PERSPECTIVE-DRIVEN RADIOSITY ON GRAPHICS HARDWARE

A Thesis

by

JUSTIN TAYLOR BOZALINA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2011

Major Subject: Computer Science

PERSPECTIVE-DRIVEN RADIOSITY ON GRAPHICS HARDWARE

A Thesis

by

JUSTIN TAYLOR BOZALINA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2011

Major Subject: Computer Science

ABSTRACT

Perspective-Driven Radiosity on Graphics Hardware. (May 2011)

Justin Taylor Bozalina, B.S., Texas A&M University

Chair of Advisory Committee: Dr. Glen Williams

Radiosity is a global illumination algorithm used by artists, architects, and engineers for its realistic simulation of lighting. Since the illumination model is global, complexity and run time grow as larger environments are provided. Algorithms exist which generate an incremental result and provide weighting based on the user's view of the environment. This thesis introduces an algorithm for directing and focusing radiosity calculations relative to the user's point-of-view and within the user's field-of-view, generating visually interesting results for a localized area more quickly than a traditional global approach.

The algorithm, referred to as perspective-driven radiosity, is an extension of the importance-driven radiosity algorithm, which itself is an extension of the progressive refinement radiosity algorithm. The software implemented during research into the point-of-view/field-of-view-driven algorithm can demonstrate both of these algorithms, and can generate results for arbitrary geometry. Parameters can be adjusted by the user to provide results that favor speed or quality.

To take advantage of the scalability of programmable graphics hardware, the algorithm is implemented as an extension of progressive refinement radiosity on the GPU, using OpenGL and GLSL. Results from each of the three implemented radiosity algorithms are compared using a variety of geometry.

Dedicated to the memory of my father, Michael Bozalina.

ACKNOWLEDGEMENTS

I graciously thank my advisor, Dr. Glen Williams, for allowing me the opportunity to work on the Texas A&M Autonomous Ground Vehicle project, and for continuing to offer guidance and encouragement throughout my graduate classes and subsequent in-absence research. I am grateful to my committee members: Dr. John Keyser, for all his help and instruction during my undergraduate and graduate studies, and Dr. Make McDermott for his patience with my lack of automotive knowhow during the autonomous vehicle project.

Many thanks are given to my parents, Glenda and Michael Bozalina, who supported me through seven years of classes without asking anything in return.

Finally, this thesis could not have been completed without the love and support of my wife, Marie, who sacrificed many weekends and evenings to allow me to continue my work.

NOMENCLATURE

2D	Two-Dimensional
3D	Three-Dimensional
Cg	C for Graphics
CPU	Central Processing Unit
FBO	Frame Buffer Object
FOV	Field-of-View
GLSL	OpenGL Shading Language
GPGPU	General Purpose Computing on the GPU
GPU	Graphics Processing Unit
HLSL	High Level Shading Language
IDE	Integrated Development Environment
MFC	Microsoft Foundation Classes
MRT	Multiple Render Targets
NPOT	Non-Power-of-Two
OBJ	A file format for 3D models, developed by Wavefront
PBO	Pixel Buffer Object
POV	Point-of-view
SDK	Software Development Kit
SIMD	Single Instruction, Multiple Data
STL	Standard Template Library

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

1. INTRODUCTION

Realistic illumination of surfaces is a fundamental problem in the field of computer graphics. The desire to emulate real-world lighting conditions has been of interest to researchers for several decades, and continues to provide a source of computing challenges. Speed, quality, and artistic control are all qualifications considered when discussing lighting in computer graphics.

Global illumination is the concept of achieving realistic lighting results based not only on the effect of light on a surface, but also taking into account the influence of light radiating from each surface in the environment to every other surface in the environment. By definition, this is a problem whose solution demands $O(n^2)$ complexity.

Radiosity is one such view-independent global illumination algorithm, which models the interaction of light between Lambertian surfaces [1]. Although completely solving a radiosity solution remains $O(n^2)$, algorithms have been introduced which generate viewable results iteratively with $O(kn)$ complexity, where k is the number of iterations through the algorithm [2-3].

In the past decade, the GPU has emerged as a powerful coprocessor for general purpose, parallel computing. This general-purpose computation using a GPU is referred to as GPGPU. In addition to other global illumination algorithms, the radiosity algorithm has been implemented using GPGPU [4].

This thesis introduces an extension of the GPU-based progressive refinement radiosity algorithm, referred to as "perspective-driven radiosity," which limits radiosity calculations to a user-specified area-of-interest. This area-of-interest is defined by the point-of-view (POV) and field-of-view (FOV) of the user's view of the scene. A user interface is supplied to allow changes to parameters used in radiosity calculations.

This thesis follows the style of *IEEE Transactions on Visualization and Computer Graphics*.

1.1 Objectives

The purpose of this thesis, as stated in the thesis proposal, is to present an algorithm that limits radiosity calculations to the surfaces in the user's view. The implementation details outlined in the thesis proposal require the algorithm to execute primarily on the GPU. Features of the implementation include:

i. Loading arbitrary geometry of environments modeled in Wavefront OBJ format, with diffuse surface color and light sources specified as material properties.
ii. An implementation of progressive refinement radiosity on the GPU.
iii. An extension of progressive refinement on the GPU to importance-driven radiosity on the GPU.
iv. An extension of importance-driven radiosity on the GPU to perspective-driven radiosity on the GPU.
v. A user interface that allows control of speed and quality-affecting parameters.
vi. User view interaction through mouse control of a camera.

1.2 Significance

Algorithms have been introduced which weight radiosity calculations based on the user's view of the environment. The perspective-driven radiosity algorithm is unique in that none of these algorithms limit calculations to the user's view. By restricting computation to the surfaces encompassed by the user's view, the radiosity calculations are solved for the area of most immediate importance. Additionally, this thesis shows that the progressive refinement radiosity algorithm on the GPU can be extended to additional radiosity algorithms.

1.3 Outline

The radiosity algorithm is nearly thirty years old at the time this thesis is written. Although this thesis assumes a certain degree of familiarity with computer graphics, an explanation of radiosity will be provided. Additionally, the basic concepts of GPU computing will be introduced. A review of topics related to radiosity, global illumination, and the utilization of the GPU for implementing global illumination will be presented.

Next, technical aspects of the software implementation will be shown, and an explanation of the choice of each technology will be given. Technologies include the use of MFC for application development, C++ for the programming language, OpenGL for the interactive graphics API, and GLSL for the shading language.

The progressive refinement radiosity on the GPU algorithm will be the first radiosity algorithm examined. An in-depth analysis of the techniques used for the implementation of this algorithm will follow, including the fragment shader computation of form factors, which is integral to all radiosity algorithms. Attention will be paid to the GPU techniques used in this implementation that are later applied to importance-driven and perspective-driven radiosity on the GPU.

The extension of progressive refinement radiosity on the GPU to importance-driven radiosity on the GPU follows. Technical details relevant to the perspective-driven algorithm will be examined, including weighting based on the user's view, and propagation of this weight through the environment using concepts from progressive refinement radiosity.

Next, the perspective-driven radiosity on the GPU algorithm will be introduced. The implementation of this algorithm depends heavily on the implementation of the progressive refinement and importance-driven algorithms. Limiting the radiosity calculations to the user's view is accomplished by storing a table of radiosity values that should contribute to the illumination of surfaces outside the user's view. The area-of-interest is gradually expanded over time, to ensure computation is not wasted.

Following the presentation of each radiosity algorithm used, results will be presented from the experimentation with the software implementation of each algorithm. Each algorithm will be compared in scenarios involving the generation of radiosity lighting for a single-enclosure environment, for an environment comprised of multiple enclosures, for an environment where a single enclosure has been subdivided, and for an environment where multiple enclosures have been subdivided.

Finally, conclusions reached during research are given, along with suggestions for future work that could improve performance.

2. BACKGROUND

Radiosity is a global illumination algorithm with decades of research invested in improving the speed and quality of its results. The extension of radiosity to the GPU is a more recent development, and the concepts involved in GPGPU are relatively new.

2.1 Radiosity

Many improvements to the radiosity algorithm have been introduced. The algorithms most relevant to this thesis are progressive refinement radiosity and importance-driven radiosity.

2.1.1 Classical Formulation

Radiosity is based on principles of radiative heat exchange from thermal engineering [1]. Most commonly, an environment of N surfaces is subdivided into a collection of differential elements. The radiosity of surface i, B_i, can be described as

$$B_i = E_i + \rho_i \sum_{j=1}^{N} B_j F_{ij}$$

where E_i is the emissive energy of surface i, ρ_i is the reflectivity of the surface i, and F_{ij} is the form factor representing the fraction of energy leaving surface j and impinging on surface i.

The form factor formulation for two finite surfaces i and j, F_{ij}, is defined as

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{cos\varphi_i cos\varphi_j}{\pi r^2} dA_j dA_i$$

where A_i is the area of surface i, φ_i and φ_j are the angles between the surface normals of the surfaces and the line between them, dA_i and dA_j are elemental areas of the surfaces, and r is the distance between them.

If the first equation is to be solved for B_i for every surface in the environment of N surfaces, it can be represented as a matrix of radiosity equations

$$\begin{bmatrix} 1-\rho_1F_{11} & -\rho_1F_{12} & \cdots & -\rho_1F_{1N} \\ -\rho_2F_{21} & 1-\rho_2F_{22} & \cdots & -\rho_2F_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_NF_{N1} & -\rho_NF_{N2} & \cdots & 1-\rho_NF_{NN} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_N \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_N \end{bmatrix}$$

Since ρ is a scalar, the system of equations must be solved for each of the three-color components in an RGB color model. The resulting vector of radiosity values can be mapped to generate a view-independent image of the lit environment.

2.1.2 Progressive Refinement

The most significant improvement to radiosity lighting algorithms is progressive refinement radiosity: an iterative approach that takes the concept of gathering radiosity to surfaces by solving the radiosity equation matrix and reverses it, in favor of "shooting" radiosity from a single surface to every other surface in the environment [3].

The original formulation of the radiosity equation solves for the radiosity of surface i, B_i. Progressive refinement instead states that surface i can be thought of as contributing to all surfaces in the environment, such that for a surface j

$$B_j = B_j + \rho_j B_i F_{ij}\, A_i/A_j$$

where each term holds its definition established in the classical radiosity equation. The initial radiosity value for a surface i, B_i, is set to the emissive value E_i if the surface is a light, or it is set to zero for non-light sources. Once a surface receives radiosity, it can distribute that received radiosity back to the rest of the surfaces in the environment that are not occluded from the shooting surface when looking in the direction of the surface normal.

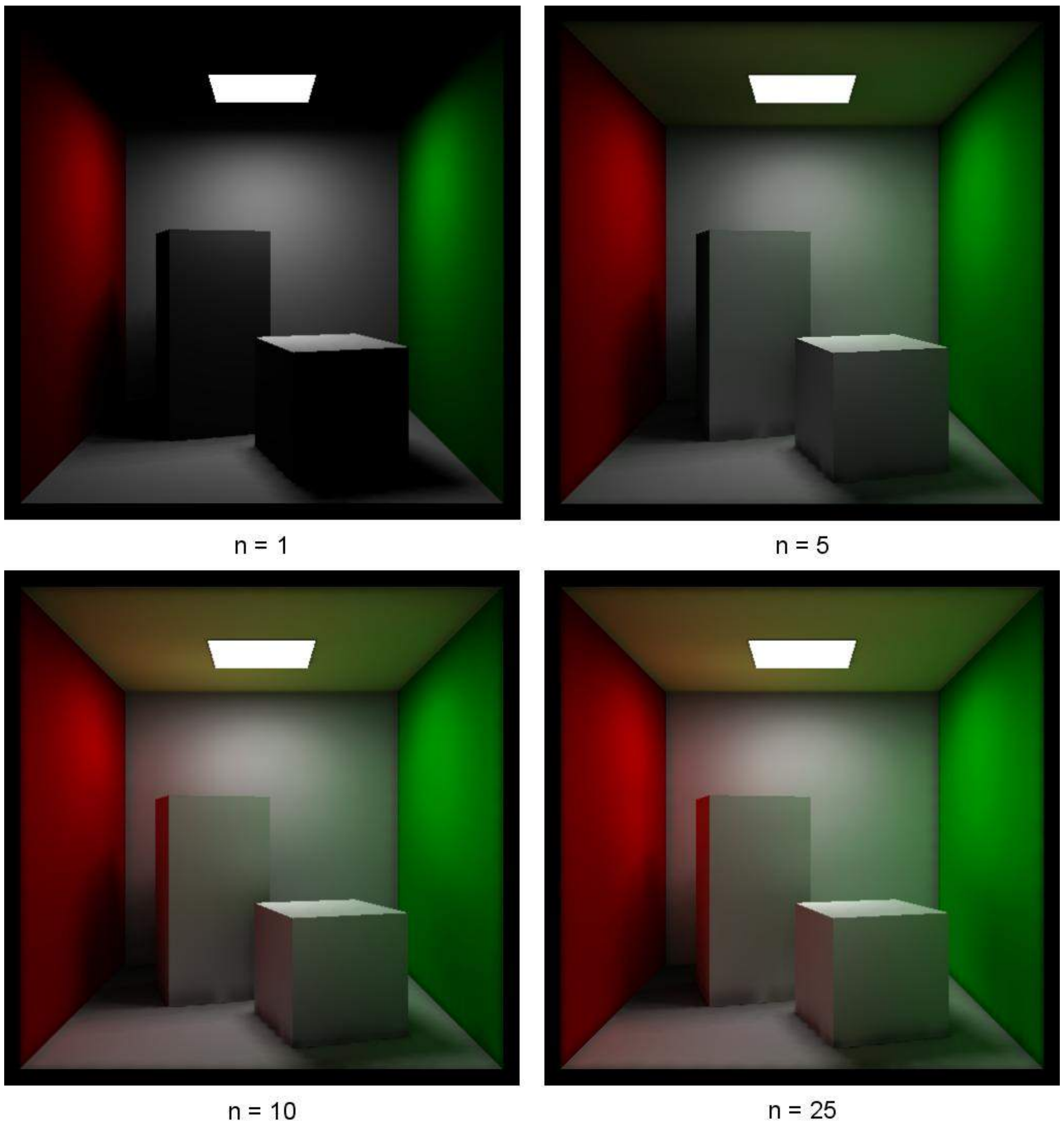


Figure 1: Radiosity textures after n progressive refinement radiosity iterations

Each time a surface contributes its radiosity to all other non-occluded surfaces in the environment, an image can be generated from the resulting radiosity values. Initially, these images represent an incomplete solution. Through continued iteration of each surface in the environment and accumulation of radiosity values, a more complete solution is eventually reached. Figure 1 shows an image generated using several iterations of progressive refinement. The user may stop the iterations at any time, once a satisfactory image has been reached. The resulting lit environment remains view-independent.

The largest contribution towards the convergence of a desirable image will be made by surfaces with the highest amount of energy. Therefore, it is desirable to select surfaces that will contribute the most energy for a single iteration. This can be accomplished by selecting the surface i with the largest unshot radiosity energy

$$\Delta B_i A_i$$

where ΔB_i represents radiosity the surface has received from all other surfaces that has not yet been distributed to any other surfaces, or “unshot” radiosity. Using this approach, an image that represents a close approximation of the actual solution may be reached in one or two iterations.

2.1.3 Importance-Driven

Importance-driven radiosity introduced the idea that a surface can be selected for shooting radiosity based on criteria other than its total energy, since view-independent radiosity algorithms “over-solve globally and under-solve locally” [5]. The algorithm uses the concept of an importance value being propagated through the environment using the familiar radiosity equation

$$I_i = R_i + \rho_i \sum_{j=1}^{N} I_j F_{ij}$$

where I_j is the importance of the surface and R_i is directly received importance. R_i is initialized by an infinitesimally small camera surface, which is the source of directly received importance. Importance is transported in a manner that is dual to radiosity in this way, as depicted in Figure 2.

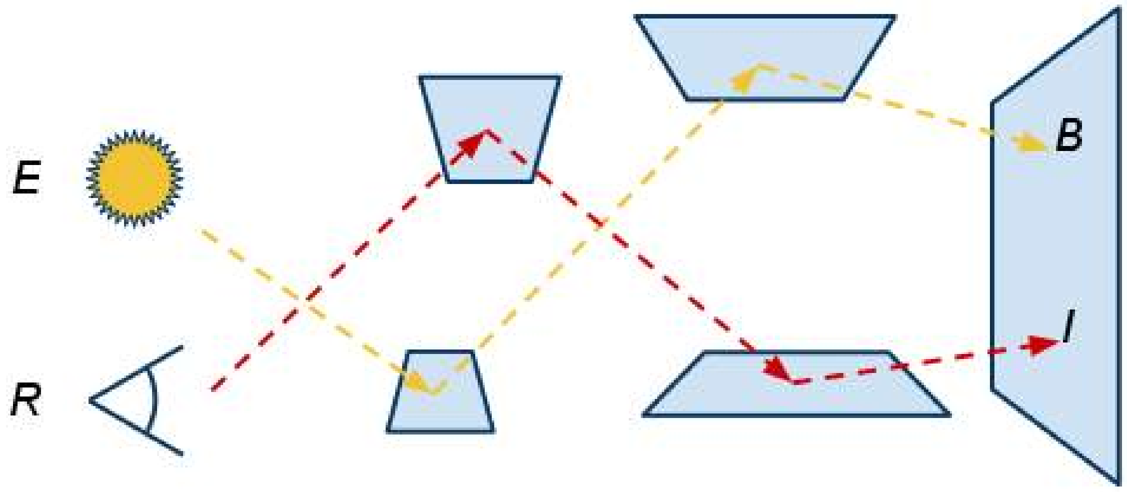


Figure 2: The duality of radiosity and importance [5]

The importance-driven algorithm was developed to determine which area of the environment should have the highest resolution radiosity data when computing a radiosity solution. It was later extended to the selection of an optimal shooting surface during progressive refinement radiosity [2]. By weighting the total energy of a surface with its total importance, the surface that will have the greatest contribution to image convergence can be selected. The shooting surface selection equation becomes

$$I_i \Delta B_i A_i$$

where the optimal shooting surface maximizes the resulting value.

2.2 GPGPU

GPU programming has advanced significantly in the past decade as more features have been introduced to commercial graphics processors. The software implementation uses the GPU to display textures containing radiosity values that have been mapped to the environment via the traditional graphics pipeline. GPGPU is used for radiosity calculations, including the computation of form factors and the storage of bookkeeping data structures. Advanced techniques possible with GPGPU are beyond the scope of this thesis, but the basic principles common to GPU programming are relevant.

2.2.1 Textures

A texture represents GPU memory, arranged two-dimensionally. Floating-point texture formats provide the precision necessary for GPGPU. Texel values are one to four-dimensional vectors that correspond to RGBA color channels or XYZW homogeneous coordinates. The GPU performs operations on each vector channel in a texel simultaneously [6].

Texture data is analogous to an array in CPU memory, and is indexed using texture coordinates. Non-rectangular textures are indexed using normalized coordinates in the range [0,1], shown in Figure 3.

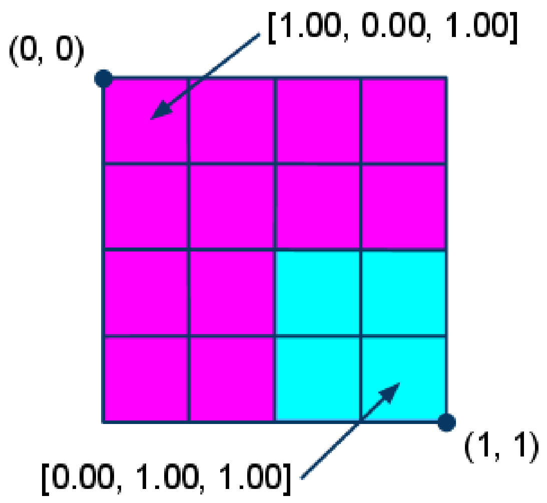


Figure 3: Floating point RGB texture with normalized texture coordinates and color channel values

The software implementation uses an OpenGL frame buffer object (FBO) to perform computations on a texture. An FBO allows direct rendering to a texture.

Texture memory can be in either "read" or "write" mode, but not "read/write." Therefore, to perform computations using data in a texture, the source texture must be used as an input, and output values must be written to another texture of equivalent size and format.

For example, in Figure 4, to perform a general computation using the right texture as input, the output values of the computation are written to a different output texture, attached to an FBO. General computation on the GPU will be explained in Section 2.2.2.

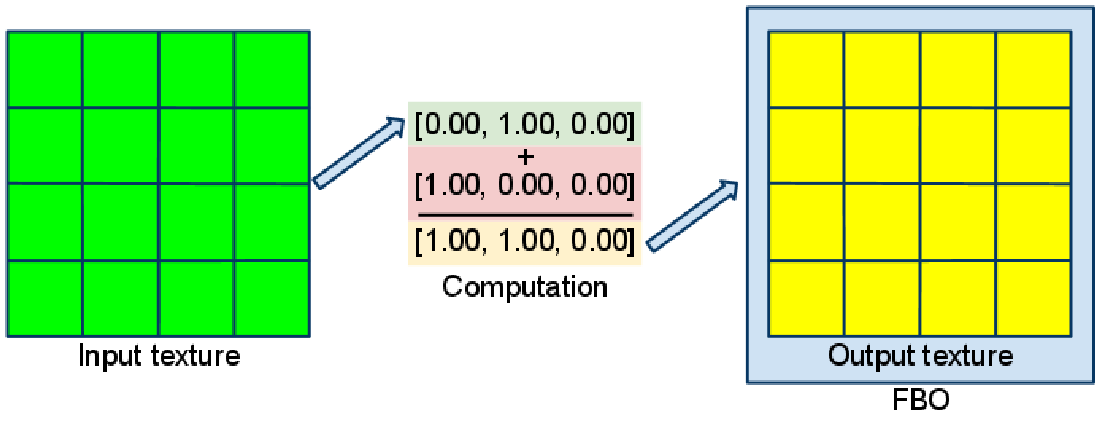


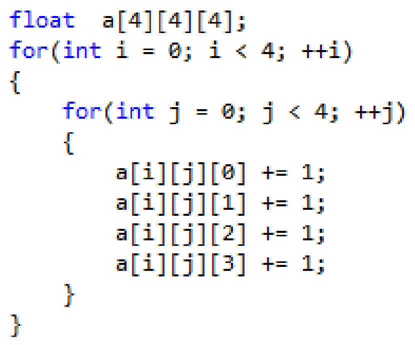
Figure 4: Computation performed on an input texture written to an output texture

Texture data may be transferred between graphics memory and system memory, but this is a time-intensive operation and is avoided if possible. The software implementation initializes textures using initial values as the color input to a draw operation, rather than sending values from system memory.

2.2.2 Shaders

The next crucial component of GPGPU programming is shaders. A shader is a small program the GPU runs during a draw operation. In the software implementation, two different kinds of shaders are used. Vertex shaders transform the vertices of input geometry; fragment shaders modify the value of a pixel before it is written to a draw buffer. Shaders can be used to produce complex render effects, but they can also be used for general-purpose parallel computation.

When programming in C++, computations involving a two-dimensional (2D) array with dimensions [4,4] containing 4-tuples similar to the texture in Figure 3 are performed with two `for` loops, shown in Figure 5.

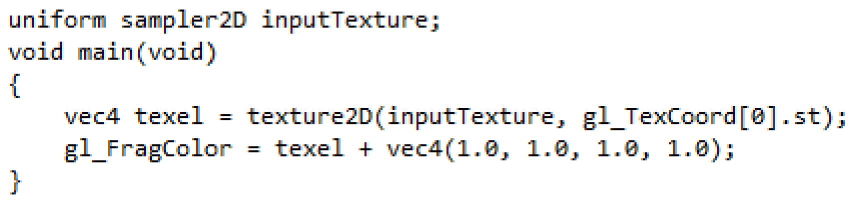


```
float  a[4][4][4];
for(int i = 0; i < 4; ++i)
{
    for(int j = 0; j < 4; ++j)
    {
        a[i][j][0] += 1;
        a[i][j][1] += 1;
        a[i][j][2] += 1;
        a[i][j][3] += 1;
    }
}
```

Figure 5: Computation on a 2D array of 4-tuples in C++ using two for loops

Each iteration through the loop will be performed sequentially when executed on the CPU. Additionally, each component of a 4-tuple must be indexed separately.

To perform an analogous operation on a 2D texture with dimensions [4,4] in GPU memory, a fragment shader is written to take the value of the input texel and add the value of 1 to each color channel. Figure 6 contains GLSL code, detailed in Section 3.3.



```
uniform sampler2D inputTexture;
void main(void)
{
    vec4 texel = texture2D(inputTexture, gl_TexCoord[0].st);
    gl_FragColor = texel + vec4(1.0, 1.0, 1.0, 1.0);
}
```

Figure 6: Computation on a 2D, 4-channel texture in GLSL using a fragment shader

The fundamental difference between CPU and GPU computing is that this shader operation is performed in parallel for every output value generated on the GPU. This principle is known as “single instruction, multiple data,” or SIMD. When shading the input texture, the shader itself is considered the “loop,” since it performs the computation for each texel in the texture. In this manner, a shader can be thought of as a kernel function, applied to each texel in the texture as a whole. This is the implementation of the computation step in Figure 4.

Since instructions operating on the input texture data are executed in parallel, no output values can have data dependencies on other output values. Any texel in the input texture, however, can be accessed in a single shader operation [6].

2.2.3 Drawing

Computation on a texture using a shader is performed by executing a draw operation in OpenGL using the full range of the normalized texture coordinates. Before the draw operation takes place, a texture is bound as input to a fragment shader, and an output texture is attached to an FBO. The shader is enabled and a polygon is drawn, which shades the value of each texel in the output texture. Each texel in the input texture must correspond to a pixel output by the fragment shader for computation to be performed correctly. To achieve this one-to-one mapping of input to output values, both the projection and viewport used in the draw operation must be considered.

The input and output data is 2D, therefore, the projection must also be 2D. This corresponds to an orthographic projection in OpenGL. The dimensions of the drawn polygon will match the extents of the orthographic projection. By achieving a mapping of each input texel to each output texel, all input values in the texture will be processed by the shader.

For the output texture to receive all shaded pixel values, the viewport used for rendering must match the dimensions of the output texture. In Figure 7, a quad with dimensions [1,1] is drawn into an orthographic projection with dimensions [1,1], using normalized texture coordinates. A shader takes each fragment from the draw operation, performs computation on the value using a texture with dimensions [4,4] as an input parameter, and writes the computed value to an output texture with dimensions [4,4], with a viewport set to equal dimensions.

