used to hold the RGB values of the per-hemicube-cell delta radiosity. The alpha channel is used for the receiving patch id. Projective texture mapping is used to multiply by the delta form factors. The delta form factors are encoded into a floating point texture, and these textures are situated on the appropriate face of the hemicube for rendering.

7 The Reconstruction Phase In PRRoG

I have not yet found a good way of adding the amount of radiosity shot back to the scene using the GPU. By iterating over the hemicube I can extract the receiving patch id from the alpha channel, but this is currently done on the CPU. Pseudocode for the reconstruction step is given in Figure 1.

An obstacle to performing the reconstruction on the GPU is that the *add* blending function is not available for floating point render targets on the current generation of commodity graphics cards. In fact, no blending operation other than *source copy* is available for these formats. It is possible that support for blending operations on floating point render targets will be available on the GPU sometime in 2004.

```
Lock the hemicube render target surfaces
For each pixel in the render target
    id for patch j = alpha channel of pixel
    received delta radiosity is rgb channels of pixel
    increment delta radiosity of patch j by received delta radiosity
    increment patch j radiosity by received delta radiosity
```

Figure 1: Psuedocode for reconstruction phase

8 Results

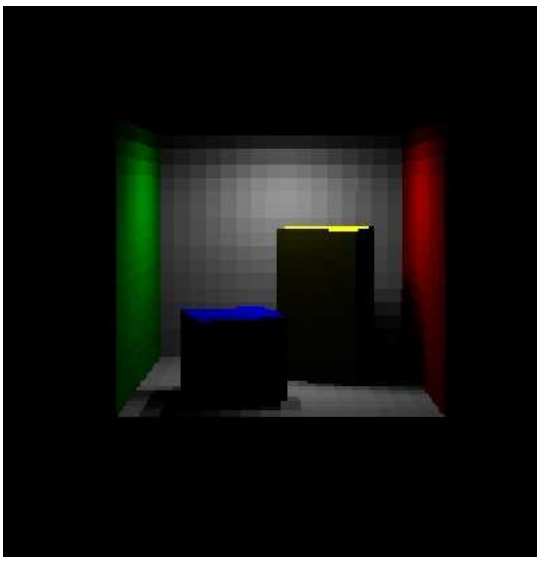

Figure 2: Cornell Box Rendering From Progressive Refinement on a GPU

Figure 2 shows a rendering of the Cornell Box with 1024 scene patches. The hemicube is 256×256 on the top, and 256×128 on each side. The simulation took 84 seconds to run 200 iterations of progressive refinement radiosity on a 600 MHz Pentium III with 384 MB RAM. The GPU is an ATI RADEON 9700 Pro with 128 MB RAM.

Typically, the lighting in the image is smoothed with Gouraud shading. In this example the patches are left square to indicate the resolution of the computation.

9 Future Work

There are many opportunities for further improvements to the existing system, and there are many unanswered questions. Some pressing issues are:

1. It is possible that some of the primary PRRoG equation can be done in a vertex shader for performance optimization. In most cases vertex shaders should execute fewer times per frame than pixel shaders. It is also possible to implement the projective texture mapping step to multiply by the delta form factor in the fixed function pipeline. This might be faster than a programmable shader.
2. More test scenes are needed. Currently, only the Cornell Box has been rendered with PRRoG.
3. Gouraud shading or light maps should be added to give smooth renderings.
4. PRRoG should be compared to other radiosity rendering packages, especially other implementations of progressive refinement radiosity.
5. Some metric should be added to determine convergence of the radiosity solution for the scene. Currently, I just notice that the rendered scene does not change substantially after N iterations.
6. PRRoG uses the constant basis function for patch radiosity. Ideally, it would extend to something more sophisticated.
7. Can substructuring be implemented on a GPU?
8. Is there any benefit in this application to storing radiosity and delta radiosity as textures? Is this feasible to implement on the the current generation of GPUs?

References

[Cohen and Greenberg 1985] Michael F. Cohen and Donald P. Greenberg. The Hemi-Cube: A Radiosity Solution for Complex Environments. *Computer Graphics, Volume 19, Number 3 (Proceedings of SIGGRAPH 1985).*

[Cohen et al. 1988] Michael F. Cohen, Shenchang Eric Chen, John R. Wallace and Donald P. Greenberg. A Progressive Refinement Approach to Fast Radiosity Image Generation. *Computer Graphics, Volume 22, Number 4 (Proceedings of SIGGRAPH 1988).*

[Cohen and Wallace 1993] Michael F. Cohen and John R. Wallace, *Radiosity and Realistic Image Synthesis*, Morgan Kaufmann, Inc., 1993.

[Coombe et al. 2003] Greg Coombe, Mark Harris and Anselmo Lastra. Radiosity on Graphics Hardware. Technical Report from Department of Computer Science, University of North Carolina at Chapel Hill. Coombe UNC TR03-020. ftp://ftp.cs.unc.edu/pub/publications/techreports/03-020.pdf

[Glassner 1995] Andrew S. Glassner. *Principles of Digital Image Synthesis*, pp. 887-986. Morgan Kaufmann, Inc., 1995.

[Harris 2001] Mark Harris, Point-based Radiosity, Project for a graduate-level computer graphics course in the Department of Computer Science at UNC-Chapel Hill.

[Sillion and Puech 1994] François Sillion and Claude Puech, *Radiosity and Global Illumination*, Morgan Kaufmann, Inc., 1994.

Acknowledgments

I would like to thank Hanspeter Pfister (MERL), John F. Hughes (Brown University), Frédo Durand (MIT), Morgan McGuire (Brown University), and Mark Harris (University of North Carolina - Chapel Hill), for helpful conversations. Both the Cornell Box model and the MDL library, which is used to load the model, are from the Cornell University Program of Computer Graphics.

Source Code

The source code for this project is available at the author's web page:
`http://www.cs.brown.edu/~brl/prrog/`