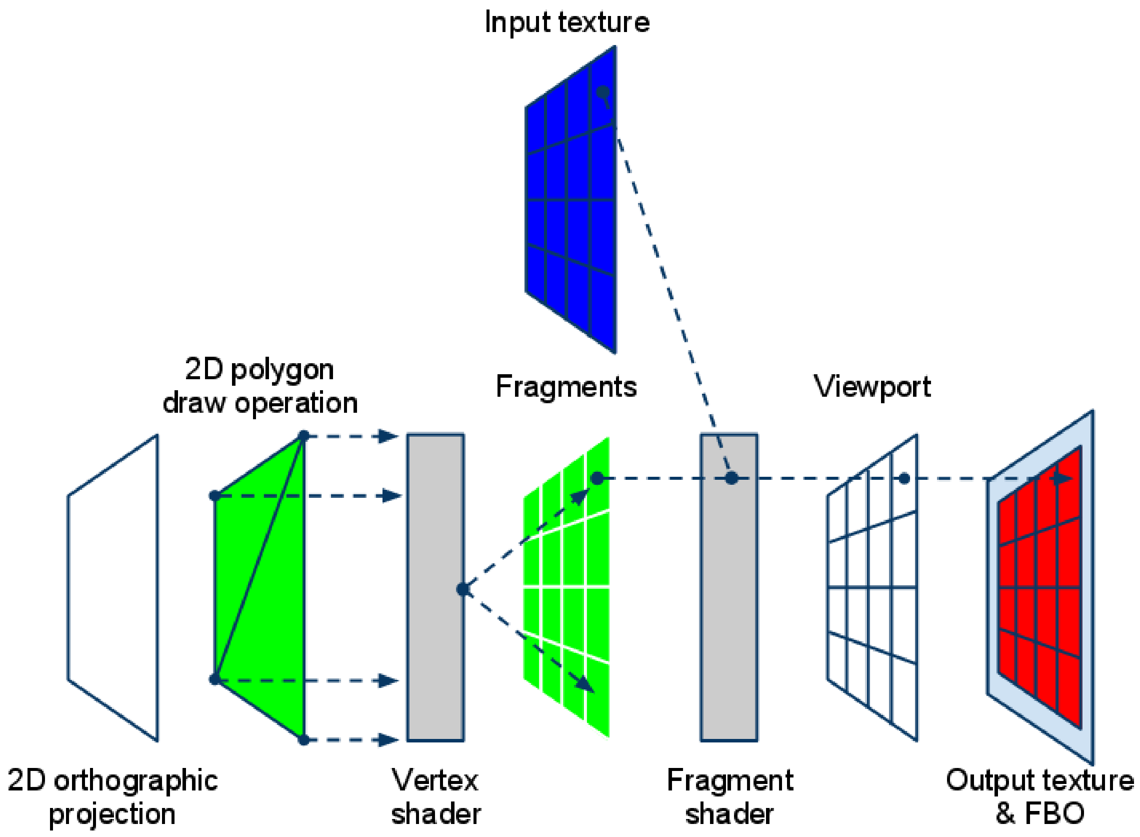


Figure 7: One-to-one mapping of input texture data to shaded output data through the shader pipeline

By using these basic operations, a shader can perform complex computation on an input data set using multiple render passes. Continuously swapping the input and output textures, the results of each render pass can be used as the argument for the consecutive render pass. In GPGPU, this is referred to as "ping ponging" [6].

2.3 Previous Work

Radiosity is a well-developed research topic, and a wealth of literature exists concerning implementation details and improvements to the algorithm. Radiosity, however, is not the only global illumination technique. Modeling the interaction of light with an environment is a complex problem with many solutions, some of which have also been implemented using the GPU.

2.3.1 Radiosity

As stated previously, radiosity was presented as a means of transferring light between surfaces in a closed environment. Early environments using radiosity were simple, as the calculation of form factors and the determination of surface visibility was computationally intensive.

The hemicube was introduced as a comprehensive means of determining surface to surface reflections [7]. This technique was used to solve complex environment radiosity solutions when the progressive refinement algorithm was introduced [3]. The progressive refinement algorithm has been augmented by other global illumination techniques, such as ray tracing, which attempt to improve the results and performance of radiosity lighting [8-10]. Radiosity itself can generate the diffuse lighting for environments to be used with real-time dynamic illumination algorithms, such as precomputed radiance transfer [11].

The precision of radiosity solutions has attracted lots of attention, particularly in improving the appearance of areas where the gradient of radiosity exceeds a certain threshold. Radiosity precision has been improved using adaptive environment meshing and hierarchies of surfaces for areas of high detail [12-16]. The original importance-driven algorithm is one such technique which refines the mesh in areas that are important to the user's view [5].

Optimizing the precedence of radiosity calculations using visual importance has been the goal of this thesis and other work. The importance-driven algorithm was first applied to progressive refinement by selecting which shooting surface will have the most visually important effect towards image convergence for the user's view [2]. Limiting radiosity calculations to an area-of-interest defined by the camera FOV angle was also attempted at Texas A&M University, using the CPU to store vectors of radiosity values which could be contributed to the image when needed [17].

2.3.2 GPU-Based Global Illumination

Since radiosity was first introduced, research has attempted to improve the speed of the radiosity solution and keep the results accurate. Advances in graphics hardware trivialized the applicability of the radiosity solution for complex environments with hidden surfaces, by using real-time graphics systems to generate each plane of the hemicube. Hardware-accelerated hemicube generation also simplified form factor computation when environments projected on the hemicube walls were discretized from continuous surfaces to rasterized pixels with pre-computed form factors. Additionally, solving the radiosity equations for surfaces in parallel was approached using networked computers and parallel processing systems [18-21].

In addition to progressive refinement radiosity, several other global illumination techniques have been implemented using the GPU. With the introduction of floating-point textures, classical radiosity was shown to run on the GPU with the use of an iterative Jacobi matrix solver. This technique accompanied the demonstration of subsurface scattering using the GPU [22].

Ray tracing and ray casting were shown to run at increased speeds relative to the CPU when using programmable graphics hardware [23-24]. These early tests encouraged the use of the GPU for general purpose computing applied to computer graphics. The introduction of GPGPU allowed the preceding radiosity hardware acceleration techniques and parallel processing techniques to be combined by simplifying the hemicube generation to one hemisphere rendering pass and calculating form factors for a surface using scalable stream processors [4].

Ambient occlusion was also applied to models using the GPU. When combined with a pre-processing step, the ambient occlusion light model could be approximated dynamically at an interactive framerate [25]. Finally, photon mapping was implemented using the GPU, although its performance was initially compute bound because of the performance of early floating-point textures [26]. The introduction of newer graphics hardware and improved floating-point operations allowed this algorithm to scale in performance.

3. TECHNICAL SPECIFICATION

The development of the software used to demonstrate the perspective-driven algorithm required the selection of multiple technologies and libraries.

3.1 Graphics API

At present, the two major industry standards for real-time graphics development are OpenGL, by the Khronos Group, and Direct3D, by Microsoft [27-28]. OpenGL was chosen, owing to ease of development and the ability to use extensions.

OpenGL is a cross-language, cross-platform graphics API [29]. It is commonly used for scientific visualization and computing, because of its operating system independence. New and experimental technologies can be added to OpenGL using extensions. Core functionality is advanced with each version release, most of which are backwards compatible.

Terminology and concepts related to the core functionality and extensions used in this research are outside of the scope of basic OpenGL. They are presented in this section because of their relevancy to the discussion of radiosity algorithms on the GPU.

3.1.1 Core Functionality

The software implementation was developed against the OpenGL 2.1 specification [30]. If a hardware vendor provides support for a specific version of OpenGL, they must implement core behavior outlined in that version's specification. The following native OpenGL 2.1 functionality was used as a supplement to the Microsoft implementation of OpenGL 1.1 supported by the Visual Studio IDE.

3.1.1.1 Volume Textures

Volumetric data can be displayed using three-dimensional (3D) volume textures. In the software implementation, 3D textures are used in conjunction with the texture array extension to provide indexed access to a 3D stack of 2D textures. The use of this functionality will be explained further in Section 3.1.2.6.

3.1.1.2 Multi-Texturing

Multiple texture states and the mapping of multiple textures to a single polygon are allowed with multi-texturing. This also permits binding multiple textures as inputs to a shader. Any function that requires more than one texture argument to complete its computation makes use of multi-texturing.

3.1.1.3 Multiple Render Targets (MRT)

Some shaders perform computation that is relevant to more than one output texture. Binding textures one at a time and duplicating render passes also duplicates work already performed. With MRT, a shader can write multiple output values, avoiding the need to re-compute identical results for an additional output.

3.1.1.4 Pixel Buffer Objects (PBO)

Sending texture memory from the GPU to the CPU and vice-versa is a time-consuming operation, as memory must be sent across the graphics bus. A PBO allows graphics memory to be mapped to an address in CPU space. Reading texture data to CPU memory and writing CPU memory to textures can occur asynchronously.

3.1.1.5 Occlusion Queries

Hardware occlusion queries allow a program to retrieve a count of the number of pixels that were output to the draw buffer during rasterization of a primitive. Primitives in OpenGL are the basic shapes that are output to the screen when interpreting incoming vertices. An occlusion query that returns a pixel count equal to zero indicates that the primitive could not be “seen” by the current screen projection.

3.1.1.6 Non-Power-Of-Two (NPOT) Textures

Originally, textures were restricted to dimensions defined by powers of two. The relaxation of this restriction makes GPGPU less cumbersome, since only the space required for a 2D dataset must be defined.

3.1.1.7 Shaders

As stated in Section 2.2.2, shaders have enabled general computation on the commercial GPU. The software implementation uses shaders almost exclusively for computation. A simple shader is used to augment the displayed texture, as explained in Section 4.1.6.

3.1.2 Extensions

The OpenGL API is extended anytime a hardware vendor wishes to introduce and promote new functionality. As a feature matures and new hardware is released, an extension can be promoted to core functionality. The software implementation uses the following extensions, all of which have since been promoted to core as of OpenGL 3.1 [31].

3.1.2.1 ARB_texture_float

This extension adds 32-bit and 16-bit floating-point texture capabilities to OpenGL [32]. Any shader operations on a floating-point texture's texel values will result in floating-point output values; however, only values in the range [0,1] can be viewed on-screen. When used for data storage, they can contain any valid floating-point values, with one caveat, explained in Section 3.1.2.3.

3.1.2.2 EXT_framebuffer_object

Textures are the primary data structure in GPGPU. This software implementation uses textures for both computation and display. As described in Section 2.2.3, drawing to a texture is the mechanism by which shaders access input data and write out computed values. An FBO allows a texture to be used as a drawing destination in a simple and efficient manner, just as a buffer is used to display draw operations on-screen [33].

3.1.2.3 ARB_color_buffer_float

Before floating-point texture support was introduced to the GPU, the fixed-function graphics pipeline used fixed-point data types to represent texture values. The default behavior of the standard pipeline clamps texture values to the range [0,1], to accommodate these fixed-point values. Using this extension, floating-point textures can store negative values, or values greater than one [34].

3.1.2.4 ARB_texture_rectangle

Textures are accessed using texture coordinates, similar to the indexing of data in an array. Traditionally, texture coordinates for a 2D texture are normalized to the range [0,1]. Rectangular textures are NPOT textures which are indexed with non-normalized texture coordinates in the range $[w, h]$, where w is the width of the texture, and h is the height [35]. An example of this kind of texture coordinates is shown in Figure 8. These textures are useful when performing operations where specific individual texels are indexed in a shader.

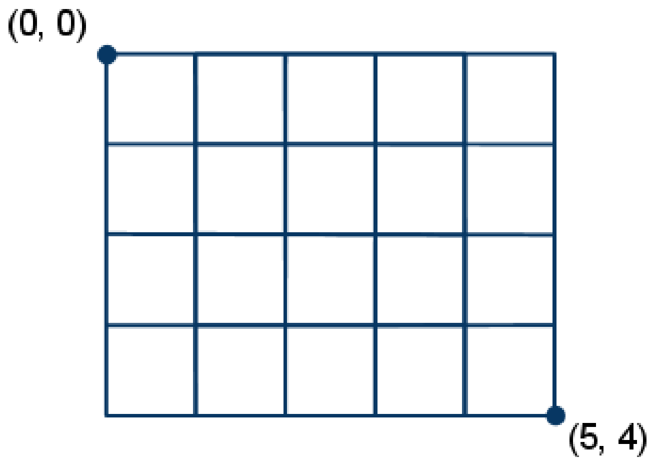


Figure 8: Rectangular texture with dimensions [5,4] showing texture coordinates

3.1.2.5 EXT_texture_integer

In fixed-function OpenGL textures, textures are stored as fixed-point values. Although the actual data storage type for these textures can be integer type, these integer values represent normalized fixed-point values, and do not return expected integer values if accessed [36]. True integer support is added with this extension. An integer texture can be attached to an FBO, but cannot be used in draw operations without the assistance of a fragment shader to interpret the integer value.

3.1.2.6 EXT_texture_array

A texture array is a collection of textures of the same dimensions and format, which can be indexed in the range $[0, n-1]$, where n is the number of layers in the texture array [37]. Texture arrays can be represented by two-dimensional or three-dimensional texture data. If two-dimensional, it is referred to as a 1D texture array: a collection of 1D textures. Similarly, a 2D texture array is three-dimensional. In Figure 9, the third layer in a 2D texture array is accessed using a zero-based array index.

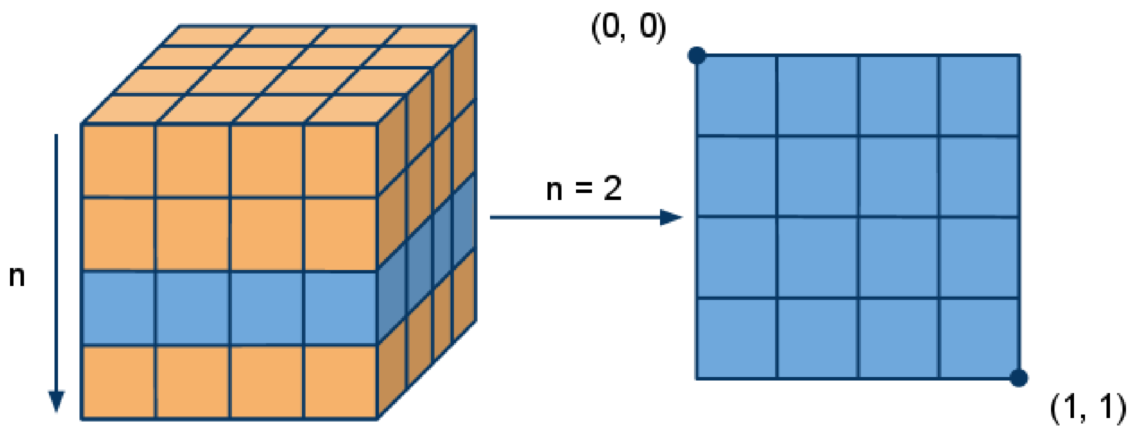


Figure 9: 3D texture volume as 2D texture array with a depth value of four

Once a layer in a 2D texture array is accessed, the returned texture layer is indexed as a 2D texture, and can be bound to an FBO.

3.1.2.7 EXT_gpu_shader4

New GPU technology requires additions to the OpenGL API, and the means to utilize the new functionality in a shader. This extension exposes the shader-based means of utilizing texture arrays and integer textures. It also includes an API function to specify the binding location for a true integer output value of a shader operation, since integers cannot be used in the normal fixed-function pipeline [38].

3.2 Programming Language

Because OpenGL is an open standard, it is supported on all major operating systems, and bindings to the API are available for many programming languages [29]. Because of this flexibility, the use of a specific programming language is not a foregone conclusion. For the software implementation, the two programming languages considered were C/C++ and C#.

At present, C# is an attractive choice for programming on a Microsoft Windows platform. With each release of the Visual Studio IDE, Microsoft continues to improve programming tools that support C# and the .NET framework. Additionally, several open source projects for OpenGL bindings in C# allow easy integration with managed applications [39-40].

Official support from the Khronos Group on the OpenGL website is available in the form of an extension registry written for C/C++ [41]. Since the software implementation needed active support for extensions to the OpenGL specification, C++ was selected as the development language.

3.3 Shading Language

Two shading languages were considered for this research: GLSL, by the Khronos Group, and Cg, by NVIDIA [42-43]. Cg is the same language as HLSL, developed by Microsoft for use with Direct3D. It is cross-platform and uses a C-like syntax. Cg is a versatile platform that can build shaders for use with Direct3D or OpenGL. Additionally, shader assembly files from the Cg compiler can be examined. NVIDIA offers powerful debugging tools for use with Cg development through their support of CgFX.

Cg requires the presence of the Cg Runtime Libraries and the use of the Cg API for application development. GLSL is supported natively in OpenGL 2.1 and accepts shaders in the form of character strings. Whereas Cg uses input and output binding syntax for shader arguments, GLSL has built-in constants and variables.

Native data types in GLSL include primitive data types such as Boolean, integer and floating-point, as well as primitive vectors of length one to four, matrices, and textures, among others. Common hardware-accelerated mathematics functions can be used, such as the exponential operator, vector normalization, and vector norm.

GLSL, like other shading languages, can accept two different types of input parameters when shading a primitive. Attribute variables are input per-vertex for input geometry, thus, they are only available in a vertex shader. Attribute variables are used for values such as position, color, or texture coordinates. Uniform variables are input per-primitive, and can be accessed from fragment and vertex shaders.

Additionally, GLSL can perform operations in a vertex shader and send the results to a fragment shader as a varying value. These varying values are automatically interpolated from each output vertex over the output primitive.

Since cross-platform flexibility was not a requirement for the software implementation, GLSL was selected because of its ease of integration with OpenGL.

3.4 Mesh Structure

To facilitate this research, a third-party mesh structure was selected. This was driven by two integral requirements: the flexibility of importing a model in OBJ format, and the need to subdivide that model for generation of data used in radiosity calculations. OpenMesh was chosen to fulfill these requirements. Additionally, it can be easily extended using generic programming in C++ [44].

OpenMesh can represent meshes of arbitrary polygons, and provides an STL-like interface for iterating through model geometry stored using the half-edge data structure.

The half-edge data structure is similar to the winged-edge structure used in geometric modeling [45]. Half-edges in the mesh store connectivity using one vertex, one face, the next half-edge, the opposite half-edge, and the previous half-edge, see Figure 10.

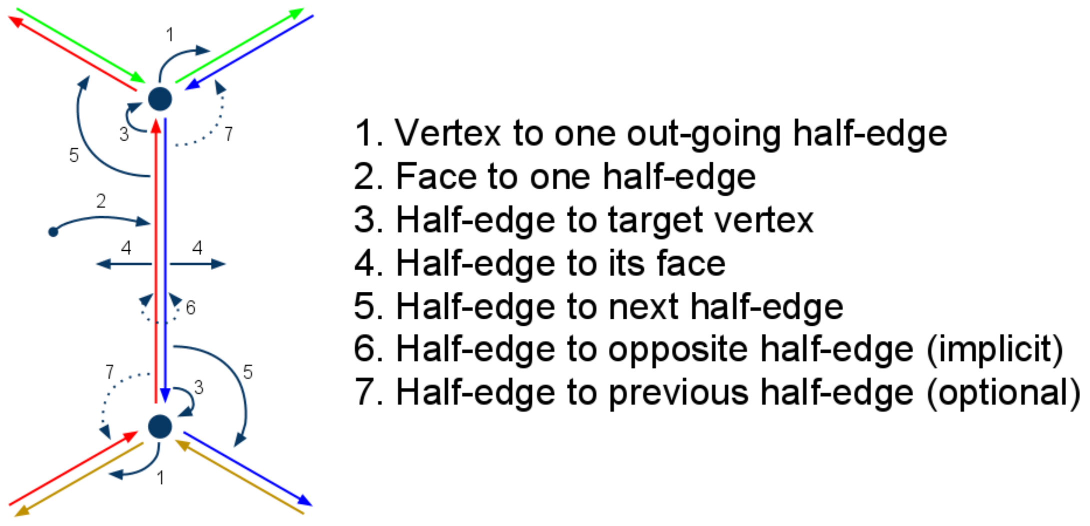

1
3
7
5
2
4
4
6
7
3
5
1
1. Vertex to one out-going half-edge
2. Face to one half-edge
3. Half-edge to target vertex
4. Half-edge to its face
5. Half-edge to next half-edge
6. Half-edge to opposite half-edge (implicit)
7. Half-edge to previous half-edge (optional)

Figure 10: Half-edge data structure [44]

Using this connectivity information, mesh geometry can be traversed quickly, and in a repeatable manner. The extensibility of OpenMesh also enables the association of custom traits with mesh geometry. Traits can include data types and functions, allowing operations on faces, vertices, and edges. For example, the software implementation uses face traits to associate textures with their corresponding surface, represented by a face in the mesh.

OpenMesh provides templates to read and write persistent data, allowing the developer to extend the OpenMesh IO framework to build importers and exporters for the model format of their choice. The OBJ format is ubiquitous, so an OBJ importer was already provided as part of the OpenMesh SDK [46].

The other requirement for this research was the ability to subdivide model geometry, for reasons explained in Section 4.1.4. OpenMesh simplified this task by providing a template subdivision class that operates on triangle meshes. Topological compositing rules are specified as operators that are added to custom subdivision classes defined by the developer. When a triangle mesh is provided to the subdivision class, the rules are applied to the mesh. The software implementation uses rules that subdivide triangle faces 1-to-4. The mesh is re-composited using additional OpenMesh rules.

3.5 Windows API

Although antiquated, MFC was chosen as the development library for application development. The most recent version, MFC 10.0 was shipped by Microsoft with Visual Studio 2010 [47]. MFC continues to be actively developed, with Visual Studio providing the ability to generate dynamic applications from pre-packaged MFC templates.

3.6 Graphics Hardware

Modern GPU-based hardware, like the CPU-based hardware of previous decades, is notoriously transient. Since OpenGL is backwards compatible and GPU applications scale effectively, however, development can proceed on any graphics card that provides the necessary OpenGL extensions.

The experimentation for this research was conducted on a computer using an NVIDIA Quadro 5000. Part of the Quadro series using NVIDIA's Fermi architecture, the Quadro 5000 is built to the specifications shown in Table 1.

Table 1: NVIDIA Quadro 5000 specifications [48]

CUDA Cores	352
Core clock	750 MHz
Memory bandwidth	120 GB/sec
Total frame buffer	2.5 GB
Memory interface	320-bit

Using the OpenGL Extensions Viewer from Realtech VR, the Quadro 5000 was benchmarked using three runs of the render tests for OpenGL versions 1.1 through 2.1, using multiple spinning cubes at v-sync fullscreen [49]. The results are displayed in

Table 2. Each version's benchmark adds additional operations and tests the functionality introduced in that version of OpenGL.

Table 2: NVIDIA Quadro 5000 render benchmark tests

OpenGL version	1.1	1.2	1.3	1.4	1.5	2.0	2.1
Average framerate	170.3	310.7	306	197.3	204.3	192.7	197.7
Standard deviation	1.2	4.9	1.7	3.1	0.6	0.6	0.6

3.7 Development Tools

Developing GPGPU applications using a graphics API such as OpenGL is not a trivial task. OpenGL is an enormous finite state machine whose output function draws geometric primitives. Achieving the correct configuration of the state machine can be difficult without the proper software development practices. At present, GPU programming with shaders is a recent technology relative to CPU programming, and the development tools have not yet matured. In addition to the Microsoft Visual Studio C++ IDE, several other GPU-specific debuggers were used.

The most fundamental GPU debugging tool is the "`printf` technique," referring to the classical use of the `printf` command to output variable values when programming in C. Shaders can be modified to write out an intermediate value from their computational flow to a texture bound to an FBO. When the texture is mapped to a polygon and displayed, the intermediate shader results are visible. This assumes that the OpenGL machine is in a correctly configured state to display texture, which is not always the true. It also does not account for the developer being interested in specific numerical values at texel-level granularity.

A popular tool for OpenGL state debugging is Graphic Remedy's gDEBugger [50]. An academic license for this software was used during research. A lack of integration with the Visual Studio C++ IDE makes it unwieldy for general-purpose use, but it has multiple viewers for texture memory, readouts for OpenGL state machine variables, and the ability to enable or disable OpenGL functionality to help narrow down the scope of a problem. It also has the ability to break when any error in the OpenGL configuration is detected. This is valuable since OpenGL must be polled constantly to see when it enters an error state. At present, gDEBugger is undergoing active development and new features are added frequently.

Although gDEBugger is improving in its ability to help with shader debugging, at present it remains primarily useful for keeping track of the OpenGL finite state machine configuration. A less-developed but powerful tool for GLSL debugging is glslDevil, from the University of Stuttgart [51]. Using this application, the contents of GPU registers can be retrieved, as with a CPU debugger. Graphical displays can show the values of each shader variable for all in-flight fragments, and each branch in shader code can be examined for every output pixel. Vertex, fragment, and geometry shaders (available in extended OpenGL 2.1) are all supported in unique debug output forms.

For now, GPU development remains less-established than CPU development. As the technology continues to gain popularity, the development tools will improve in their sophistication.

4. PROGRESSIVE REFINEMENT AND IMPORTANCE-DRIVEN RADIOSITY

Importance-driven radiosity builds upon progressive refinement radiosity, which had previously been implemented for execution on the GPU. Implementation of both techniques was necessary for purposes of comparison with perspective-driven radiosity.

4.1 Progressive Refinement Radiosity

The software implementation of progressive refinement on the GPU follows the original implementation details, with minor changes [4]. Some features were not implemented, as they had been previously solved and were deemed unnecessary for demonstration of the perspective-driven algorithm.

4.1.1 Texture Creation

Each surface in the environment has associated textures that are the same resolution as the surface's polygon, used for radiosity computation and display. There are two textures per surface: one for the total radiosity of surface j, B_j, and one for the unshot radiosity of the surface, ΔB_j. Textures are created as `GL_RGB` format and `GL_RGB16F` internal format, meaning they have three-color channels with 16-bit floating-point precision per channel. During display of the environment model at runtime, the B_j texture is mapped to the polygon drawn for surface j, resulting in the display of total radiosity for that surface.

4.1.2 Emissive Energy

Before a surface is chosen to shoot radiosity into the environment, all surfaces must have their total radiosity and unshot radiosity textures initialized. As shown in Section 2.1.2, the initial radiosity of a surface in the scene is the emissive energy of the surface. For light surfaces, this emissive energy is the light intensity. For non-light surfaces, the emissive energy is zero.

In the software implementation, light surfaces are specified using the ambient material property of the input OBJ model, and reflectivity is specified using the diffuse material property. As textures are created for surfaces, any surface j which has emissive energy will have its B_j and ΔB_j textures attached to an FBO and initialized to the emissive value using a `GL_COLOR_BUFFER_BIT` clear operation. Both textures are initialized at the same time using MRT. If a texture does not have an associated emissive value, its B_j and ΔB_j textures are attached to the FBO and cleared to black, which amounts to zero in every RGB channel. Texture initialization is illustrated in Figure 11.

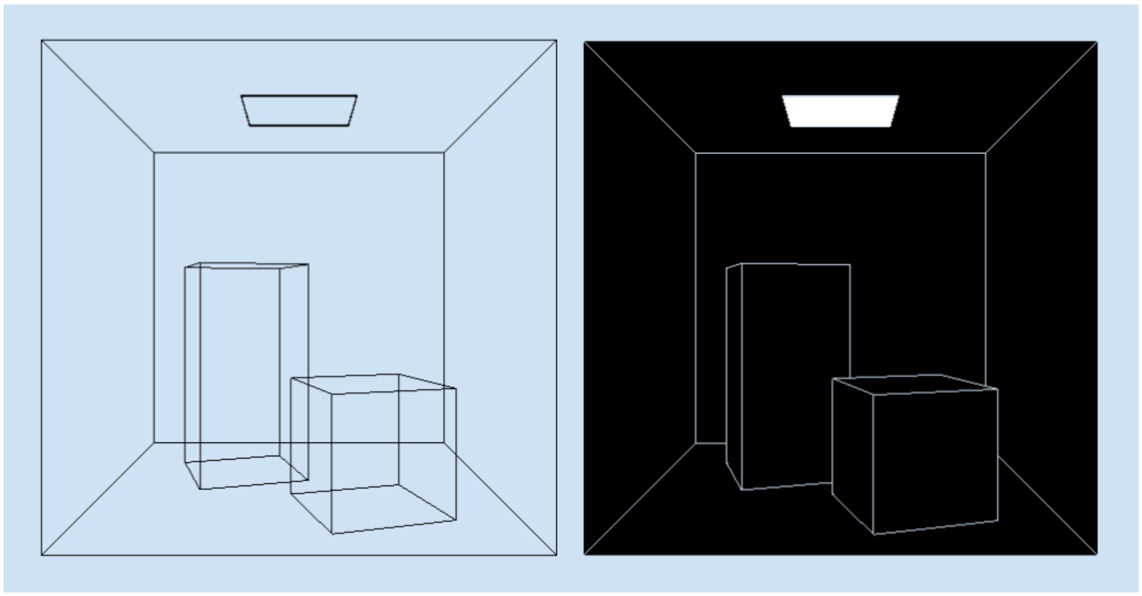


Figure 11: Wireframe compared to emissive texture-mapped environment model

4.1.3 Next Shooting Surface Selection

For radiosity to be propagated through the environment, a surface must be chosen to shoot its radiosity. As stated in Section 2.1.2, the optimal shooting surface i is the one with the largest unshot radiosity energy, $\Delta B_i A_i$. In progressive refinement radiosity on the GPU, a shooting surface is chosen by comparing these unshot radiosity energy values without transferring graphics memory to the CPU. The unshot radiosity of the surface is obtained by reading the highest-level mipmap of the surface's ΔB_i texture.

The top-level mipmap is result of the minification of the entire texture down to one texel. Thus, the unshot radiosity of the surface is averaged to a single RGB vector. This RGB vector can be converted to HSI space to find the intensity of the color, which is multiplied by the area of the surface to find the unshot radiosity energy, seen in Figure 12.

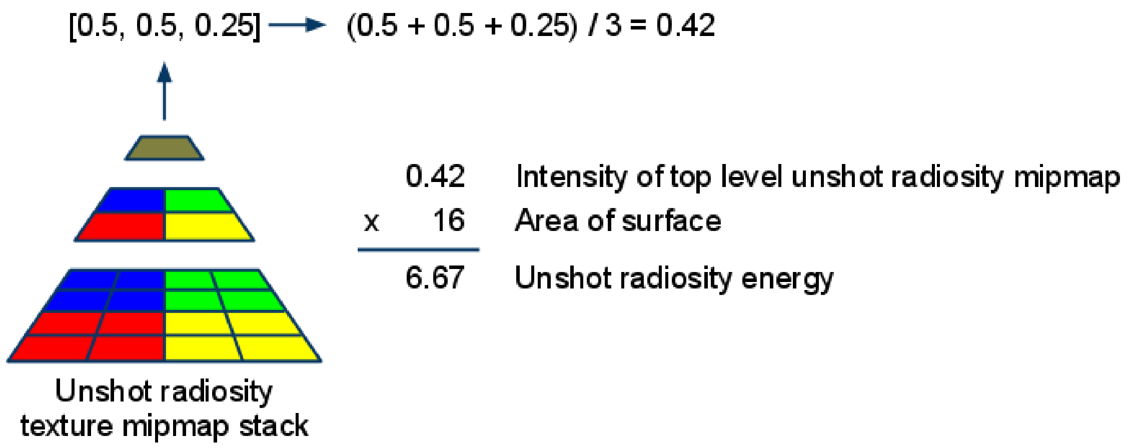


Figure 12: Computation of the unshot radiosity energy of a surface

To accomplish this comparison of energy using the GPU, a one-pixel orthographic projection is set on a draw buffer. For all N surfaces in the environment, a single pixel whose value is the integer ID of surface i is written to the draw buffer. A shader is enabled which uses the inverse of the unshot radiosity energy of surface i to set the depth value of the shaded fragment. The ID of the surface with the highest unshot energy will be drawn to the buffer as in Figure 13. The ID in the buffer can be read back using a PBO. A far plane can be set on the orthographic projection to control when convergence is reached: once every plane has a minimum unshot radiosity, all surface IDs will have a depth greater than the far plane, so no ID value will be drawn to the FBO.

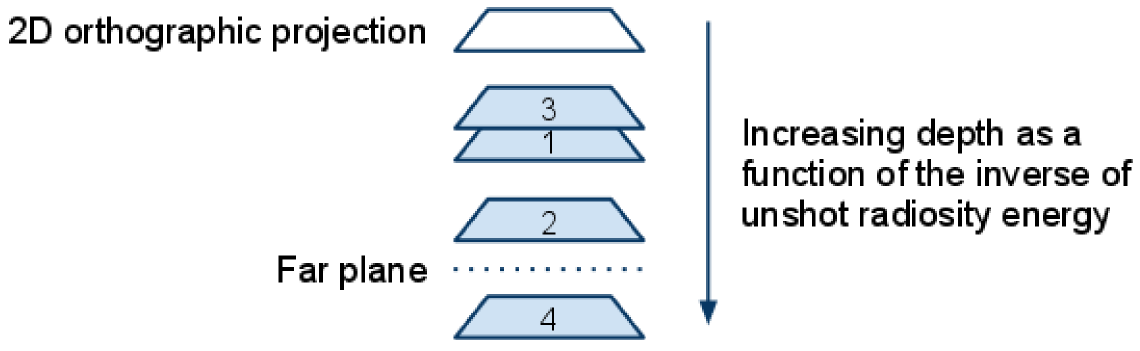

2D orthographic projection
3
1
2
Far plane
4
Increasing depth as a function of the inverse of unshot radiosity energy

Figure 13: Selection of shooting surface via depth test where surface 3 has the highest energy

When a surface is selected for shooting, radiosity is shot from points on the surface that correspond to the centers of texels for a mipmap in the unshot radiosity texture. Shooting from mipmap level $n + 1$ averages every four texels in mipmap level n to a single texel, and reduces the number of shooting locations by a power of two.

Shooting positions must be kept on the CPU to set the matrices used to project the geometry from the eyepoint of the center of the shooting surface, where the surface normal is the look-at direction. In the software implementation, the CPU was assisted by utilizing the GPU to interpolate vector values over a 2D polygon. By setting an orthographic projection to the size of the shooting surface's level n mipmap, a polygon can be drawn using the shooting surface's vertices as color values for each vertex in the drawn polygon. The result of the draw operation can be read back from the draw buffer using a PBO. The values in the returned buffer will be the positions of the shooting surface's shooting locations. The same technique can find the shooting texture coordinates which index the ΔB_j texture.

4.1.4 Hemisphere Projection

Computing the form factor between two surfaces in a complex environment involves finding the visibility of the receiving surface's elements from the shooting location. The earliest hardware-accelerated means of determining visibility in an environment containing hidden surfaces was achieved by projecting the environment geometry onto a hemicube surrounding the shooting location, shown in Figure 14 [7].

This technique requires five passes over the environment geometry: one for each of the hemicube planes. The values of pixels in the hemicube are compared to the IDs of surfaces in the environment. Each pixel in the hemicube represents a known delta form factor. By summation of the R pixels of a surface that pass the hemicube visibility test, the total form factor from the shooting surface to the receiving surface may be determined.

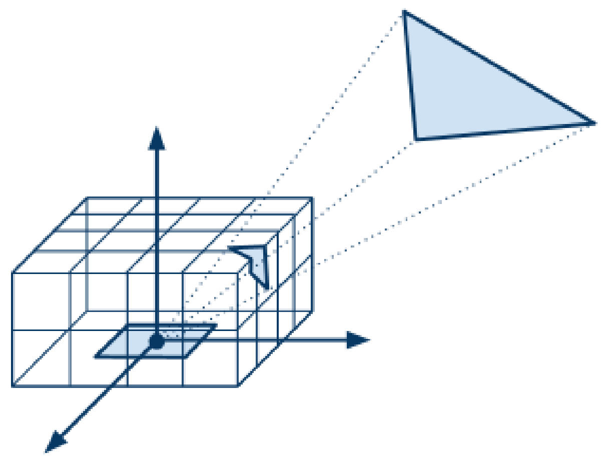


Figure 14: Projection of environment geometry to a hemicube

Progressive refinement radiosity on the GPU uses a hemisphere projection, made possible by a stereographic vertex shader. A single render pass over the environment geometry is made with the stereographic shader enabled, using the ID of surface j as the color. Once the item buffer has been generated, each surface j for all N surfaces except the shooting surface is drawn using an orthographic projection of the surface with a one-to-one mapping to its B_j and ΔB_j textures. During the draw operation, a fragment shader is enabled which back-projects every element of surface j into the hemisphere item buffer generated from the shooting location. If the ID in the item buffer matches the ID of the surface, it is considered visible from the shooting location.

Two hemisphere item buffers generated from the environment’s light source are shown in Figure 15. Geometry used to generate the item buffer on the right has been subdivided twice, and contains four times as many polygons. Surfaces in the left figure appear blocky and misshapen. This demonstrates the observed disadvantage of using a vertex shader to produce the stereographic projection: in a true stereographic projection, lines project as curves, but GPU rasterization produces straight edges [4]. These straight edges can produce rasterized images where surfaces are occluded in unexpected ways, resulting in false positives and negatives for fragment visibility tests. If surfaces are not sufficiently subdivided, they will not produce an accurate representation of the environment from the shooting location’s perspective.

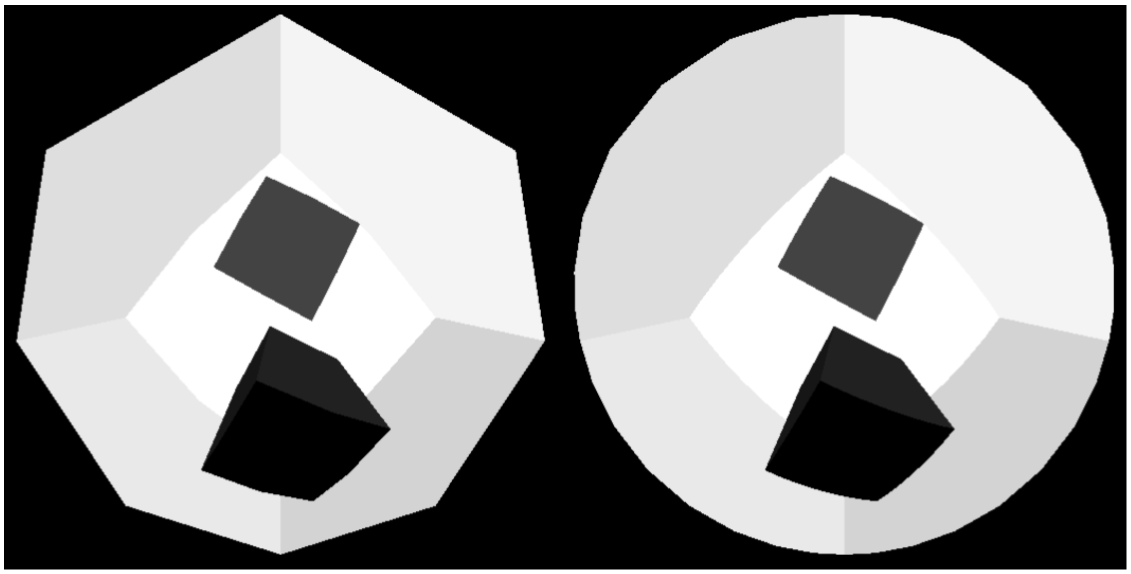


Figure 15: Two stereographic hemisphere item buffers, where the right mesh has been subdivided twice

Based on implementation specifics described in the original GPU progressive refinement radiosity algorithm, the software implementation handles this problem by representing the environment with two separate meshes. The input model retains its original level of subdivision, and is used for displaying radiosity textures. Derived from this model is an “item buffer mesh,” shown in Figure 16. The mesh is subdivided to a level specified by the user and used for drawing the hemisphere item buffer. The higher the level of subdivision, the greater the accuracy of the stereographic projection will be.

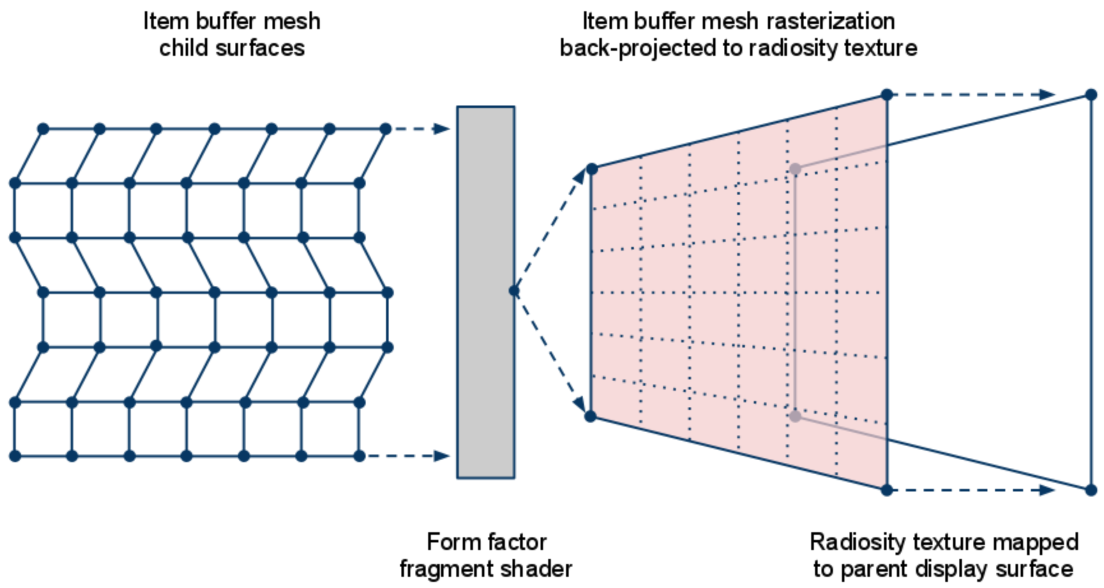

Item buffer mesh child surfaces
Item buffer mesh rasterization back-projected to radiosity texture
Form factor fragment shader
Radiosity texture mapped to parent display surface

Figure 16: Item buffer mesh rasterization mapped to radiosity texture during form factor shading

The surfaces in the item buffer mesh exhibit a child relationship to the parent surface from which they were derived. The parent mesh retains its association with the textures used for radiosity calculations. When a surface is shaded during the form factor computation stage, however, the vertices and texture coordinates of the polygons in the item buffer mesh are used as input geometry for back-projection into the hemisphere item buffer.

4.1.5 Form Factors

Form factor calculation is the most computationally intense step in progressive refinement. When the algorithm is adapted for the GPU, form factor computation for each surface element is performed in parallel, where a surface element in the environment is represented by a texel in the radiosity textures associated with surfaces.

The standard form factor equation assumes that “the areas of the differential elements are small compared to the distance between them” [4]. This means that each surface element must be the same relative size. Rather than make this assumption for surface elements represented by statically sized textures, progressive refinement radiosity on the GPU uses the disc approximation to the finite-area-to-differential-area form factor

$$F_{ij} = A_i \sum_{i=1}^{m} \frac{cos\varphi_i cos\varphi_j}{\pi r^2 + \frac{Ai}{m}}$$

where surface i has been subdivided into m oriented discs and all other term definitions remain the same as previously defined [9]. When surface i is chosen to shoot the unshot radiosity in its ΔB_i texture to surface j, surface j accumulates radiosity to all visible surface elements in its B_j and ΔB_j textures, where a surface element's visibility is determined using the technique explained in Section 4.1.4. After surface i shoots all of its radiosity from the ΔB_i texture, all texels in the ΔB_i texture are reset to zero.

Occlusion queries are generated for each polygon in the item buffer mesh during the stereographic draw operation. These occlusion queries represent the number of pixels that passed the rasterization step when rendering a polygon. By traversing the results of the occlusion queries on its children, a parent surface can determine if it was visible from the shooting location. As soon as a child surface query has a non-zero result, traversal stops. If no child surface occlusion queries pass, form factor shading on the parent surface textures can be avoided.

4.1.6 Ambient Term

During research, a contribution to the original GPU progressive refinement algorithm was made through the addition of an ambient term, which is used for display only and is not added to a surface's unshot radiosity. Similar to the determination of the optimal shooting surface, computation of the ambient term takes place on the GPU.

An ambient term is used to compensate for early iterations in the progressive refinement radiosity algorithm when "global illumination is not yet accurately represented" [3]. Without knowing the visibility between surfaces in the environment, a form factor from all N surfaces in the environment to surface j can be approximated as

$$F_{*j} = \frac{A_j}{\sum_{i=1}^{N} A_i}$$

and an average reflectivity for the environment can be represented as

$$\rho_{avg} = \frac{\sum_{i=1}^{N} \rho_i A_i}{\sum_{i=1}^{N} A_i}$$

where a unit of energy that is sent into the environment can be expected to reflect, on average, ρ_{avg}. If this energy is reflected by ρ_{avg} continuously against other surfaces, a total interreflection factor $\mathcal{R}$ is computed using the geometric sum such that

$$\mathcal{R} = 1 + \rho_{avg} + \rho_{avg}^2 + \rho_{avg}^3 + \cdots = \frac{1}{1 - \rho_{avg}}$$

If $\mathcal{S}$ is the total area-averaged unshot radiosity in the environment such that

$$\mathcal{S} = \sum_{i=1}^{N} (\Delta B_i F_{*i})$$

for all N surfaces, an ambient term $\mathcal{A}$ can be computed as

$$\mathcal{A} = \mathcal{R}\mathcal{S}$$

and a surface j can be displayed with a better estimation of its radiosity, B_j', where

$$B_j' = B_j + \rho_j \mathcal{A}$$

In the software implementation, the sum of areas of all N surfaces, $\sum_{i=1}^{N} A_i$, and the total interreflection factor, $\mathcal{R}$, can be computed when an input model is given. The area sum is given as input to a shader that will compute the ambient term, and the interreflection factor is given as input to a shader that will display the mesh with the ambient term added to the surface's radiosity textures.

Once the radiosity for each surface has been initialized, each surface j that has unshot radiosity is input to the ambient term computation shader. In the shader, the fractional form factor of the surface, F_{*j}, is computed by dividing the surface's area by the environment area sum. Next, the top-level mipmap of the ΔB_j texture is read and multiplied by F_{*j} to obtain $\Delta B_i F_{*i}$. The result of this operation is drawn to a one-pixel texture. Multiple render passes are performed on the texture, one for each surface that has unshot radiosity. The value in the texture is incremented by the result of each render pass, resulting in the computation of $\mathcal{S}$.

At display time, when the mesh is rendered with the B_j texture mapped to surface j for all N mesh polygons, a fragment shader is utilized to add the ambient term to each texel in the B_j texture. The one-pixel $\mathcal{S}$ texture is given as an input to the display fragment shader. This fragment shader multiplies the area averaged unshot radiosity in the environment, $\mathcal{S}$, the interreflection factor, $\mathcal{R}$, and the reflectivity of the surface, ρ_j, which is added to the texel values in the B_j texture.

When a surface finishes contributing its unshot radiosity to all other surfaces in the environment, its B_j texture is cleared, and $\mathcal{S}$ is re-computed with the updated amount of unshot radiosity in the environment. As unshot radiosity is distributed through the environment during successive iterations, the ambient term becomes less noticeable, as seen in Figure 17.

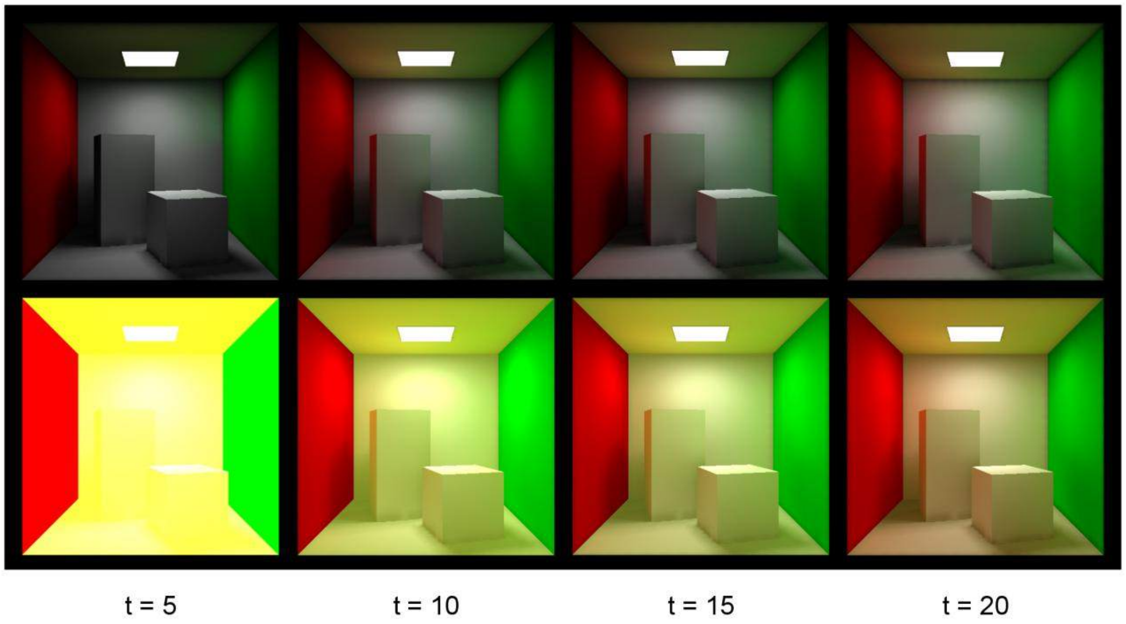


Figure 17: Progressive refinement without (top) and with (bottom) use of an ambient term, at t iterations

4.1.7 Adaptive Subdivision

In the original implementation of progressive refinement radiosity on the GPU, adaptive subdivision was added to improve the quality of generated radiosity images [4]. Surfaces are represented using quad trees, where the leaves of the tree contain radiosity textures. By evaluating the gradient of radiosity lighting across textures using a fragment shader, a surface can be subdivided or collapsed. This leads to more detail for high frequency image data and less detail for areas of lower frequency data. Therefore, compute time is focused on areas that will contribute more high frequency details to the convergence of an image.

Mesh subdivision for the purpose of efficiently focused detail and computation is a familiar subject in the study of radiosity [12-13, 52]. The original purpose of importance-driven radiosity was to direct mesh subdivision to areas where higher detail could be of most value to the image generated from the user's view [5].

This technique has already been demonstrated to both run efficiently on the GPU and produce higher quality imagery for an interactive solution, demonstrated in the two images show in Figure 18. It was deemed unnecessary for the demonstration of the perspective-driven algorithm, and has been omitted from the software implementation.

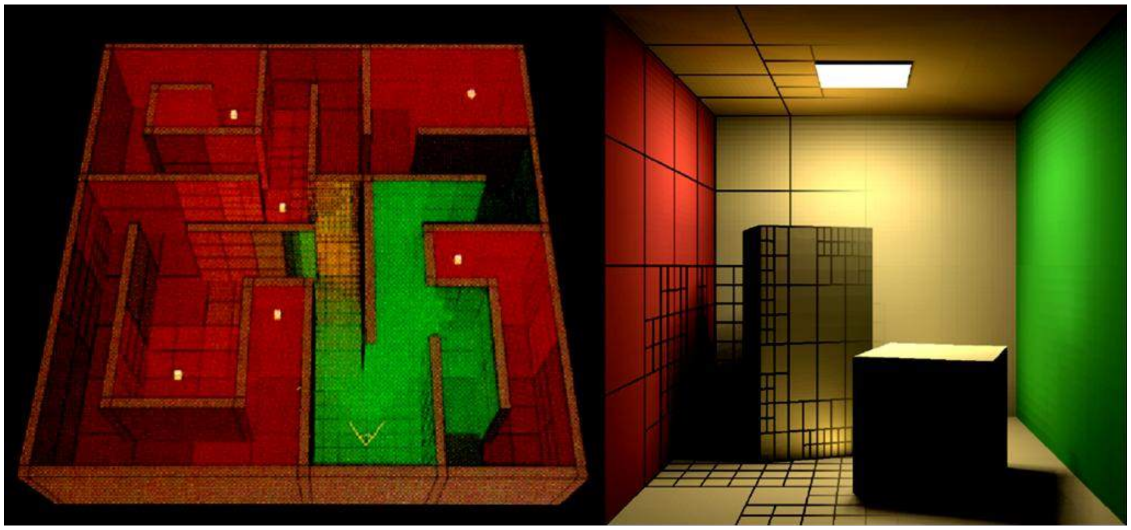


Figure 18: Adaptive subdivision in importance-driven radiosity (left) and GPU radiosity (right) [4-5]

4.1.8 Interactivity

Since the radiosity calculations are performed on the GPU, most of the processing time is spent calculating radiosity lighting, rather than displaying the results. Consequently, an update of the image is unnecessary for computation to proceed. Updating the image is an encumbrance to solution convergence, since GPU cycles are redirected for non-computational purposes. Disregarding updates to the resulting radiosity image, however, can produce undesired results if the framerate drops below eight frames per second. A framerate of eight frames per second is commonly accepted as the limit of an “interactive” display.

Surfaces can vary greatly in size, and therefore the number of shooting locations per surface can vary as well. The naïve approach to displaying updates to the image is to wait for a surface to finish shooting radiosity to all other surfaces for each of its shooting locations. Once the surface has finished and the radiosity texture is cleared, the display can be redrawn with the radiosity textures mapped to mesh polygons.

During research, it was determined that a high level of interactivity is necessary for the perceived performance of a GPU-based radiosity solution, regardless of the overall increase in time to convergence. A timer-enforced update to the display was added to guarantee that the user would be able to see the convergence of their image at expected intervals. Each time a single shooting location finishes contributing its unshot radiosity to the other surfaces in the environment, the time since the last frame is checked. If there is still time left in the current frame, the next shooting location is allowed to contribute its unshot radiosity. If the frame timer has expired, the shooting state is saved, and the computation loop exits to update the image.

The user can select the framerate at which they wish to see their image refreshed. A higher framerate means more time will be devoted to updating the image, and the time to convergence will be increased. A lower framerate means convergence can proceed with fewer interruptions, but at the cost of a reduced level of interactivity.

4.2 Importance-Driven Radiosity

Importance-driven radiosity was the first extension to progressive refinement radiosity on the GPU that was implemented during research. Perspective-driven radiosity seeks to limit radiosity calculations to an area specified by the user, but importance-driven radiosity ensures that the surfaces that contribute the most light for the area-of-interest will be given priority when selecting shooting surfaces. Therefore, the addition of visual importance to the progressive refinement algorithm is necessary for a perspective-driven solution to produce satisfactory results.

4.2.1 View Dependence

Radiosity is a view-independent global illumination algorithm [1]. Some illumination algorithms, such as ray-casting and ray-tracing, generate an image based on the location of the viewing plane in the scene [53-54]. Since the viewing plane drives the image computation, the image must be re-rendered whenever the viewing plane changes. Once illumination of an environment has been computed by radiosity, the solution need not be re-computed unless the environment changes.

Importance-driven radiosity was developed to address shortcomings in the imprecision of radiosity view-independence. Because of its view-independence, a radiosity solution is often rendered with uniform priority given to each surface. Surfaces that are hidden or are not within the view of the user are given the same processing precedence as a surface directly in front of the user. Conversely, since all surfaces are processed homogeneously, a surface closer to the user is paid as much attention as a hidden surface. Surfaces such as these should be given higher priority since they directly affect convergence of the radiosity solution from the user's view.

The idea of visual importance in a radiosity solution was introduced as a means of driving the precision of a radiosity solution [5]. In this importance-driven radiosity solution, surfaces that are visually important are given a higher degree of precision in the solution by subdividing the mesh used for radiosity calculations at areas where it benefits the image generated from the user's viewpoint. This visual importance vector is initially directed by the user's view and propagated throughout the environment using the same mechanism as radiosity, seen in Figure 2. By introducing a view-dependent weighting metric to the radiosity solution, the image can be more accurate for the user's view of the environment and retain the interactive features of a view-independent illumination algorithm.

The importance-driven radiosity algorithm used in this research is based on the idea that in progressive refinement radiosity, visual importance can be used to select a shooting surface which contributes the most radiosity to other surfaces and proceeds towards the convergence of the image the user is interested in [2]. Surfaces that receive importance directly from the user's viewpoint or through propagation of importance have a higher visual weight and are more likely to be selected.

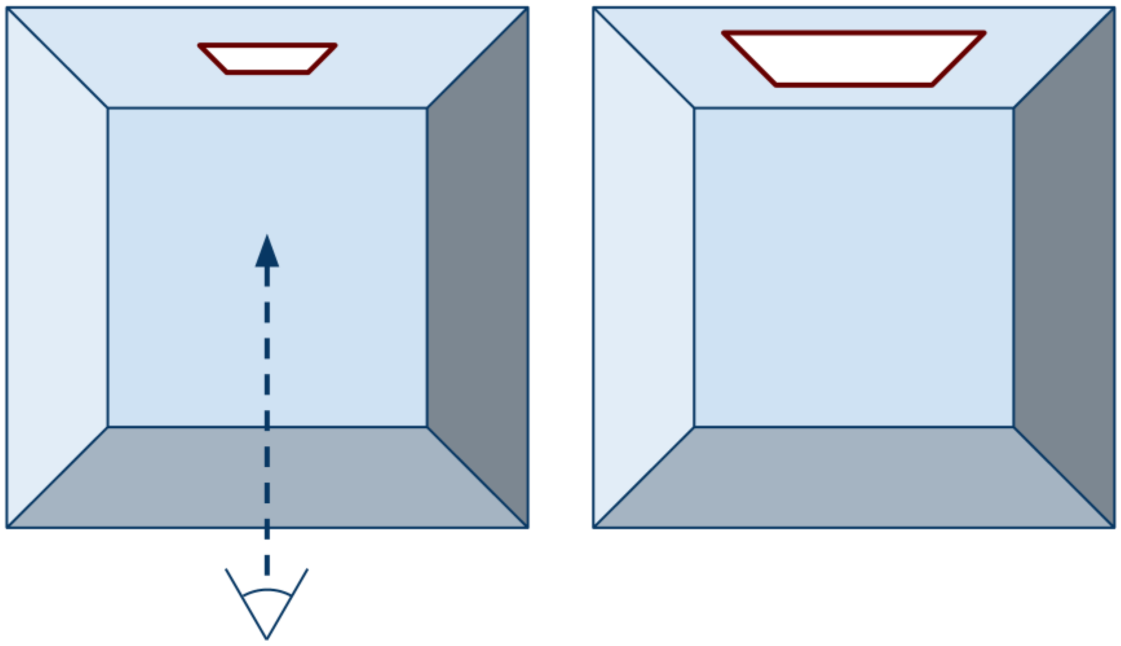


Figure 19: Selection of a shooting surface weighted by radiosity energy and visual importance

For example, in Figure 19, assuming the intensities of both light sources are equal, the light source on the right will have a higher amount of radiosity energy because of the larger surface area. In progressive refinement radiosity, this surface is chosen for shooting unshot radiosity into the environment. In importance-driven radiosity, the light source on the left has a higher visual importance weight since the user's viewpoint gives it directly received importance. Hence, when considering both the radiosity energy and the visual importance, the surface on the left should be chosen to shoot its unshot radiosity.

4.2.2 Texture Creation

In the software implementation, progressive refinement radiosity on the GPU was supplemented with visual importance to create an importance-driven radiosity solution. For all N surfaces in the environment, an additional two textures for surface i are required: I_i, which represents all received importance, and ΔI_i, which represents all received importance that has not yet been contributed to every other surface. These textures are the importance-driven analogies to the B_i and ΔB_i textures used in progressive refinement radiosity, and are created using an intensity format. The I_i and ΔI_i textures are never displayed as part of the radiosity image and are therefore stored at half the resolution of radiosity textures.

4.2.3 Directly-Received Importance

In the equation for the radiosity of a surface i,

$$B_i = E_i + \rho_i \sum_{j=1}^{N} B_j F_{ij}$$

directly received radiosity is represented by the term E_i. In the equation for importance of a surface i

$$I_i = R_i + \rho_i \sum_{j=1}^{N} I_j F_{ij}$$

there is a term analogous to E_i: R_i, which represents directly received importance. The directly received importance value of a surface in the environment will be determined by its visible area when projected in the user's view. The sole emitter for this direct importance is the user's view, defined as a surface of infinitesimal area in the environment [5]. To accomplish this contribution of directly received importance, a technique similar to back-projection into the stereographic hemisphere item buffer was utilized.

At run time, importance textures are initialized to zero. Once the environment has been initialized, an item buffer is rendered from the user's view of all surfaces in the environment. Unlike the stereographic projection used for the hemisphere item buffer, this user view item buffer exactly matches the projection the user sees on screen, as demonstrated in Figure 20. Once again, the surface ID is used as color.

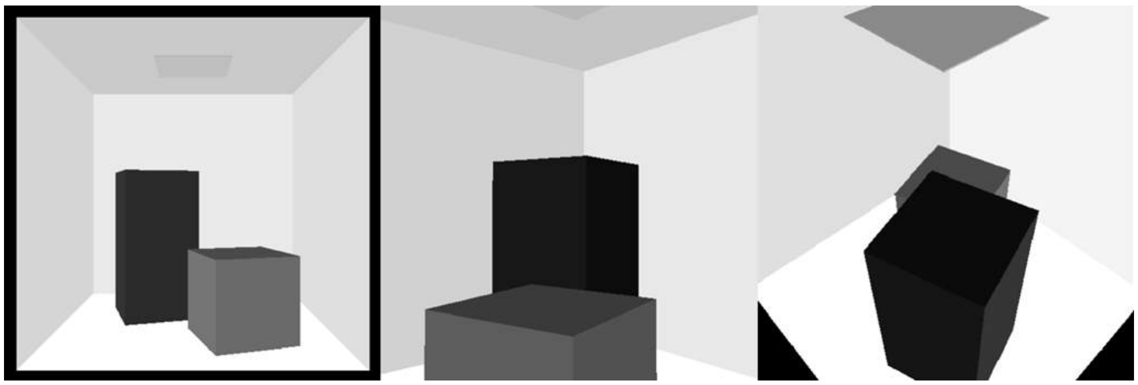


Figure 20: User view item buffer with surface ID as color, from three perspectives

Occlusion queries are initiated for each polygon rendered to the user view item buffer, where rendered polygons correspond directly to all N surfaces in the environment. The surface i whose occlusion query returns a non-zero pixel count has its I_i and ΔI_i textures bound to an FBO with a one-to-one orthographic projection. A fragment shader is enabled which back-projects each fragment in the surface polygon to the user view item buffer. The ID value in the surface polygon is compared to the item buffer for each fragment in the polygon. If the two values match, the surface element represented by the texel is considered visible from the user's view, so the I_i and ΔI_i textures receive direct importance from the user's view. This direct importance is simply an intensity value. To receive the importance value, the form factor is computed between the receiving surface element and the user's view, assuming that the camera has an area of zero. Once the form factor is computed, the importance textures can be shaded just as the radiosity textures are shaded during progressive refinement. The results of this shading operation is shown in Figure 21.

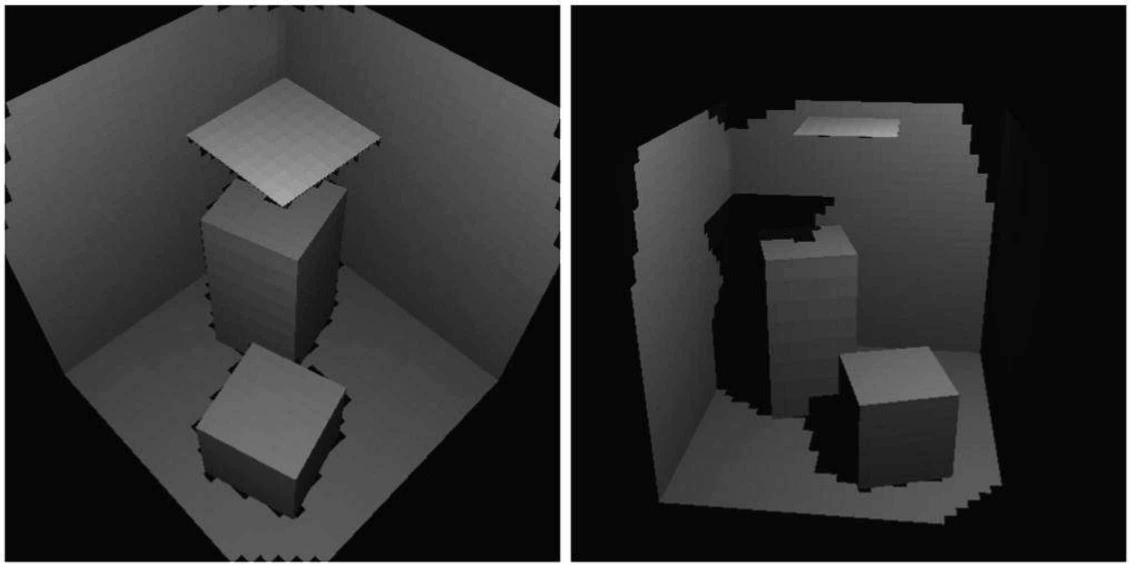


Figure 21: Visualization of directly received importance, shown from the user's view and a rotated view

4.2.4 Next Important Shooting Surface Selection

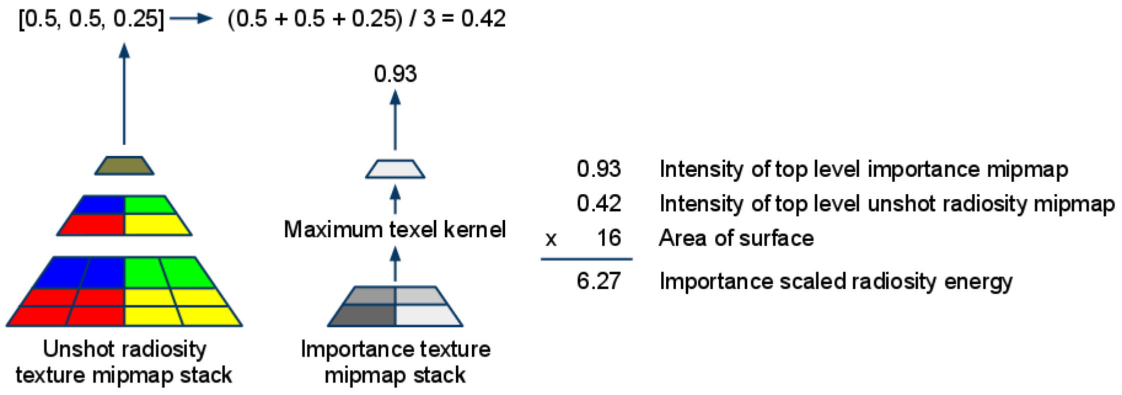
The software implementation uses both visual importance and radiosity energy to select an optimal shooting surface. This "important" shooting surface can be selected in a manner similar to the selection of the optimal shooting surface in progressive refinement radiosity.

To find the average importance of a surface, a naive approach would be to read the top-level mipmap of the importance texture, similar to the process of finding the average radiosity of a surface. However, importance is view-dependent, and the directly-received importance of a surface can be occluded by other surfaces in the scene. This can lead to error when considering a surface X very close to the user's POV, which has been partially occluded by another surface Y directly between the surface and the POV. Surface X may be shaded with a very high importance in several texels, but when the top-level mipmap of the texture is read back, the texels with high importance will be averaged over the entire texture. Therefore, even though surface X has areas of high importance, it may not be selected for shooting due to the low average importance. A more effective approach is the use of a maximum kernel, applied to the texture in place of the linear interpolation kernel which is applied automatically when mipmps are generated by the GPU.

The “maxmap” can be generated by performing a reduction with a fragment shader. An importance texture’s level-0 mipmap is rendered to its level-1 mipmap using the maxmap shader. The four texels which correspond to each rendered fragment’s texture coordinate are read, and the maximum of the four is selected and written to the level-1 mipmap. The process repeats for the higher mipmap levels until a single texel is generated, which contains the maximum importance value for all texels in the texture.

The shader used to select the next shooting surface for progressive refinement radiosity takes the ΔB_i texture of surface i. The top-level mipmap of the texture is read and used as the depth value of the fragment whose value contains the integer ID of the surface. An additional input texture is added for importance-driven radiosity, which accepts the I_i texture. The top-level mipmap of this texture is read, which contains the maximum importance value for the surface. To find the optimal shooting surface, the shader writes the ID value fragment to the one-pixel buffer using the inverse of $I_i \Delta B_i A_i$as the depth value of the fragment. An example is show in Figure 22.

The ID of the surface with the maximum unshot radiosity energy scaled by importance will be at the top of the buffer when the buffer contents are read back to the CPU using a PBO. Once the shooting surface has been selected, radiosity is shot from the surface as described in Section 4.1.3.


[0.5, 0.5, 0.25]
(0.5 + 0.5 + 0.25) / 3 = 0.42
0.93
Maximum texel kernel
Unshot radiosity texture mipmap stack
Importance texture mipmap stack
0.93 Intensity of top level importance mipmap
0.42 Intensity of top level unshot radiosity mipmap
x 16 Area of surface
6.27 Importance scaled radiosity energy
Figure 22: Computation of importance scaled unshot radiosity energy of a surface

4.2.5 Propagation of Importance

Importance can be propagated to surfaces besides those that receive importance directly from the user's view, similar to the way light in an environment emitted from a light source will be transported to surfaces throughout the environment. Therefore, it is necessary to transport importance in the same manner as radiosity.

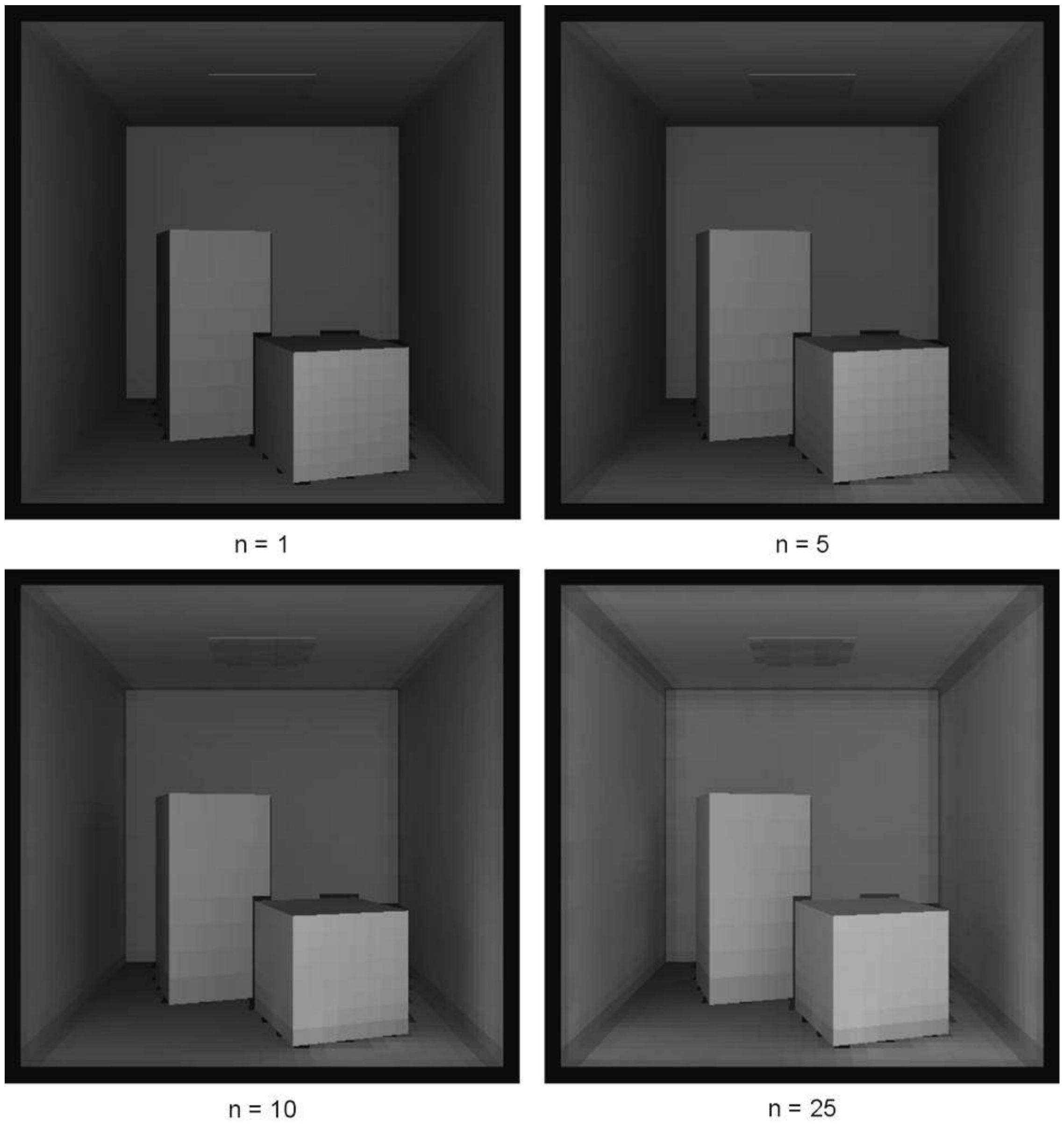


Figure 23: Importance textures after n importance-driven radiosity iterations

Directly received importance is contributed to all surfaces visible from the user's view, and is collected in the I_i and ΔI_i textures of surface i. The surface in the environment with the highest unshot importance, ΔI_i, is found using the same shader which selects the surface i with the highest unshot radiosity, ΔB_i. Once this surface is found, importance is shot from the surface in the same way radiosity is shot from a surface. The result of importance propagation is show in Figure 23. Since importance textures are stored at a lower resolution, the number of importance shooting locations for a surface will be half the number of radiosity shooting locations for the same surface.

In the software implementation, the same code is used for radiosity textures and importance textures when generating the stereographic item buffer for a shooting location, back-projection of each visible surface element in the environment to the shooting location, and computing the form factor from the shooting location to each receiving surface element. During a single iteration of importance-driven radiosity, importance propagation begins when the surface with the highest unshot importance is selected. This is followed by radiosity propagation using the propagated importance as the weight for the optimal shooting surface.

4.2.6 View Changes

Light surfaces in the environment are the static sources of directly received radiosity; the user's view, which is the source of directly received importance, is dynamic. Each time the user's view changes, importance must be re-shot into the environment, potentially affecting areas that have already received direct importance. Therefore, changes in the user's view must be anticipated in the importance-driven radiosity solution [2].

To account for a change in the user's view, the previous amount of directly received importance must be known. If the previous directly received importance for a surface i is R_i and the newly computed directly received importance is R_i', the change in directly received importance, $\mathit{\Delta} R_i$, is such that

$$\Delta R_i = R_i' - R_i$$

Therefore, the importance values for the surface are adjusted by the addition of ΔR_i

$$\Delta I_i = \Delta I_i + \Delta R_i$$
$$I_i = I_i + \Delta R_i$$

The newly computed directly received importance, R_i', is stored as R_i, in anticipation of the next change in the user's view.

To account for changes in the user's view in the software implementation, an additional texture, R_i, is added for surface i. The R_i texture is given as an input to the shader which contributes directly received importance, along with the I_i and ΔI_i textures. The newly contributed directly received importance, R_i' is computed by finding the form factor between the user's view and all surface elements determined to be visible during back-projection into the user view item buffer. The one-to-one mapping of texels to fragments allows us to subtract the R_i texel value from each fragment's R_i' texel value. The values for I_i, and ΔI_i are written out to the textures attached to an FBO, and R_i' is written out to the R_i texture, which is used the next time the user's view changes. All textures are written simultaneously with MRT.

5. PERSPECTIVE-DRIVEN RADIOSITY

Progressive refinement radiosity presents an iterative approach to the radiosity solution. Importance-driven radiosity maximizes the visible work done per-iteration by considering the user's view in the environment. Perspective-driven radiosity seeks to increase the amount of useful work towards the user's perception of image convergence by limiting radiosity calculations to the user's view. The most computation time in radiosity is spent calculating the form factors between two surfaces in an environment. By performing these calculations only for surfaces in the user's view, the amount of visually important work per iteration can be increased.

5.1 View Dependence

Importance-driven radiosity adds the concept of view-dependence to the radiosity solution by directing refinement based on the user's view. Some of the view-independent elegance of progressive refinement is lost, but in return, the user gains an image that converges more quickly for the view they are interested in. With perspective-driven radiosity, the solution becomes further view-dependent, since image generation is localized to the area of the environment that their gaze encompasses.

Selection of the shooting surface in perspective-driven radiosity is the same as importance-driven radiosity; therefore, the importance propagation will not be limited by the user's view and proceeds as it does in the importance-driven algorithm. Even though a surface falls outside the user's view, it can be selected for shooting if it has unshot radiosity. This ensures that any directly received radiosity will be contributed if it can affect the user's view of the lit environment. A surface outside the user's view, however, will not receive radiosity because of the strict view-dependence of the algorithm. Therefore, it cannot be guaranteed that a surface in the user's view will receive secondary radiosity from a surface that has not yet received direct radiosity. This view-dependence requires some consideration to ensure convergence.

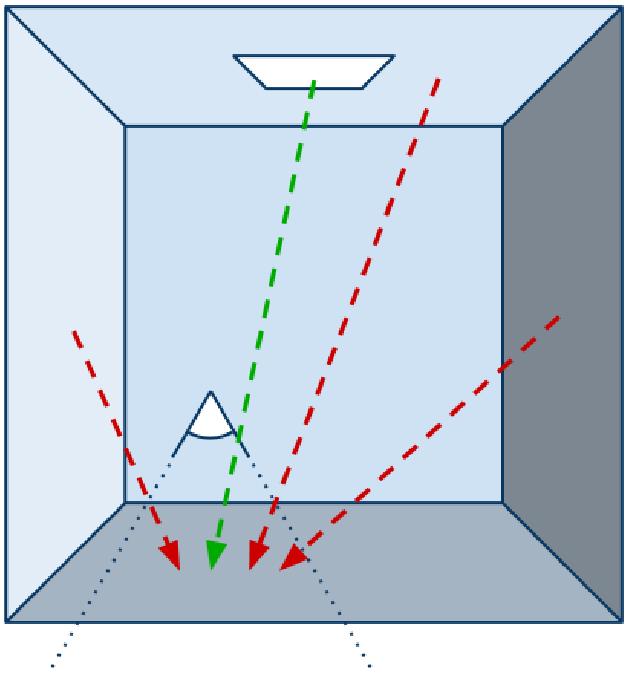


Figure 24: Perspective-driven view-dependence

In the environment in Figure 24, the user's view encompasses only the bottom surface. Because there is a light source in the environment which has emissivity, this surface will be selected for shooting. The bottom surface of the environment, which is in the user's view, will receive radiosity directly from the chosen shooting surface. Since all other surfaces in the environment are outside the user's view, no other surfaces will receive radiosity directly from the light surface. Consequently, the bottom surface will not receive secondary radiosity from any other surface in the environment.

A relaxation in the strict view-dependence of perspective-driven radiosity allows the solution to proceed to convergence. Over time, the FOV angle used to generate the user's view of the environment remains constant, but a "simulated FOV angle" will grow to include other surfaces once the area-of-interest has received a pre-determined amount of processing. The user does not detect this change, since the FOV angle of their view remains the same.

A user-defined decay interval can be applied to grow the simulated angle of the FOV. As the FOV angle grows, more surfaces are included in the area-of-interest. These surfaces receive direct radiosity, and in turn shoot their received radiosity to the bottom surface. When the user changes their view, it is assumed that a new area-of-interest has been selected, so the simulated FOV angle will be reset to match the actual FOV angle until the decay interval begins to grow the simulated FOV angle once again.

5.2 Stored Radiosity Table Creation

In progressive refinement radiosity, a surface i is said to contribute radiosity, B_i, to a surface j for all N surfaces, such that

$$\forall j \in N: B_j = B_j + \rho_j B_i F_{ij} A_i / A_j$$

To adjust this for perspective-driven radiosity, we instead define

$$\forall j \in V: B_j = B_j + \rho_j B_i F_{ij} A_i / A_j$$
$$\forall j \notin V: \mathbb{B}_{ij} = \mathbb{B}_{ij} + B_i$$

where $\mathbb{B}_{ij}$ represents the stored radiosity which surface i should contribute to surface j, and V represents the user's view. A surface is visible if any surface element is visible from the user's view. Because $\mathbb{B}_{ij}$ is indexed by two variables, it can be thought of as a table of stored radiosity. No form factors are computed for updates to the table entry when surface j is outside the user's view. The value is only incremented by the amount of radiosity that the surface is to contribute. When the FOV angle changes, due either to user interaction or the decay interval, the radiosity for surface j can be updated as

$$\forall j \in V, \forall i \in \mathbb{B}_{ij} \neq 0: B_j = B_j + \rho_j \mathbb{B}_{ij} F_{ij} A_i / A_j$$

which guarantees that any surface j that becomes visible in the user's view and has stored radiosity it should receive from surface i will have its radiosity incremented by the stored radiosity table entry for the two surfaces, $\mathbb{B}_{ij}$. Once the radiosity for surface j has been updated, the table entry $\mathbb{B}_{ij}$ can be cleared to zero.

In the software implementation, a rectangular texture lends itself as a suitable data structure for the stored radiosity table, since texels can be indexed directly by the surface ID value, rather than a normalized texture coordinate. The texture size is N^2; shooting surfaces are column-indexed, and receiving surfaces are row-indexed. The table in Figure 25 is created for an environment with eight surfaces.

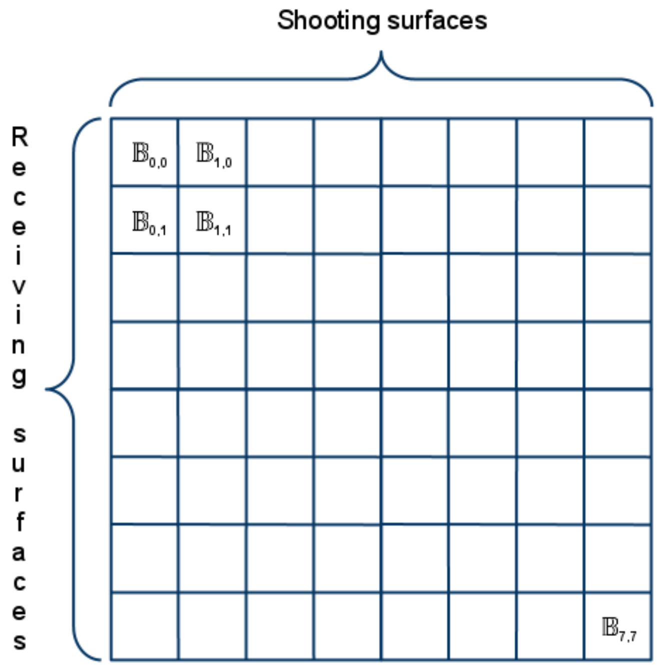


Figure 25: Stored radiosity table where rectangular texture coordinates correspond to table entry indices

5.3 View Visibility

It is important to determining if a surface lies in the user's view, since the stored radiosity table must be updated based on surface visibility. Visibility to the user's view is a simple test. The FOV is defined by an angle; if the angle between the user's look-at vector and the surface is greater than half the angle of the FOV, the surface is not contained in the FOV. Although this test can be performed on the CPU, the GPU does the work more effectively, since it can also eliminate hidden surfaces that would be contained in the FOV angle of the user's view.

The display mesh is rendered to an off-screen buffer, and an occlusion query is initiated for the displayed polygon of each surface. The perspective projection is set to the dimensions of the user's display, and the FOV angle is set to the value used for testing visibility. Once the mesh is rendered as in Figure 26, the occlusion query results determine if a surface is visible in the user's FOV. Because the perspective projection of the environment uses a combination of `gluPerspective` and `gluLookAt`, the FOV angle can range from (0,180), since the computed projection matrix contains a column with the value cot(FOV/2).

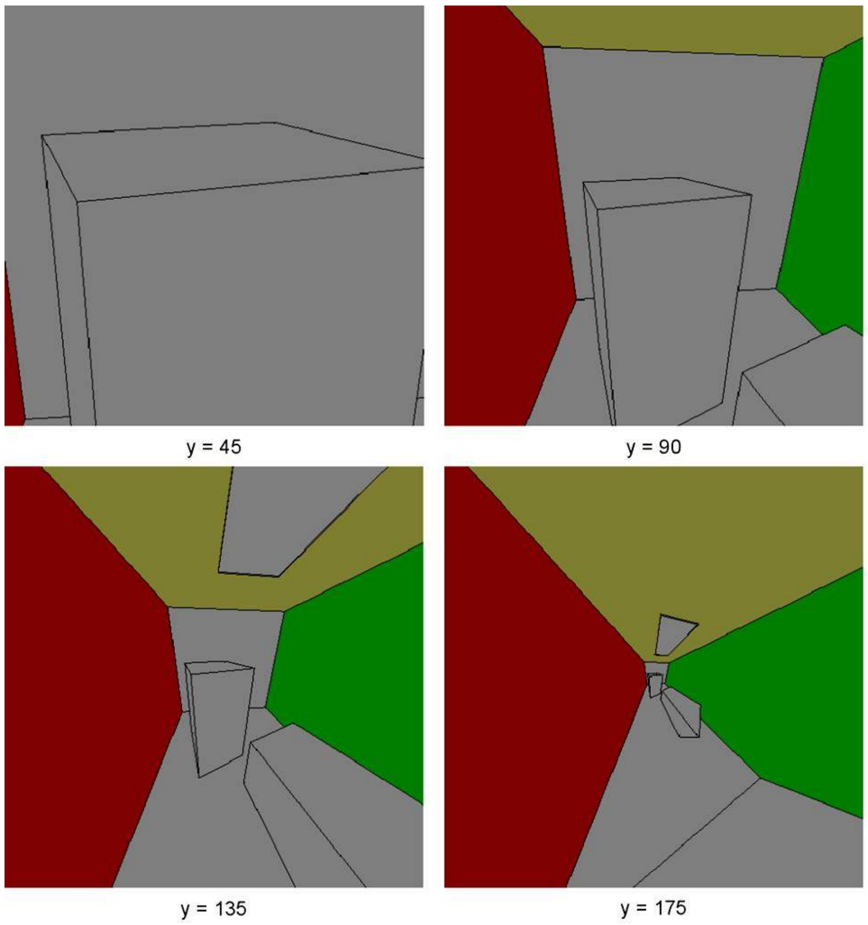


Figure 26: Visualization of FOV visibility test at FOV of y degrees

The results of the occlusion queries are placed in a vector mapped to GPU memory using a PBO, so surface visibility results can be accessed in a shader.

5.4 Stored Radiosity Table Gathering

Stored radiosity table entries are incremented by the amount of unshot radiosity a surface i needs to shoot to surface j. The value for the unshot radiosity that should be transferred is stored in GPU memory. Therefore, our stored radiosity table is also generated on the GPU to prevent the need to access radiosity textures on the CPU.

5.4.1 Table Entries

The rectangular texture containing the stored radiosity table is attached to an FBO with a one-to-one orthographic projection, and a stored radiosity table gathering shader is enabled for the stored radiosity table texture's single draw operation. Texture coordinates are rectangular, and correspond to the surface ID of each surface in the environment. An example of the stored radiosity table is shown in Figure 27.

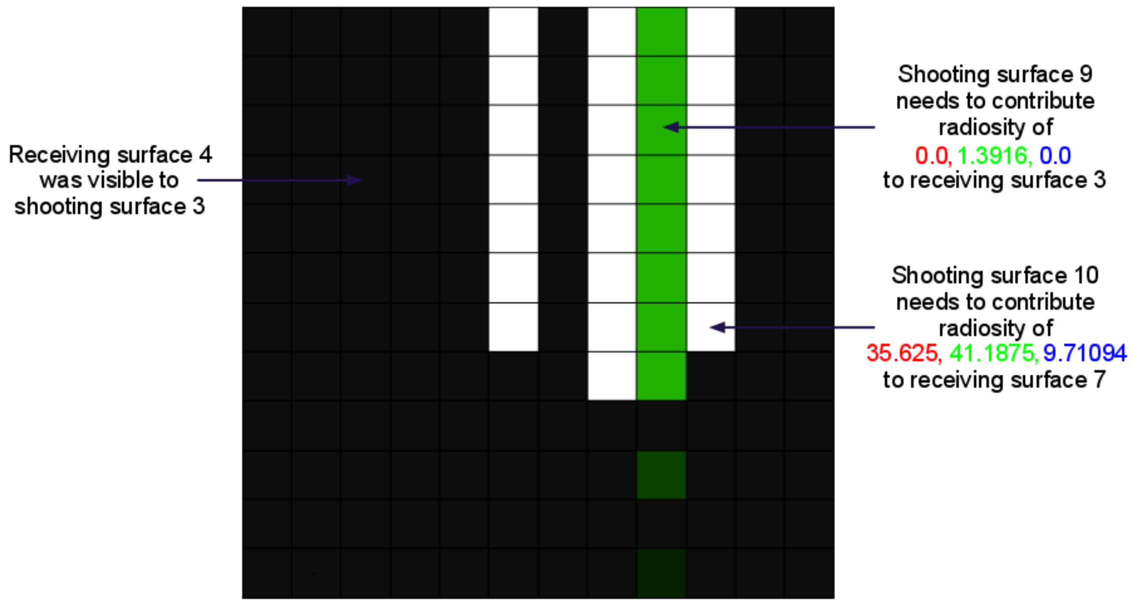

Receiving surface 4 was visible to shooting surface 3
Shooting surface 9 needs to contribute radiosity of 0.0, 1.3916, 0.0 to receiving surface 3
Shooting surface 10 needs to contribute radiosity of 35.625, 41.1875, 9.71094 to receiving surface 7

Figure 27: Stored radiosity table with selected table entry values and their meanings

Since the stored radiosity table is updated one shooting surface at a time, the shader is able to discard fragments for every texture column except the one indexed by the surface ID of the active shooting surface i. Since the vector of surface occlusion query results have been mapped to the GPU using a PBO, the shader is able to access the view visibility vector as a one-by-N rectangular texture, where N is the number of surfaces in the environment. The index value of each row in the shooting surface column is also used to index the view visibility vector and determine if the receiving surface j indexed by the row index value was visible to the shooting surface i. If surface j was visible to surface i, there is no need to update the stored radiosity table, since the radiosity will be contributed to the surface during progressive refinement. If surface j was not visible to surface i, the table is incremented at the texel in column i and row j by the amount of unshot radiosity the active shooting surface should otherwise be contributing, ΔB_i. This is obtained from the ΔB_i texture that is input to the shader.

5.4.2 Multiple Shooting Locations

In the software implementation of progressive refinement radiosity on the GPU, radiosity is shot from level n of the shooting surface's unshot radiosity texture. Since the size of each successive mipmap is the squared root of the previous mipmap, the number of shooting locations is reduced by a magnitude of two for each mipmap level above zero. If there are N^2 table entries in the stored radiosity table, each shooting surface i is allowed one stored radiosity value for surface j. This is the equivalent of shooting from the highest-level mipmap for each shooting surface, thereby shooting the entire surface's unshot radiosity from the center of the surface. Shooting all radiosity from the center of the surface can produce an overly simplified image if the surfaces are large enough. This can lead to artifacts such as those shown in Figure 28.

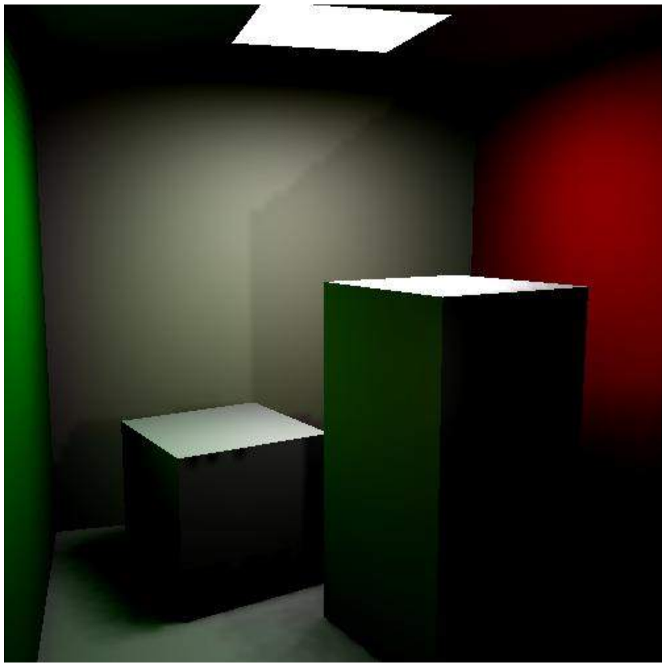


Figure 28: Artifacts from shooting unshot radiosity from one central point on a surface

To alleviate these artifacts, the creation of the stored radiosity table can be generalized to store radiosity values for more than one shooting location for surface i. If the stored radiosity table is expanded to hold up to k shooting locations per surface, the table entries can be updated by using the shooting texture coordinates to direct stored radiosity values into one of k bins, shown in Figure 29. When shooting from the surface location that maps to texture coordinates (u, v) in the ΔB_i texture of shooting surface i, a stored radiosity table draw operation checks whether the fragments from the draw operation are in the column that matches the bin that texture coordinates (u, v) fall into for surface i, and discards all other column fragments.

The stored radiosity table requires $O(kn^2)$ memory, where k is the number of shooting locations stored per surface, so memory requirements are quadratic. The software implementation allows storage of up to 256 bins per shooting surface.

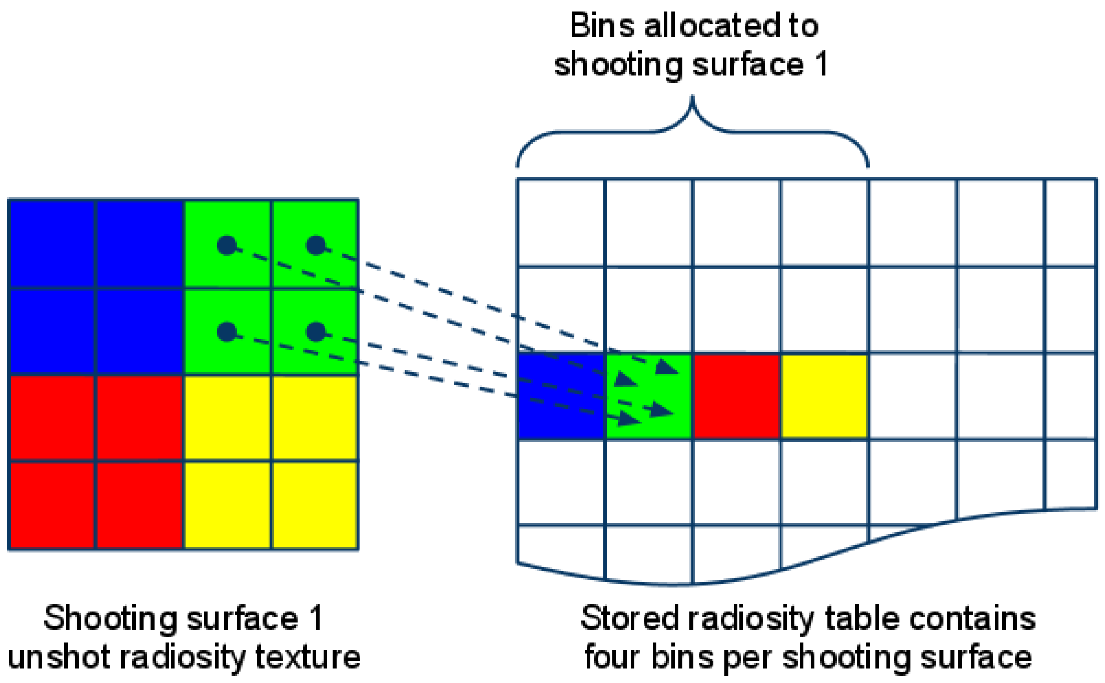

Bins allocated to shooting surface 1
Shooting surface 1 unshot radiosity texture
Stored radiosity table contains four bins per shooting surface

Figure 29: Shooting surface 1 with four radiosity values directed to appropriate bin for receiving surface 3

5.4.3 Visibility Vector

When gathering to the table entries in a draw operation on the stored radiosity table, it is useful to give a context for entries that are being written to the table. This contextual information can be used when the table entries are retrieved. The extra bookkeeping allows quick correlation between which receiving surfaces have stored radiosity table entries and which receiving surfaces are now visible because of a change in the user's view. This prevents an $O(n^2)$ search of table entries each time the user's view changes. The contextual information takes the form of a visibility vector in the first column of the texture used for the stored radiosity table.

When the stored radiosity table gathering draw operation takes place, the shader that writes to the table discards entries not in the shooting surface column or first column. The shader uses each row index value in the first column to read the occlusion query results vector that is mapped as a rectangular texture. If receiving surface j that corresponds to the row index j was not visible, the texel in row j of the visibility vector is filled with the color white, as seen in Figure 30. This color is a Boolean indication that receiving surface j has stored radiosity table entries.

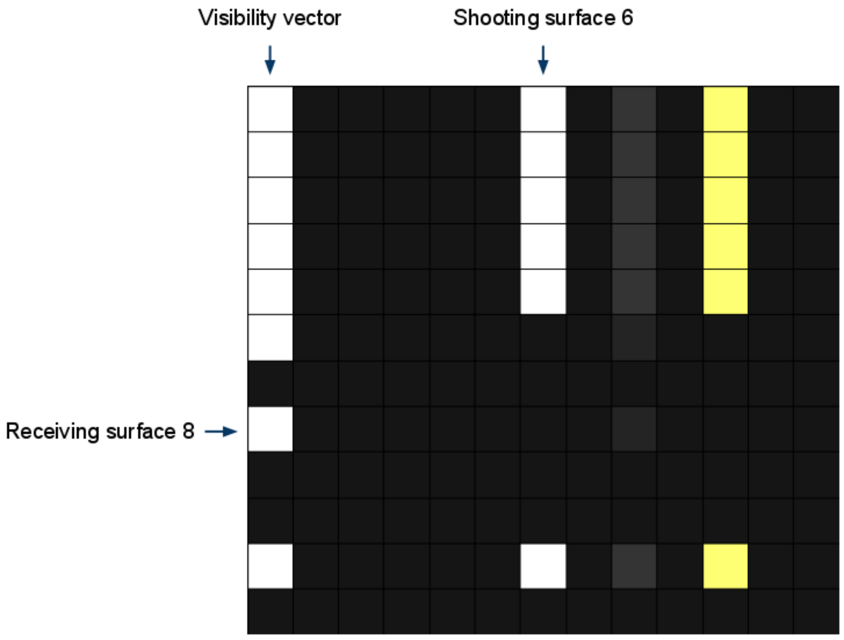


Figure 30: Visibility vector in a stored radiosity table texture

The receiving surfaces that have stored radiosity table rows containing entries may be quickly identified by reading this resulting vector after a stored radiosity table gathering draw operation.

5.4.4 Shooting Locations

When reconstructing the conditions necessary to propagate radiosity from shooting surface i to receiving surface j, it is necessary to redraw the stereographic hemisphere at the shooting location on shooting surface i so that the visibility test can be performed for each element in surface j. Section 4.1.3 describes how shooting locations for a surface are computed using the GPU. Rather than redraw these shooting location values for each surface that needs to shoot entries in the stored radiosity table to receiving surfaces, a vector of shooting locations is added to the top of the stored radiosity table texture, shown in Figure 31. The value in the vector at index i

corresponds to the shooting location bin that is stored in column i of the rectangular texture.

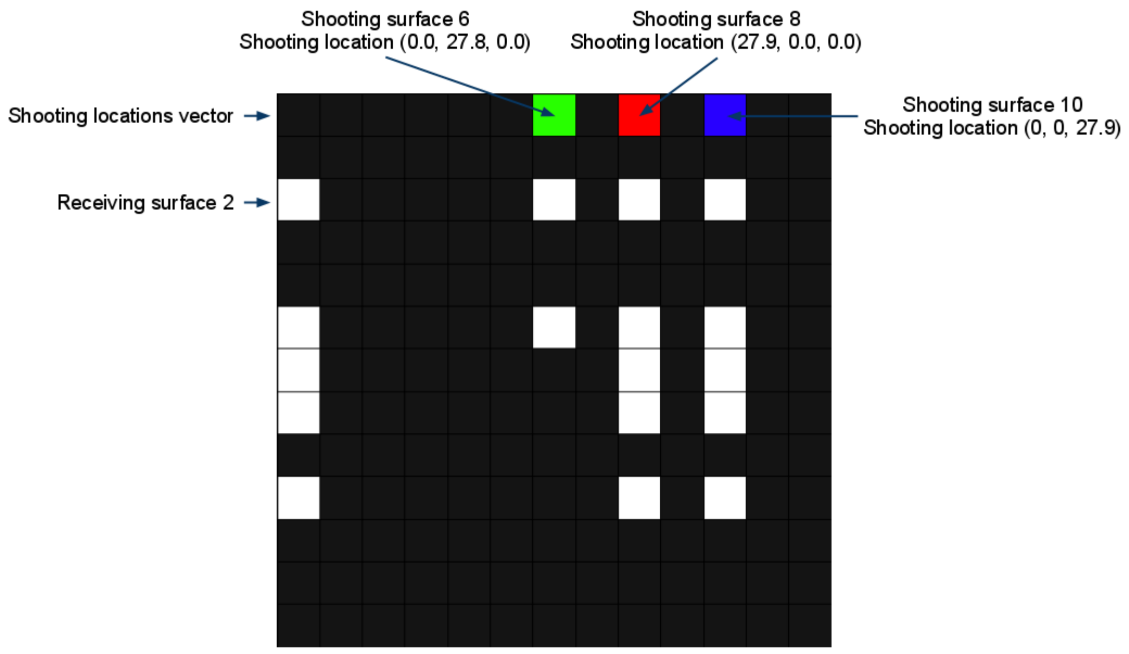

Shooting surface 6
Shooting location (0.0, 27.8, 0.0)
Shooting surface 8
Shooting location (27.9, 0.0, 0.0)
Shooting locations vector
Shooting surface 10
Shooting location (0, 0, 27.9)
Receiving surface 2

Figure 31: Shooting locations vector in a stored radiosity table texture

A 2D texture array is created to store shooting locations, which are written as vector entries. The layers in the texture array can be indexed using the same column index that is used to write to the rectangular texture. Each texture array layer is initialized by drawing a polygon using the vertex values of the surface as colors. The resulting texture array layer i contains shooting locations for the surface i in each texel. Since the software implementation stores up to 256 shooting locations for each surface, the texture array has dimensions $[16,16,N]$, where N is the number of surfaces in the environment. To store fewer shooting locations per surface, higher mipmap levels of the texture array can be accessed, which give a reduced number of shooting locations. The mipmap is indexed using the same texture coordinates that are used to write the unshot radiosity value from surface i to the stored radiosity table.

5.5 Stored Radiosity Table Distribution

Whenever the user view changes because of user interaction or the decay interval growing the simulated FOV angle, the stored radiosity table must be checked to see if any surfaces have become visible which have stored radiosity in table entries. If any of these surfaces have become visible, they are bucketed by the shooting location that needs to shoot radiosity to them. The stereographic item buffer must be regenerated from the shooting location for a receiving surface's radiosity textures to be updated, which requires a render pass of the environment geometry. Bucketing receiving surfaces by shooting surfaces allows the most work to be done per stereographic item buffer generation.

5.5.1 Becoming Visible Vector

The projection matrix must be set on the CPU when shooting radiosity from the stored radiosity table entry for a shooting surface i to a receiving surface j. Therefore, it is important to know which surfaces become visible whenever the view changes.

To determine if a surface is becoming visible, the environment is rendered from the user's view. Occlusion queries are initiated for each surface polygon to determine if the surface has at least one visible projected pixel. These occlusion queries are mapped to a one-by-N rectangular texture using a PBO. By comparing the visibility of surfaces from the user's view with the visibility vector in the stored radiosity table texture, a shader is able to determine which surfaces have become visible in and should have unshot radiosity distributed to them from the shooting surface. The result of the comparison is a "becoming visible vector," used by the CPU to bucket receiving surfaces by their shooting surfaces.

5.5.2 Stored Radiosity Table Entry Retrieval

In the software implementation, a rectangular texture is created to retrieve the contents of the stored radiosity table which should be propagated to newly visible surfaces and a shader is enabled which performs the logic to retrieve table entries. The shader checks the texels under each column in the stored radiosity table texture to see if that shooting surface has stored radiosity table entries. If it does, the value in the shooting location vector is written to the retrieved radiosity table texture. The value of the visibility vector in the stored radiosity table at index i is compared to the value in the occlusion queries texture at index i to create the “becoming visible” table in the first column of the retrieved radiosity table texture. All table entries in the stored radiosity table texture are written to the retrieved radiosity table texture if they are in a “becoming visible” surface’s row, shown in Figure 32.

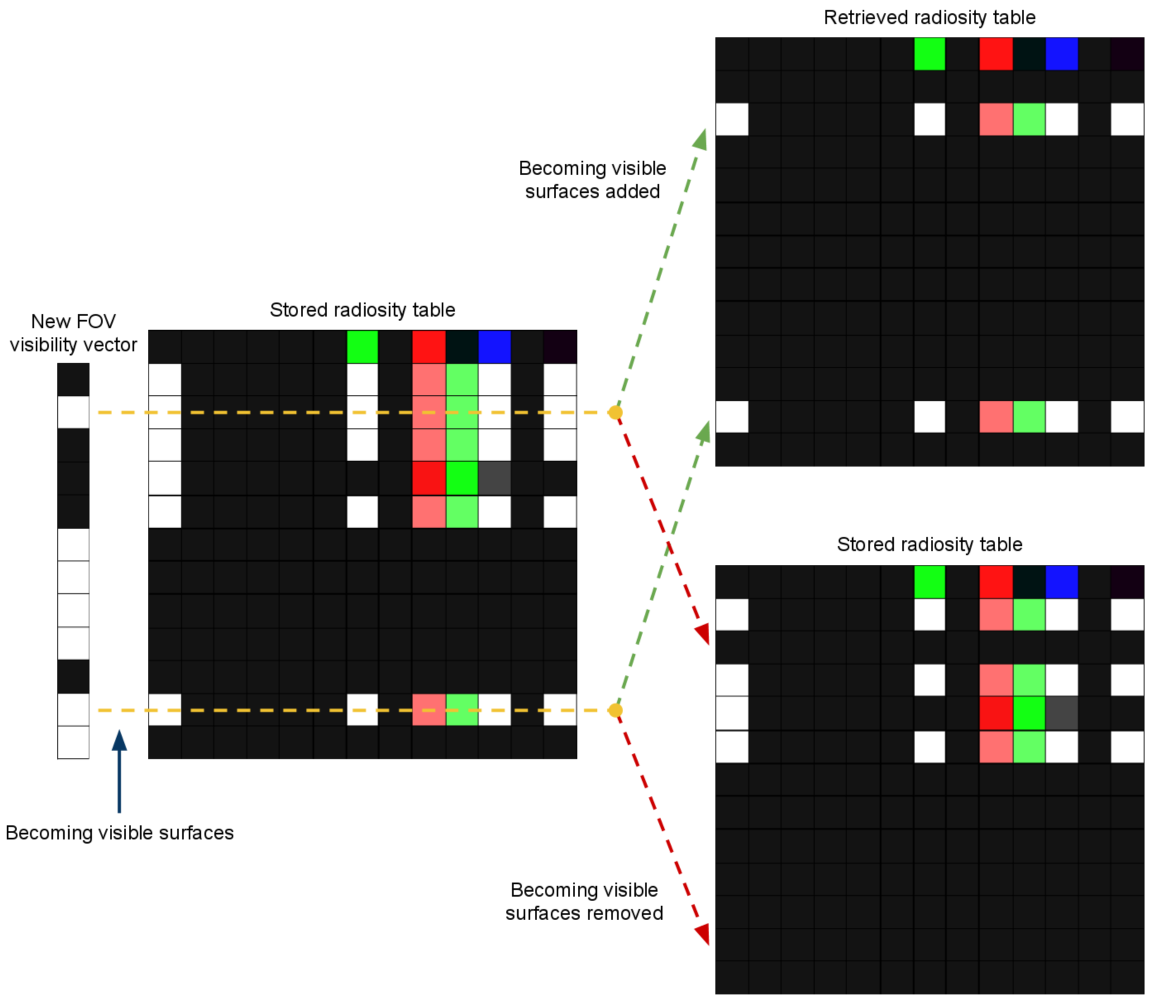

Retrieved radiosity table
Becoming visible surfaces added
New FOV visibility vector
Stored radiosity table
Becoming visible surfaces
Stored radiosity table
Becoming visible surfaces removed

Figure 32: Retrieving stored radiosity table entries for newly visible surfaces

The stored radiosity table texture is attached to the FBO along with the retrieved table entries texture, using MRT. If a receiving surface is becoming visible, the row for that surface is “cleared” in the stored radiosity table by writing out the color black for each entry in the row. The surface’s entry in the visibility vector is cleared, as well. Additionally, if a shooting surface is determined not to have any remaining radiosity table entries, the shooting location is cleared from the stored radiosity table.

5.5.3 Stored Radiosity Table Entry Shooting

The first column of the retrieve table entries texture contains the becoming visible vector; the first row contains the shooting locations vector. Once the retrieved radiosity table texture draw operation is complete, these vectors are read back to the CPU using a PBO. The shooting locations vector provides a list of shooting surface locations that have radiosity table entries that need to propagate stored radiosity. The “becoming visible” table provides a list of receiving surfaces to which shooting surfaces need to shoot stored radiosity table entries.

The stereographic item buffer is rendered for each shooting location in the shooting locations list. Receiving surfaces that are in the “becoming visible” table are bucketed by shooting surface, minimizing the number of stereographic item buffers that must be generated. The receiving surface radiosity textures are updated using the same code that is used for shooting radiosity and importance. The shader that performs radiosity calculations for these surfaces is modified slightly to use the values in the retrieved table entries texture. The radiosity values in the table entries are accessed by using the index of the shooting surface i, the shooting location bin number, and the index of the receiving surface j. The area for the shooting location used in form factor calculations is derived by dividing the shooting surface into as many discs as there are shooting location bins.

6. RESULTS

Experiments were conducted to compare the results of progressive refinement, importance-driven, and perspective-driven radiosity. Links to videos of the results may be found in Appendix B. All images in the following section were taken from the linked videos. All environment models used were variations on the Cornell box: an enclosed box with a red and green wall, and a single light source. When displayed in Autodesk Maya using flat shading, the Cornell box appears as in Figure 33. The total surface area of the environment is 23,603 units. Textures are stored one-to-one with their corresponding surfaces, so this number equals the number of surface elements to which radiosity will be shot.

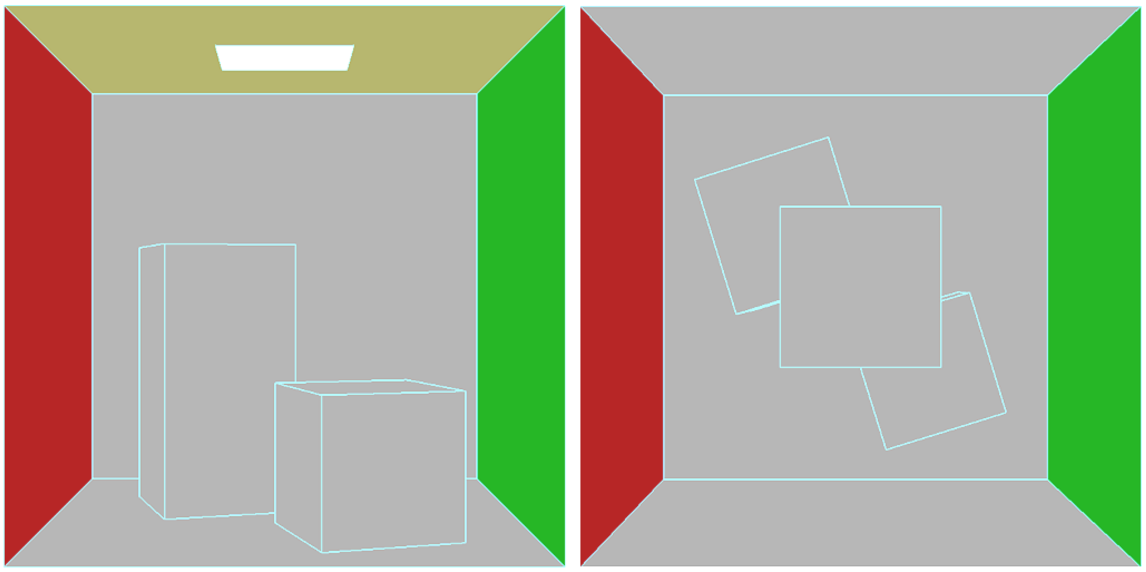


Figure 33: Cornell box displayed in Autodesk Maya from front and top perspectives

6.1 Single-Enclosure Environment

A single-enclosure environment was used to test the effectiveness of the progressive refinement radiosity implementation. It also served as a basis to which other results could be compared. Two runs of the radiosity system were conducted using progressive refinement and the single-enclosure Cornell box environment: one using the ambient term for display in Video 2, one without the ambient term in Video 1. Images from these videos are shown in Figure 34.

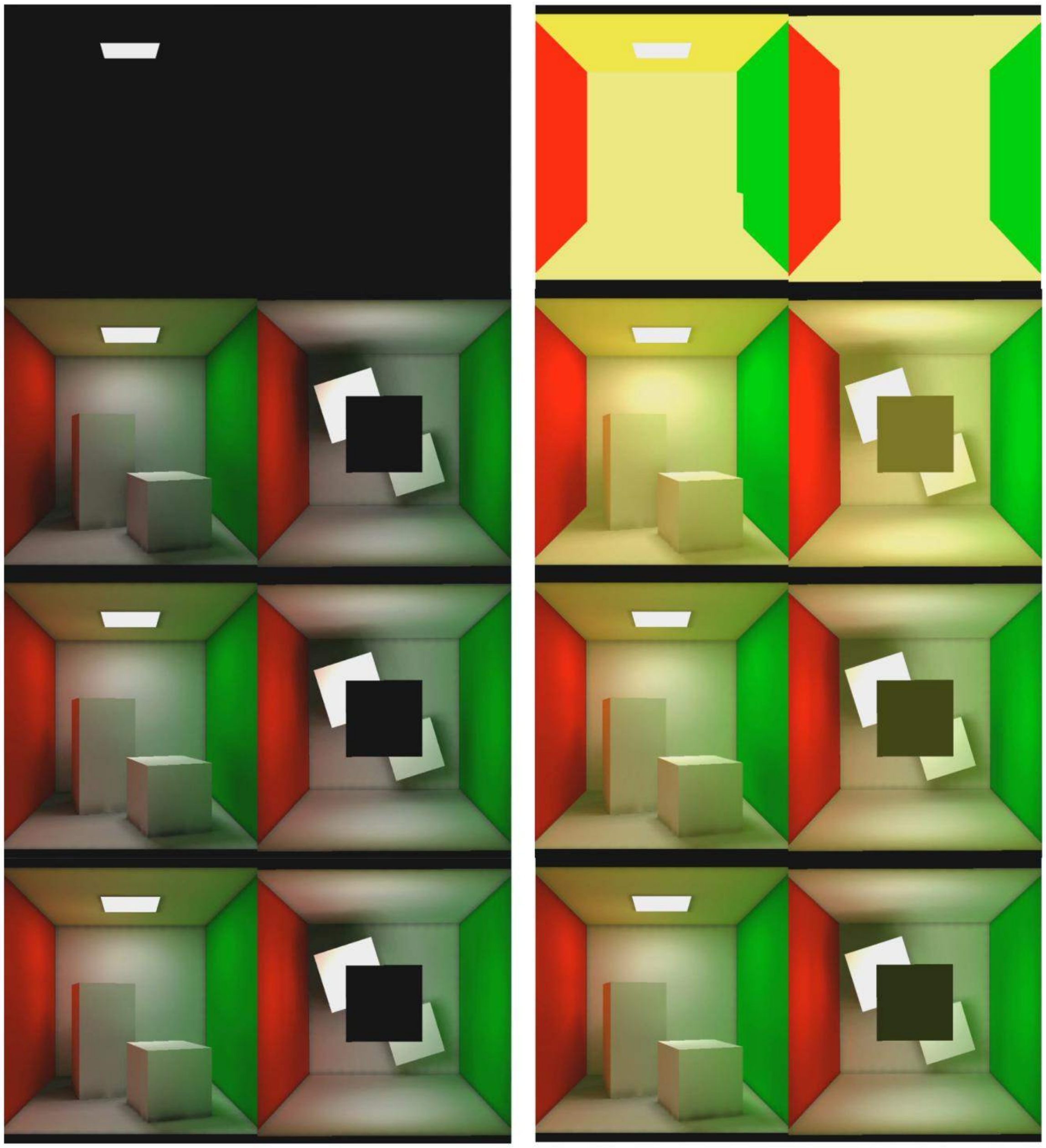


Figure 34: Progressive refinement with (right) and without the ambient term (left), at 5 second intervals

At five seconds, color bleeding from the red and green walls is reflected on the sides of the inner boxes. At 15 seconds, the ambient term is no longer noticeable and the runs are beginning to approach convergence. To demonstrate color bleeding that is more pronounced and the effect of the ambient term, the Cornell box is again rendered, this time using an exaggerated color scheme in Video 3 and Video 4, shown in Figure 35.

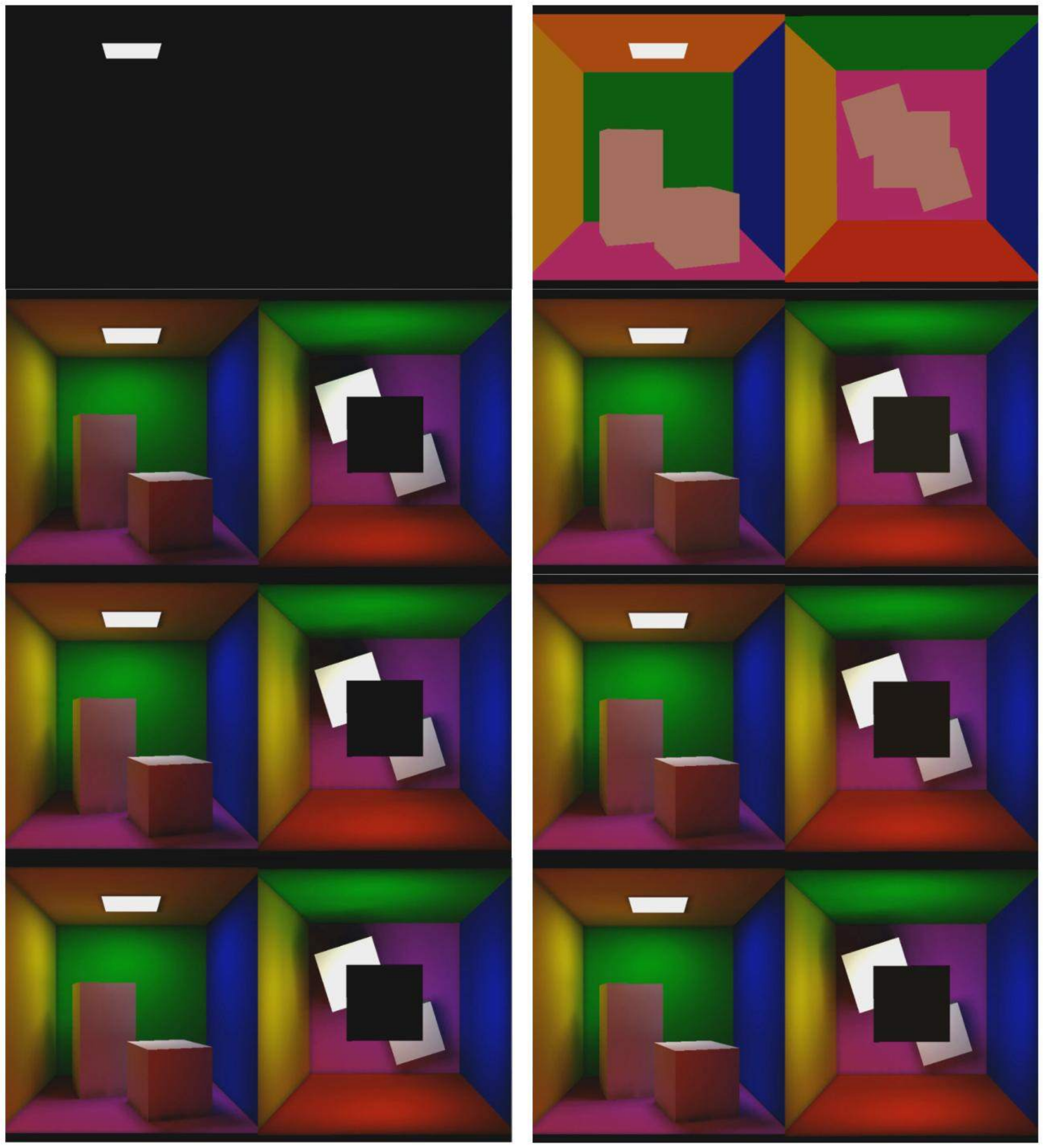


Figure 35: Progressive refinement shown in an environment using an exaggerated color scheme

6.2 Multiple-Enclosure Environment

A new environment was created by duplicating the Cornell box three times and placing one of the four boxes in its own quadrant, shown in Figure 36. This environment was created to determine the effectiveness of importance-driven radiosity. Progressive refinement will treat every box the same, since they are all identical to each other. Importance-driven, however, should perform radiosity calculations only for the single Cornell box in the user's view.

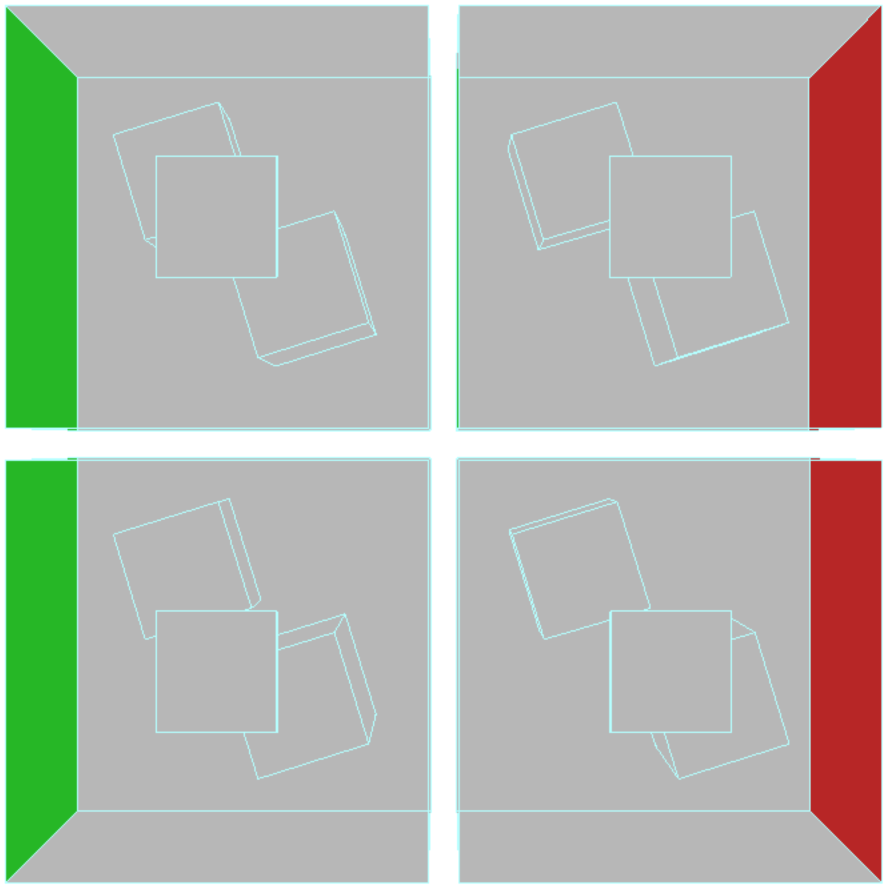


Figure 36: Multiple-enclosure environment displayed in Autodesk Maya from top perspective

The multiple-enclosure environment is used in a run of progressive refinement, captured in Video 5. Figure 37 shows the results and the times at which significant events occurred in the lighting process. The display contains two views: the user's view, and an omniscient view that has no effect on the rendered scene.

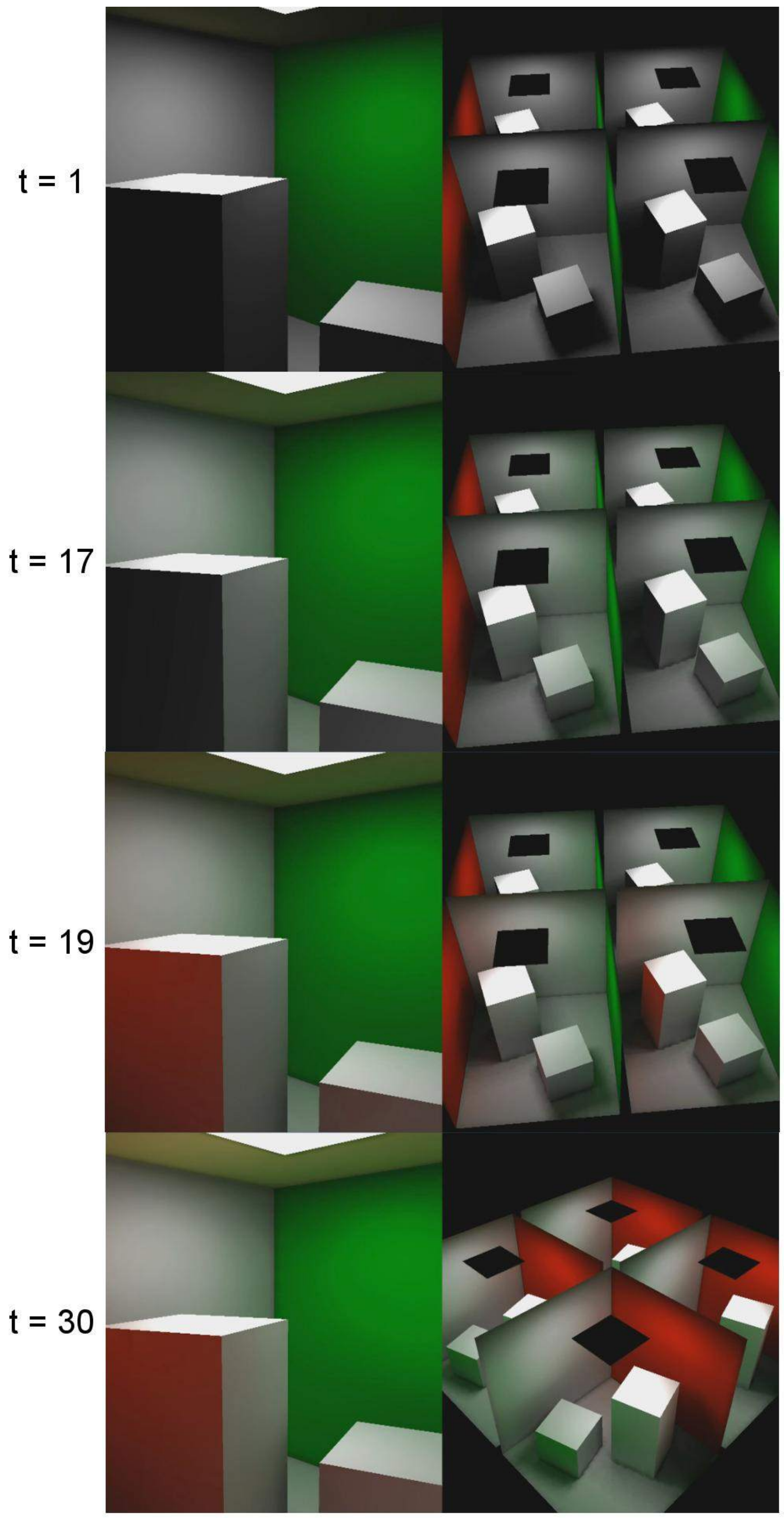


Figure 37: Time t, in seconds, of significant lighting events for progressive refinement radiosity

At one second, each of the four boxes in the environment has been lit. Only the lighting of one of these boxes contributes to the light visible to the user. At 17 seconds, radiosity is shot from the green wall of the user-visible box. At 19 seconds, the red wall also contributes its radiosity to the environment. At 30 seconds, the top of the smaller inner box contributes its radiosity, and the solution begins to approach convergence.

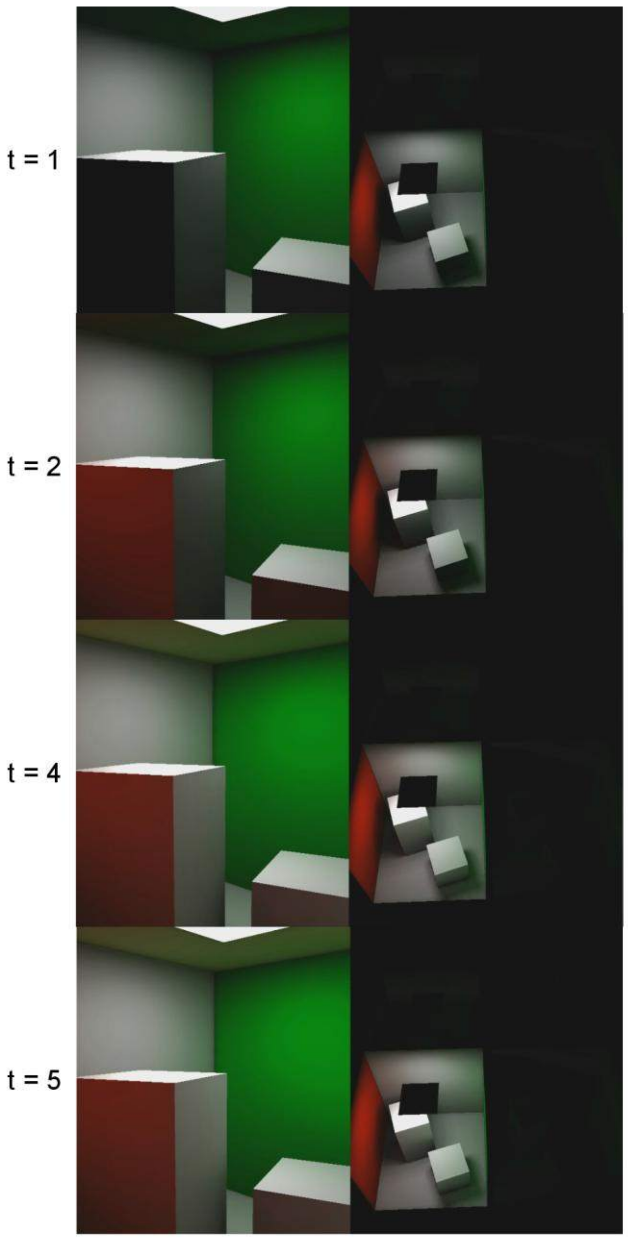


Figure 38: Time t, in seconds, of significant lighting events for importance-driven radiosity

The same environment is then used in a run of importance-driven radiosity, captured in Video 6. Figure 38 again shows timings at which significant events occur in the lighting of the environment from the user's view. At one second, the box in the user's view is lit. At two seconds, the red wall shoots its radiosity, and the green wall does the same at four seconds. At five seconds, the top of the small inner box contributes its radiosity.

The remainder of the captured video shows the user view being rotated, and the effect it has on the remaining boxes in the environment: when the user's view is focused on one of the four boxes, surfaces contained in that box are selected for shooting radiosity. The cause of this can be seen in Video 7, which displays importance values as intensity maps, shown in Figure 39.

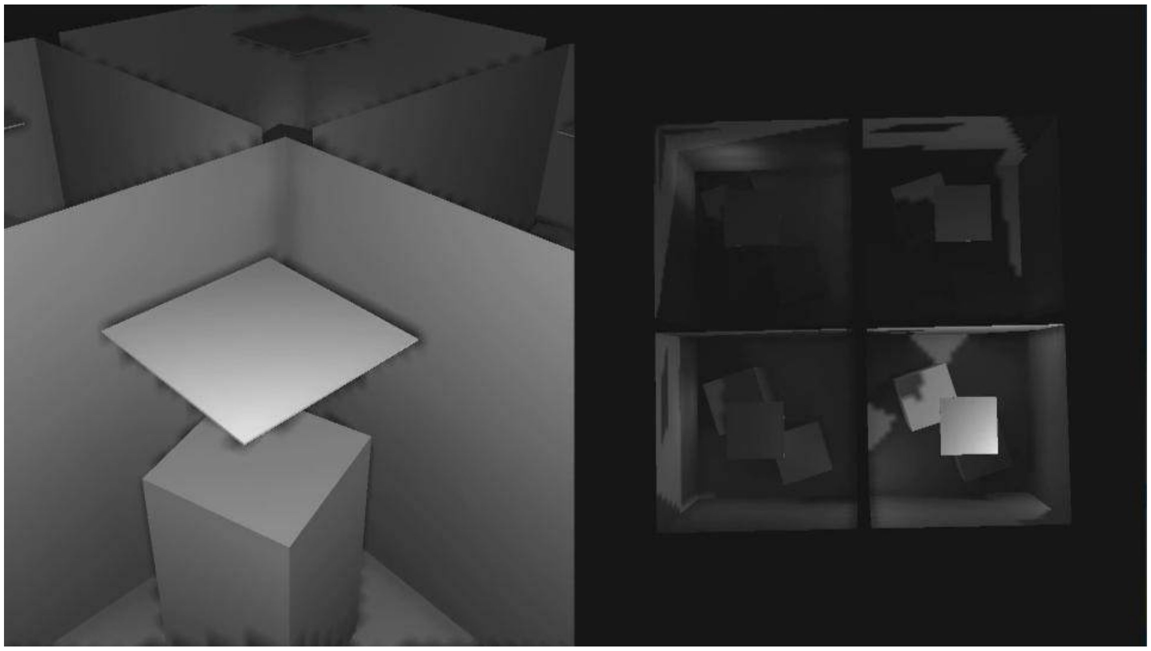


Figure 39: Importance values shown as intensity maps

6.3 Subdivided Single-Enclosure Environment

To create the next environment used for testing, the Cornell box model was subdivided twice using Autodesk Maya, as shown in Figure 40. This creates an environment with 16 times as many surfaces and light sources as the Cornell box model used in previous experiments.

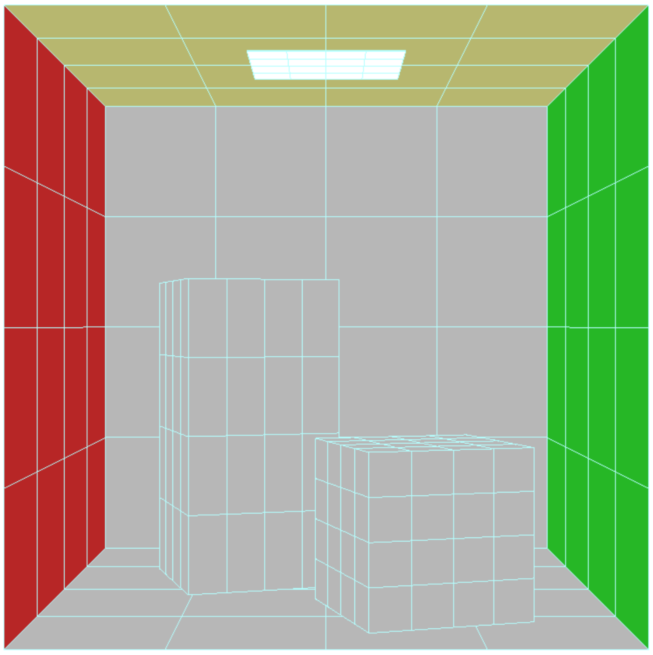


Figure 40: Subdivided Cornell box, displayed in Autodesk Maya

The subdivided model was tested using each of the three radiosity algorithms in the software implementation. Images from the videos generated during these tests are shown in Figure 41, Figure 42, and Figure 43. It was expected that progressive refinement would process the entire scene the most quickly, because of the low overhead of the original algorithm, but that perspective-driven radiosity would be able to focus radiosity calculations to a subset of surfaces in the environment contained in the user's view and light them more quickly.

For each run, the FOV of the user's view is set to 10 degrees, to limit the number of visible surfaces. The camera begins by viewing the front of the model, and is pointed to the back right corner where the white wall, white floor, and white back wall meet. The progressive refinement run is captured in Video 8.

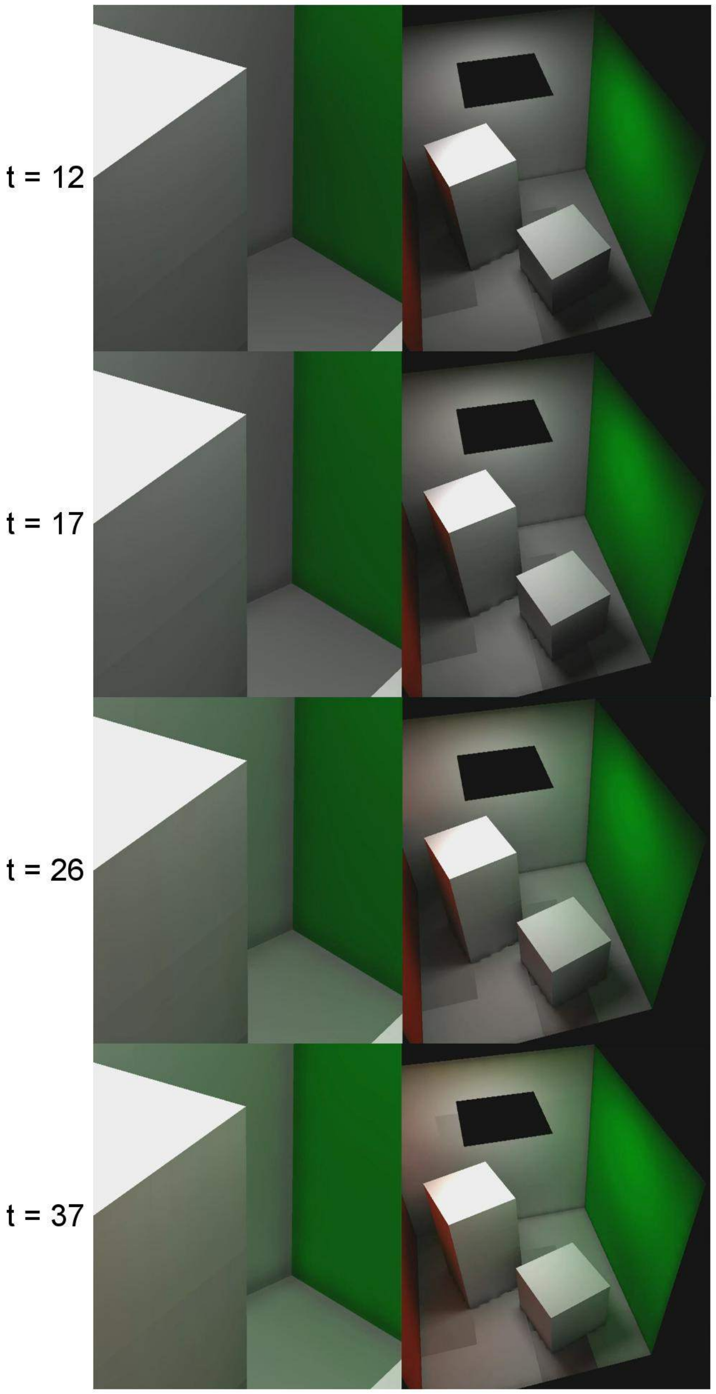


Figure 41: Time t, in seconds, of four significant lighting events for progressive refinement radiosity in a single enclosure environment

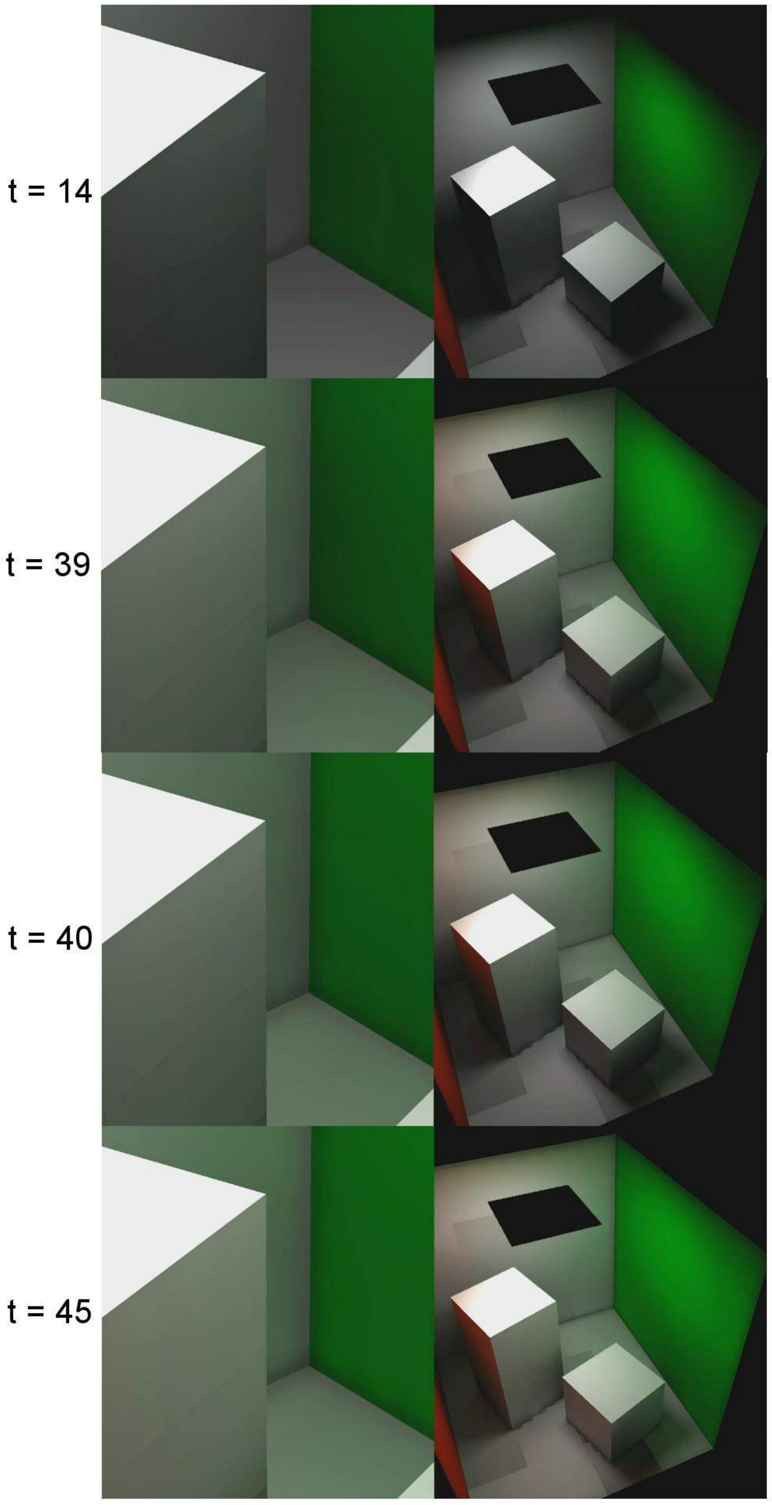


Figure 42: Time t, in seconds, of four significant lighting events for importance-driven radiosity in a single enclosure environment

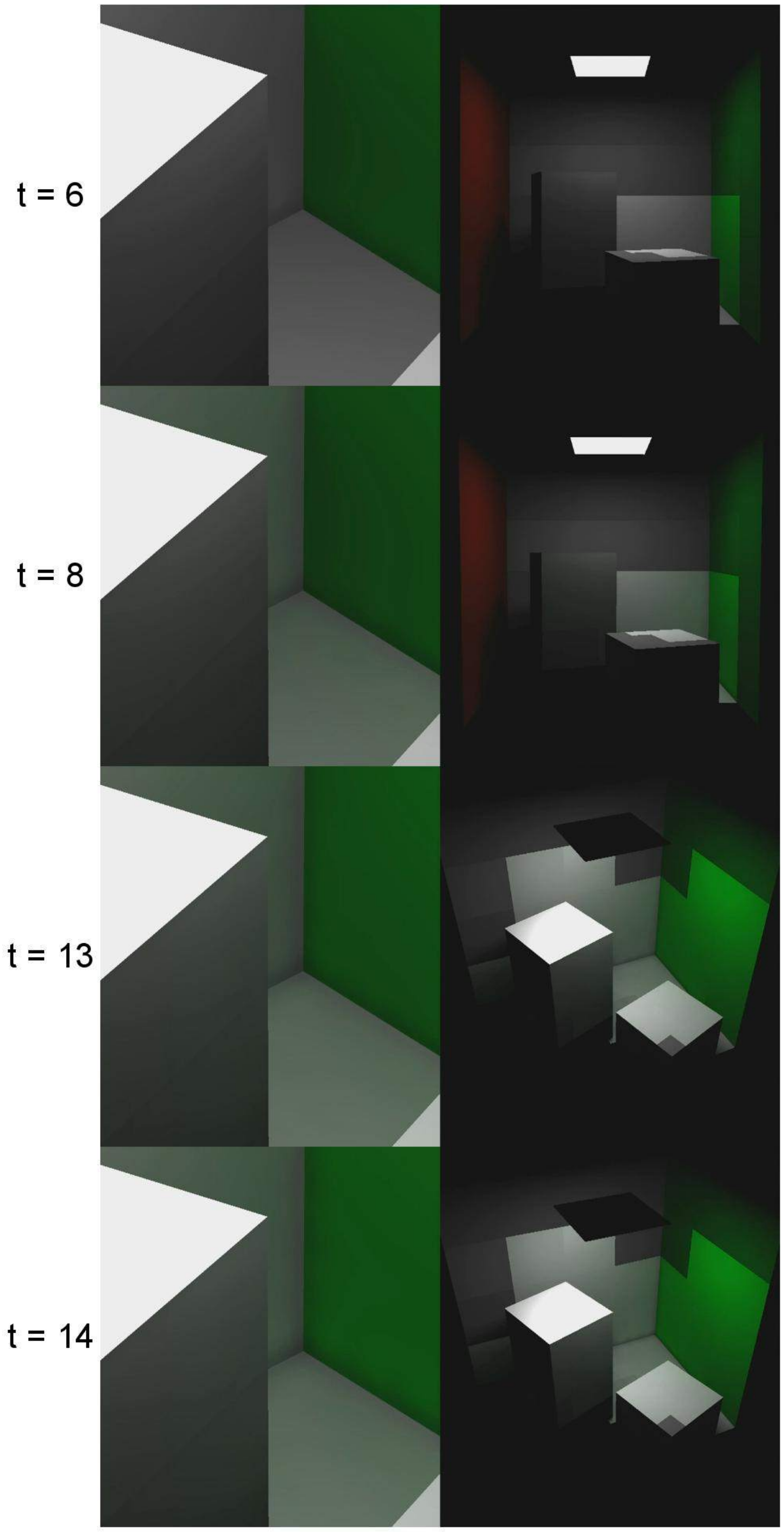


Figure 43: Time t, in seconds, of four significant lighting events for perspective-driven radiosity in a single enclosure environment

Progressive refinement radiosity must solve the radiosity calculations for each surface in the environment. The small corner of the box receives as much computational attention as any other area in the environment. Importance-driven radiosity, captured in Video 9, has more computational overhead than progressive refinement because of additional geometry passes and rendering to importance textures. All surfaces in the environment receive nearly equal importance, due to all propagated importance being confined to the single enclosure. Therefore, the importance-weighted shooting surface chosen to shoot its radiosity next is similar to the surface chosen by the progressive refinement algorithm. This amounts to no improvements being made towards a quicker convergence of the lighting for the user's view.

Perspective-driven radiosity limits radiosity calculations to the surfaces visible from the user's view, and is able to shoot radiosity multiple times to the few visible surfaces, seen in Video 10. The ten-second decay constant gradually expands the FOV of the user's view, encompassing additional surfaces in the area-of-interest.

Movement of the user's view and its effect on the lighting of a scene using perspective-driven radiosity is demonstrated in Video 11, of which an image is shown in Figure 44. The decay constant is reduced to one second, and the view is moved across the subdivided environment with an exaggerated color scheme. Surfaces are lit when they fall within the user's view, displayed in the left pane. When the view pauses, additional surfaces are quickly included in the area-of-interest.

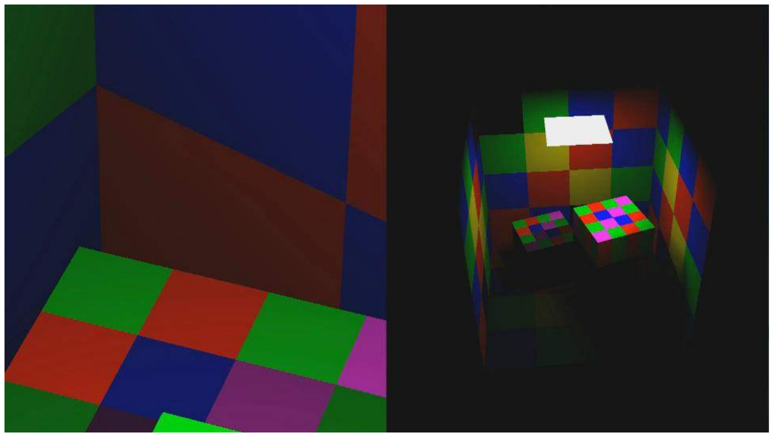


Figure 44: Demonstration of the effect of movement of the user's view in perspective-driven radiosity

6.4 Subdivided Multiple-Enclosure Environment

A final experiment was conducted with the multiple-enclosure environment previously used to demonstrate the importance-driven radiosity algorithm. The model containing multiple Cornell boxes was subdivided once using Autodesk Maya, as shown in Figure 45. Each enclosure contains four times as many surfaces and lights because of this subdivision.

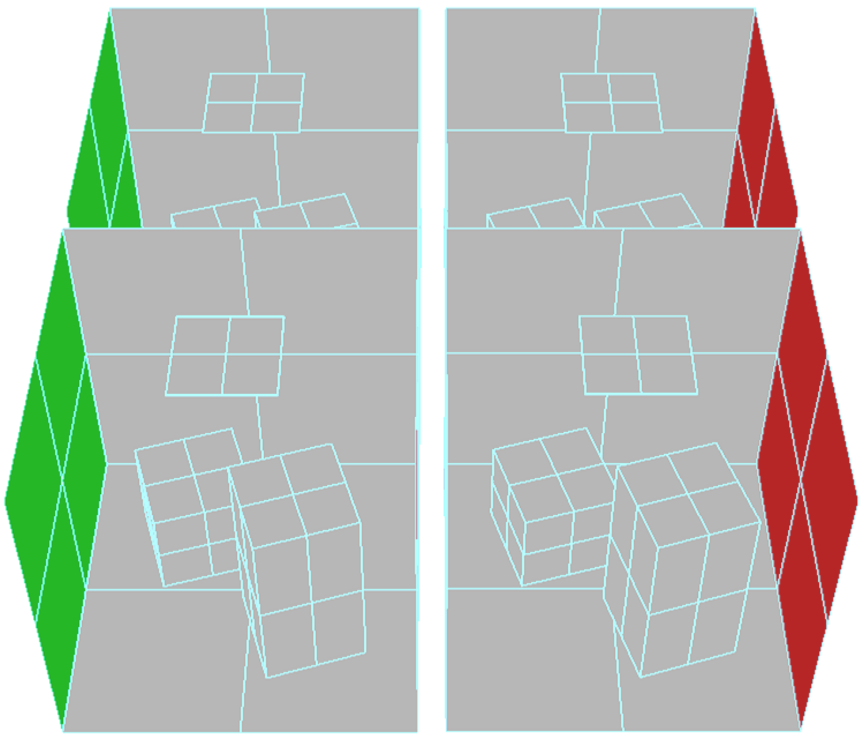


Figure 45: Multiple subdivided Cornell boxes, shown in Autodesk Maya

It was expected that this environment would demonstrate the ability of perspective-driven radiosity to focus radiosity calculations to a user-defined subset of surfaces. When perspective-driven radiosity is added, the surfaces in the user's view converge towards their final lit colors the most quickly. The results of this experiment are seen in the images in Figure 46, Figure 47, and Figure 48, which are taken from Video 12, Video 13, and Video 14, respectively.

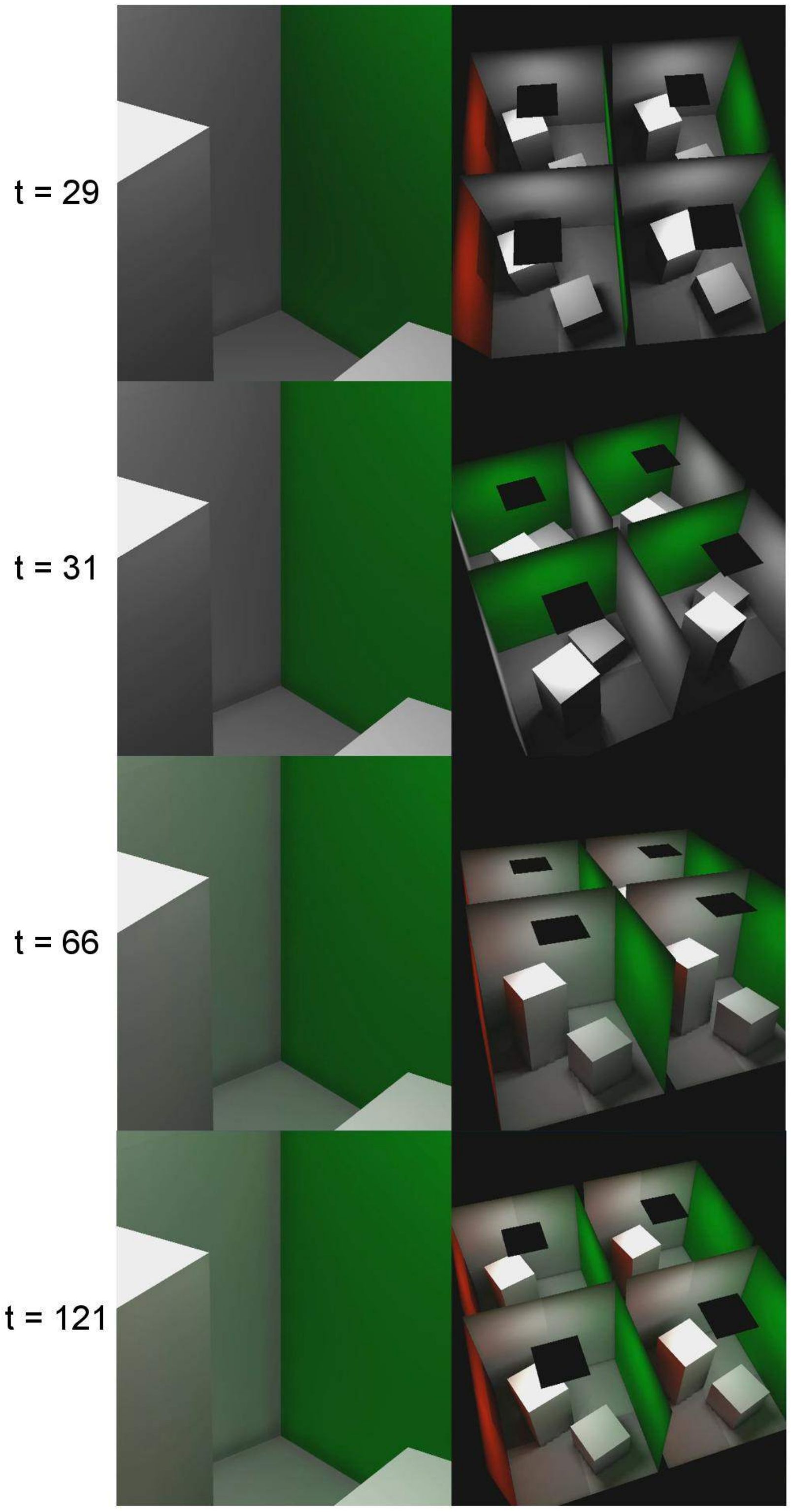


Figure 46: Time t, in seconds, of four significant lighting events for progressive refinement radiosity in an environment containing multiple enclosures

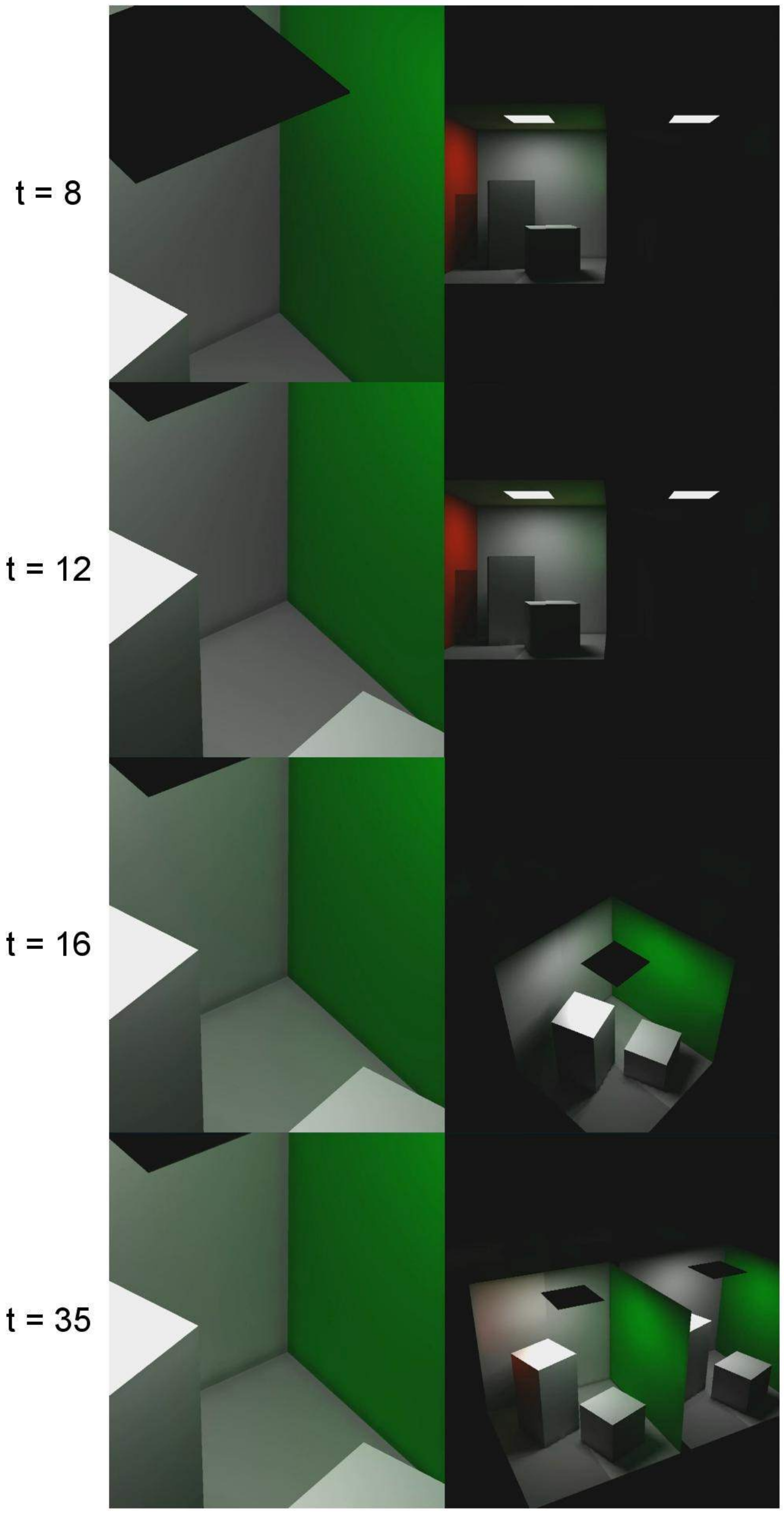


Figure 47: Time t, in seconds, of four significant lighting events for importance-driven radiosity in an environment containing multiple enclosures

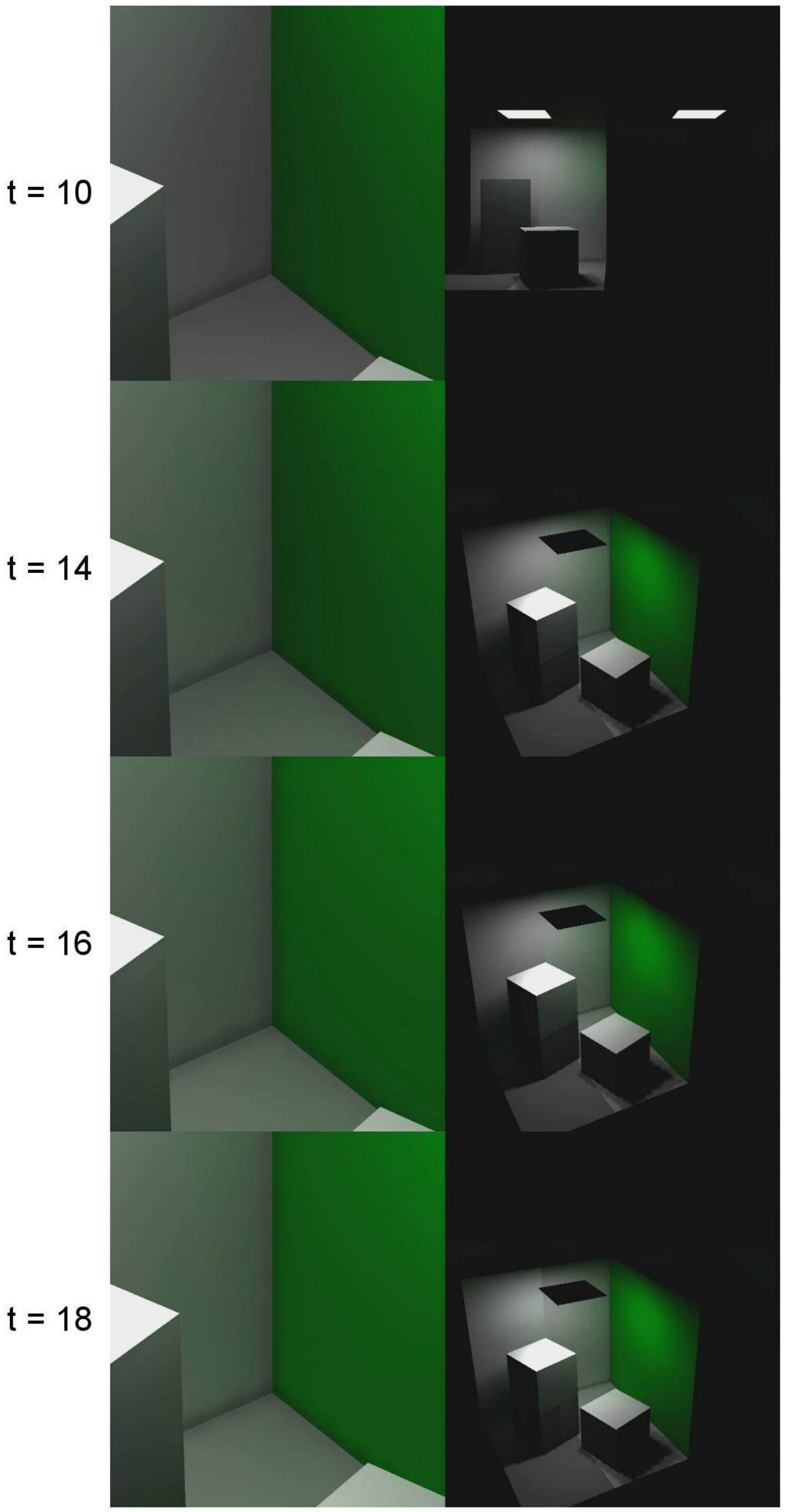


Figure 48: Time t, in seconds, of four significant lighting events for perspective-driven radiosity in an environment containing multiple enclosures

7. CONCLUSIONS

7.1 Performance

During research for this thesis, a method was developed in concordance with the thesis proposal which limits radiosity calculations to the area-of-interest defined by the user’s view: the combination of their point-of-view (POV) and field-of-view (FOV). This was successfully achieved through implementation of the perspective-driven radiosity algorithm, using the GPU for radiosity calculations. Additional work on the software implementation could potentially expand the usefulness and effectiveness of the perspective-driven algorithm.

7.2 Limitations

Because of the overhead of rendering multiple geometry passes and the additional memory associated with the algorithm, perspective-driven radiosity is limited in its applicability to any general environment model.

Several limitations are imposed due to current GPU hardware limitations. For instance, the maximum dimensions of a texture on the NVIDIA Quadro 5000 are 8192 by 8192 texels. This means that in the current implementation, no surface can be created with a texture larger than this size.

Additionally, the maximum dimensions of a 3D texture are 2048 by 2048 by 2048 texels. This limit is also placed on the number of layers in 2D texture arrays that are created, such as the one used to store surface shooting locations for generation of the stored radiosity table. The use of this 2D texture array would limit the number of surfaces in the current implementation to 2048, since each layer in the texture array is used to store position information for a surface.

The stored radiosity table is created as a 2D texture. If the maximum number of shooting bins is kept for each shooting surface, this results in 256 texture columns of stored radiosity entries per shooting surface. When the visibility vector is accounted for, this leaves enough texture columns for 31 surfaces in the environment. This would be an unrealistic limit for most applications, so the usefulness of multiple shooting bins is reduced.

In terms of artistic limitations, the perspective-driven algorithm works best with environments where the surfaces are highly subdivided. If the corner of a large surface lies partially within the user's view, the entire surface will still be considered visible to the user, and will be shaded over the entire texture. By subdividing the same large surface into smaller surfaces, only the subdivided surfaces which are visible to the user will be shaded.

This characteristic of perspective-driven radiosity means that best results will be achieved when the algorithm is applied to environments such as those presented in Section 6.4.

7.3 Future Work

For the demonstration of the perspective-driven algorithm, some techniques from the original paper on GPU progressive refinement radiosity were omitted which could be added to improve appearance. The performance of the software implementation could also benefit from further optimizations for execution on the GPU. In addition, new and upcoming GPU features that are supported by the OpenGL API could improve performance if utilized correctly.

7.3.1 Adaptive Subdivision

As explained in Section 4.1.7, adaptive subdivision was omitted from the software implementation. The addition of adaptive subdivision would improve the appearance of lighting results by providing higher resolution textures to surfaces that have high frequency radiosity gradients. The original implementation of progressive refinement radiosity on the GPU provided a simple solution to this problem, using texture quadtrees [4].

7.3.2 GPU Optimization

Optimizing a software for the best performance on the GPU is both a science and an art. It requires an understanding of the lower level execution details of the GPU, as well as knowledge of how best to design a computer graphics problem so it is simplified without sacrificing imaging quality.

OpenGL's immediate mode was used for rendering all geometry in this thesis, which is a bottleneck for performance. Using vertex buffer objects (VBOs) or display lists would have a positive impact on the performance of all three algorithms. A VBO places vertex data in GPU memory so it is not required to send model geometry across the graphics bus every time a polygon is drawn.

In addition to the elimination of immediate mode drawing, more traditional computer graphics optimization techniques could be employed, such as view frustum culling. This would use the CPU to determine which areas of the model could be skipped when drawing geometry, particularly when generating the user view item buffer for importance-driven and perspective-driven radiosity.

7.3.3 Mesh Tessellation

Rendering the stereoscopic item buffer requires a subdivided surface because of the straight rasterization of lines when transforming model vertexes in the stereoscopic vertex shader. Since the subdivided polygons used to generate the stereographic item buffer are based exactly on geometry already being sent to the GPU, subdividing model geometry on the GPU would save both time and memory. This could potentially be accomplished using instanced tessellation, or the tessellation engines found on the latest generation of GPUs.

REFERENCES

[1] C.M. Goral, K.E. Torrance, D.P. Greenberg, and B. Battaile, "Modeling the interaction of light between diffuse surfaces," *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 213-222, 1984.

[2] P. Bekaert, and Y.D. Willems, "Importance-driven progressive refinement radiosity," *Proceedings of the 6th Eurographics Workshop*, pp. 316-325, 1995.

[3] M.F. Cohen, S.E. Chen, J.R. Wallace, and D.P. Greenberg, "A progressive refinement approach to fast radiosity image generation," *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 75-84, 1988.

[4] G. Coombe, M.J. Harris, and A. Lastra, "Radiosity on graphics hardware," *Proceedings of Graphics Interface 2004*, pp. 161-168, 2004.

[5] B.E. Smits, J.R. Arvo, and D.H. Salesin, "An importance-driven radiosity algorithm," *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 273-282, 1992.

[6] D. Göddeke, "GPGPU - basic math tutorial," http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html, 2010.

[7] M.F. Cohen, and D.P. Greenberg, "The hemi-cube: a radiosity solution for complex environments," *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 31-40, 1985.

[8] D.S. Immel, M.F. Cohen, and D.P. Greenberg, "A radiosity method for non-diffuse environments," *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, 1986.

[9] J.R. Wallace, K.A. Elmquist, and E.A. Haines, "A ray tracing algorithm for progressive radiosity," *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 315-324, 1989.

[10] P.S. Heckbert, "Adaptive radiosity textures for bidirectional ray tracing," *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, 1990.

[11] P.-P. Sloan, J. Kautz, and J. Snyder, "Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments," *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 527-536, 2002.

[12] A.T. Campbell, and D.S. Fussell, "Adaptive mesh generation for global diffuse illumination," *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 155-164, 1990.

[13] D.R. Baum, S. Mann, K.P. Smith, and J.M. Winget, "Making radiosity usable: automatic preprocessing and meshing techniques for the generation of accurate radiosity solutions," *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, 1991.

[14] P. Hanrahan, D. Salzman, and L. Aupperle, "A rapid hierarchical radiosity algorithm," *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, 1991.

[15] D. Lischinski, F. Tampieri, and D.P. Greenberg, "Combining hierarchical radiosity and discontinuity meshing," *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, 1993.

[16] F. Durand, G. Drettakis, and C. Puech, "Fast and accurate hierarchical radiosity using global visibility," *ACM Transactions on Graphics,* vol. 18, no. 2, pp. 128-170, 1999.

[17] B. Bolstad, *Field-of-view directed radiosity*, Master's thesis, Department of Computer Science, Texas A&M University, College Station, 1999.

[18] D.R. Baum, and J.M. Winget, "Real-time radiosity through parallel processing and hardware acceleration," *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, 1990.

[19] C. Puech, F. Sillion, and C. Vedel, "Improving interaction with radiosity-based lighting simulation programs," *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, pp. 51-57, 1990.

[20] R.J. Recker, D.W. George, and D.P. Greenberg, "Acceleration techniques for progressive refinement radiosity," *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, 1990.

[21] T.A. Funkhouser, "Coarse-grained parallelism for hierarchical radiosity using group iterative methods," *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, 1996.

[22] N.A. Carr, J.D. Hall, and J.C. Hart, "GPU algorithms for radiosity and subsurface scattering," *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 51-59, 2003.

[23] T.J. Purcell, I. Buck, W.R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," *ACM Transactions on Graphics,* vol. 21, no. 3, pp. 703-712, 2002.

[24] N.A. Carr, J.D. Hall, and J.C. Hart, "The ray engine," *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 37-46, 2002.

[25] R. Fernando, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[26] T.J. Purcell, C. Donner, M. Cammarano, H.W. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware," *ACM SIGGRAPH 2005 Courses*, pp. 258, 2005.

[27] Microsoft Corporation, "DirectX software development kit," http://msdn.microsoft.com/library/cc440756.aspx, 2010.

[28] Khronos Group, "OpenGL software development kit," http://www.opengl.org/sdk/, 2010.

[29] Khronos Group, "OpenGL platform & OS implementations," http://www.opengl.org/documentation/implementations/, 2010.

[30] M. Segal, and K. Akeley, *The OpenGL Graphics System: A Specification (Version 2.1).* Silicon Graphics, Inc., 2006.

[31] D. Shreiner, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1.* Addison-Wesley Professional, 2009.

[32] P. Brown, J. Leech, R. Mace, and B. Paul, "ARB_texture_float," *OpenGL Extension Registry*, http://www.opengl.org/registry/specs/ARB/texture_float.txt, 2008.

[33] K. Akeley, J. Allen, B. Beretta, P. Brown, M. Craighead, A. Eddy, C. Everitt, M. Galvan, M. Gold, E. Hart, et al., "EXT_framebuffer_object," *OpenGL Extension Registry*, http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt, 2008.

[34] P. Brown, J. Jones, J. Leech, R. Mace, V. Moya, and B. Paul, "ARB_color_buffer_float," *OpenGL Extension Registry*, http://www.opengl.org/registry/specs/ARB/color_buffer_float.txt, 2007.

[35] P. Brown, D. Ginsburg, M. Gold, M.J. Kilgard, J. Leech, B. Licea-Kane, B. Lichtenbelt, B. Lipchak, B. Paul, J. Rosasco, et al., "ARB_texture_rectangle," *OpenGL Extension Registry*, http://www.opengl.org/registry/specs/ARB/texture_rectangle.txt, 2005.

[36] M. Gold, and P. Brown, "EXT_texture_integer," *OpenGL Extension Registry*, http://www.opengl.org/registry/specs/EXT/texture_integer.txt, 2006.

[37] J. Leech, and M. Kilgard, "EXT_texture_array," *OpenGL Extension Registry*, http://www.opengl.org/registry/specs/EXT/texture_array.txt, 2008.

[38] B. Lichtenbelt, and P. Brown, "EXT_gpu_shader4," *OpenGL Extension Registry*, http://www.opengl.org/registry/specs/EXT/gpu_shader4.txt, 2009.

[39] D. Hudson, R. Loach, and R. Ridge, "The Tao framework," http://sourceforge.net/projects/taoframework/, 2010.

[40] L. Dupont, R. Ridge, and A. Ghezelbash, "C# graphics library," http://csgl.sourceforge.net/, 2010.

[41] Khronos Group, "OpenGL extension header file," http://www.opengl.org/registry/api/glext.h, 2010.

[42] R.J. Rost, B. Licea-Kane, D. Ginsburg, J.M. Kessenich, B. Lichtenbelt, H. Malan, and M. Weiblen, *OpenGL Shading Language*. Addison-Wesley Professional, 2009.

[43] R. Fernando, and M.J. Kilgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[44] M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt, "OpenMesh - a generic and efficient polygon mesh data structure," *OpenSG Symposium*, Darmstadt, Germany, 2002.

[45] B.G. Baumgart, "Winged edge polyhedron representation," Technical Report No. CS-320, Stanford University, 1972.

[46] Computer Graphics Group, RWTH Aachen, "OpenMesh software development kit," http://www.openmesh.org, 2010.

[47] Microsoft Corporation, "MFC reference," http://msdn.microsoft.com/en-us/library/d06h2x6e(VS.80).aspx, 2010.

[48] NVIDIA Corporation, "NVIDIA Quadro 5000," http://www.nvidia.com/object/product-quadro-5000-us.html, 2010.

[49] Realtech VR Group, "OpenGL extensions viewer," http://www.realtech-vr.com/glview/, 2010.

[50] Graphic Remedy Group, "gDEBugger - OpenGL, OpenGL ES and OpenCL debugger, profiler and memory analyzer," http://www.gremedy.com, 2010.

[51] Institut für Visualisierung ung Interaktive Systeme, Universität Stuttgart, "glslDevil - OpenGL GLSL debugger," http://www.vis.uni-stuttgart.de/glsldevil/, 2010.

[52] M. Cohen, D. Greenberg, D. Immel, and P. Brock, "An efficient radiosity approach for realistic image synthesis," *IEEE Computer Graphics and Applications,* vol. 6, no. 3, pp. 26-35, 1986.

[53] A. Appel, "Some techniques for shading machine renderings of solids," *Proceedings of the Spring Joint Computer Conference*, pp. 37-45, 1968.

[54] T. Whitted, "An improved illumination model for shaded display," *Communications of the ACM,* vol. 23, no. 6, pp. 343-349, 1980.

APPENDIX A

This section contains pseudo-code for the three GPU-based radiosity algorithms implemented during research.

Progressive Refinement

```
    // initialize surface textures
•   for each surface $j$
        o   draw $E_j$ to textures $B_j$ and $\Delta B_j$

•   while($true$)
            // select shooting surface
        o   enable $nextShooterShader$
        o   for each surface $j$ in environment
                ▪   draw $surfaceID$ of surface $j$ to texture $nextShooter$ with depth
                    $\Delta B_j A_j$
        o   $shootingSurface = nextShooter$

        o   for each $shootingPosition$ in $shootingSurface$
                    // create hemisphere item buffer
                ▪   enable $stereographicShader$
                ▪   for each surface $j$ in environment
                        ❖   set projection from $shootingPosition$
                        ❖   draw $surfaceID$ of surface $j$ to texture
                            $hemisphereItemBuffer$
                    // perform radiosity calculations
                ▪   enable $formFactorShader$
                ▪   for each surface $j$ in environment visible in hemisphere
                        ❖ draw textures $B_j$ and $\Delta B_j$
                        ❖ back-project each $fragment$ from $shootingPosition$
                          into $hemisphereItemBuffer$
                        ❖ if $fragment$ is visible
                            ➢ $B_j = B_j + \rho_j B_i F_{ij}$
                            ➢ $\Delta B_j = \Delta B_j + \rho_j B_i F_{ij}$

                    // compute ambient term
                ▪   enable $ambientShader$
                ▪   for each surface $j$ in environment
                        ❖ draw $F_{*j} \times \Delta B_j$ contribution to texture $ambient$

                    // display environment with ambient effects
                ▪   if $currentTime - lastDisplayTime > 1/FPS$
                        ❖ enable $displayShader$
                        ❖ for each surface $j$ in environment
                            ➢ $surfaceColor = B_j + (ambient \times \mathcal{R} \times \rho_j)$
```

Importance-Driven

Text in bold denotes code added to progressive refinement radiosity for this algorithm's implementation.

```
    // initialize surface textures
•   for each surface $j$
        o   draw $E_j$ to textures $B_j$ and $\Delta B_j$
        o   **draw 0 to textures $I_j$, $\Delta I_j$, and $R_j$**

•   while($true$)
        o   **if user view changes**
                **// create user view item buffer**
                ▪   **enable $userViewShader$**
                ▪   **for each surface $j$ in environment**
                        ❖   **set projection from user POV**
                        ❖   **draw $surfaceID$ of surface $j$ to texture $userViewItemBuffer$**

                **// account for changes in user POV**
                ▪   **enable $importanceShader$**
                ▪   **for each surface $j$ in environment visible from user POV**
                        ❖   **draw textures $I_j$, $\Delta I_j$, and $R_j$**
                        ❖   **back-project each $fragment$ from user POV into $userViewItemBuffer$**
                        ❖   **if $fragment$ is visible**
                                ➢   $R'_j = importanceEnergyConstant \times F_{ij}$
                                ➢   $\Delta R_j = R'_j - R_j$
                                ➢   $I_j = I_j + \Delta R_j$
```

```
            ➢ $\Delta I_j = \Delta I_j + \Delta R_j$
            ➢ $R_j = R'_j$

    // select importance shooting surface
o   enable $nextShooterShader$
o   for each surface $j$ in environment
        ▪ draw $surfaceID$ of surface $j$ to texture $nextShooter$ with
          depth $\Delta I_j A_j$
o   $shootingSurface$ = $nextShooter$

o   for each $shootingPosition$ in $shootingSurface$
          // create hemisphere item buffer
        ▪ enable $stereographicShader$
        ▪ for each surface $j$ in environment
            ❖ set projection from $shootingPosition$
            ❖ draw $surfaceID$ of surface $j$ to texture
              $hemisphereItemBuffer$

          // perform importance calculations
        ▪ enable $formFactorShader$
        ▪ for each surface $j$ in environment visible in hemisphere
            ❖ draw textures $I_j$ and $\Delta I_j$
            ❖ back-project each $fragment$ from
              $shootingPosition$ into $hemisphereItemBuffer$
            ❖ if $fragment$ is visible
                ➢ $I_j = I_j + \rho_j I_i F_{ij}$
                ➢ $\Delta I_j = \Delta I_j + \rho_j I_i F_{ij}$
```

```
// Select radiosity shooting surface
o enable $nextShooterShader$
o for each surface $j$ in environment
    ▪ **draw $surfaceID$ of surface $j$ to texture $nextShooter$ with depth $I_j \Delta B_j A_j$**
o $shootingSurface = nextShooter$

o for each $shootingPosition$ in $shootingSurface$
    // create hemisphere item buffer
    ▪ enable $stereographicShader$
    ▪ for each surface $j$ in environment
        ❖ set projection from $shootingPosition$
        ❖ draw $surfaceID$ of surface $j$ to texture $hemisphereItemBuffer$

    // perform radiosity calculations
    ▪ enable $formFactorShader$
    ▪ for each surface $j$ in environment visible in hemisphere
        ❖ draw textures $B_j$ and $\Delta B_j$
        ❖ back-project each $fragment$ from $shootingPosition$ into $hemisphereItemBuffer$
        ❖ if $fragment$ is visible
            ➢ $B_j = B_j + \rho_j B_i F_{ij}$
            ➢ $\Delta B_j = \Delta B_j + \rho_j B_i F_{ij}$

    // compute ambient term
    ▪ enable $ambientShader$
    ▪ for each surface $j$ in environment
        ❖ draw $F_{*j} \times \Delta B_j$ contribution to texture $ambient$
    // display environment with ambient effects
▪ if $currentTime - lastDisplayTime > 1/FPS$
    ❖ enable $displayShader$
    ❖ for each surface $j$ in environment
        ➢ $surfaceColor = B_j + (ambient \times \mathcal{R} \times \rho_j)$
```

<u>Perspective-Driven</u>

Text in bold denotes code added to importance-driven radiosity for this algorithm's implementation.

```
    // initialize surface textures
•   for each surface $j$
        o   draw $E_j$ to textures $B_j$ and $\Delta B_j$
        o   draw 0 to textures $I_j$, $\Delta I_j$, and $R_j$
    **// initialize perspective-driven parameters**
•   **draw 0 to texture $\mathbb{B}$**
•   $\boldsymbol{simulatedFOV = userFOV}$

•   while($true$)
        o   **if decay interval elapses**
                ▪   $\boldsymbol{simulatedFOV \mathrel{+}= 5}$

        o   if user view changes
                **// reset the simulated FOV**
                ▪   $\boldsymbol{simulatedFOV = userFOV}$

                // create user view item buffer
                ▪   enable $userViewShader$
                ▪   for each surface $j$ in environment
                        ❖   set projection from user POV
                        ❖   draw $surfaceID$ of surface $j$ to texture
                            $userViewItemBuffer$
```

```
            // account for changes in user POV
        ▪ enable $importanceShader$
        ▪ for each surface $j$ in environment visible in user POV
            ❖ draw textures $I_j$, $\Delta I_j$, and $R_j$
            ❖ back-project each $fragment$ from user POV into
              $userViewItemBuffer$
            ❖ if $fragment$ is visible
                ➢ $R'_j = importanceEnergyConstant \times F_{ij}$
                ➢ $\Delta R_j = R'_j - R_j$
                ➢ $I_j = I_j + \Delta R_j$
                ➢ $\Delta I_j = \Delta I_j + \Delta R_j$
                ➢ $R_j = R'_j$

    o if $simulatedFOV$ changes
            // draw environment using $simulatedFOV$
        ▪ for each surface $j$ in environment
            ❖ set projection from user POV using $simulatedFOV$
            ❖ initiate occlusion query $isVisibleFOV_j$
        ▪ stream each occlusion query $isVisibleFOV_j$ to texture
          $visiblityVectorFOV$

            // retrieve visible stored radiosity table entries to be shot
        ▪ enable $retrieveStoredRadiosityTableShader$
        ▪ draw texture $C$
            ❖ if value at texel $\mathbb{B}_{0j}$ indicates previously not visible and
              value at texel $visiblityVectorFOV_{0j}$ indicates
              currently visible
                ➢ draw $fragment_{ij}$ to texture $retrieved\mathbb{B}$
            ➢ draw 0 to texture $\mathbb{B}$
        ❖ else
            ➢ draw $fragment_{ij}$ to texture $\mathbb{B}$
            ➢ draw 0 to texture $retrieved\mathbb{B}$
        ❖ if shooting location $i$ has surfaces to shoot to
            ➢ draw $fragment_{io}$ to texture $retrieved\mathbb{B}$

    // distribute retrieved results to surfaces
▪   for each $shootingSurface$ with unshot radiosity
        ❖ retrieve $shootingPositionList$ from $retrieved\mathbb{B}$
        ❖ for each $shootingPosition$ in
          $shootingPositionList$
              // create hemisphere item buffer
            ➢ enable $stereographicShader$
            ➢ for each surface $j$ in environment
                ▪ set projection from $shootingPosition$
                ▪ draw $surfaceID$ of surface $j$ to texture
                  $hemisphereItemBuffer$

              // perform radiosity calculations
            ➢ enable $formFactorShader$
            ▪ for each surface $j$ in environment visible in
              hemisphere and becoming visible in user FOV
                ❖ draw textures $B_j$ and $\Delta B_j$
                ❖ back-project each $fragment$ from
                  $shootingPosition$ into
                  $hemisphereItemBuffer$
                ❖ if $fragment$ is visible
                    ➢ $B_j = B_j + \rho_j B_i F_{ij}$
                    ➢ $\boldsymbol{\Delta B_j = \Delta B_j + \rho_j B_i F_{ij}}$

// select importance shooting surface
o enable $nextShooterShader$
o for each surface $j$ in environment
    ▪ draw $surfaceID$ of surface $j$ to texture $nextShooter$ with depth $\Delta I_j A_j$
o $shootingSurface = nextShooter$

o for each $shootingPosition$ in $shootingSurface$
      // create hemisphere item buffer
    ▪ enable $stereographicShader$
    ▪ for each surface $j$ in environment
        ❖ set projection from $shootingPosition$
        ❖ draw $surfaceID$ of surface $j$ to texture $hemisphereItemBuffer$

      // perform importance calculations
    ▪ enable $formFactorShader$
    ▪ for each surface $j$ in environment visible in hemisphere
        ❖ draw textures $I_j$ and $\Delta I_j$
        ❖ back-project each $fragment$ from $shootingPosition$ into $hemisphereItemBuffer$
        ❖ if $fragment$ is visible
            ➢ $I_j = I_j + \rho_j I_i F_{ij}$
            ➢ $\Delta I_j = \Delta I_j + \rho_j I_i F_{ij}$

// Select radiosity shooting surface
o enable $nextShooterShader$
```

```
    o for each surface $j$ in environment
        ▪ draw $surfaceID$ of surface $j$ to texture $nextShooter$ with depth
          $I_j \Delta B_j A_j$
    o $shootingSurface$ = $nextShooter$
    o for each $shootingPosition$ in $shootingSurface$
          // create hemisphere item buffer
        ▪ enable $stereographicShader$
        ▪ for each surface $j$ in environment
            ❖ set projection from $shootingPosition$
            ❖ draw $surfaceID$ of surface $j$ to texture
              $hemisphereItemBuffer$

          // gather stored radiosity table entries
        ▪ enable $gatherStoredRadiosityTableShader$
        ▪ draw texture $\mathbb{B}$
            ❖ if $fragment$ lies in column $\mathbb{B}_i$
                ➢ write nearest neighbor value of texture
                  $shootingPositions$ at layer
                  $shootingSurfaceID$ to $\mathbb{B}_{i0}$
                ➢ increment $\mathbb{B}_{ij}$ if surface $j$ is not visible in user
                  FOV by value of radiosity $shootingSurface$ is
                  shooting
            ❖ if $fragment$ lies in column $\mathbb{B}_0$
                ➢ write value to $\mathbb{B}_{0j}$ indicating if surface $j$ is not
                  visible in user FOV

          // perform radiosity calculations
        ▪ enable $formFactorShader$
```

```
▪ for each surface $j$ in environment visible in hemisphere and visible in user FOV
    ❖ draw textures $B_j$ and $\Delta B_j$
    ❖ back-project each $fragment$ from $shootingPosition$ into $hemisphereItemBuffer$
    ❖ if $fragment$ is visible
        ➢ $B_j = B_j + \rho_j B_i F_{ij}$
        ➢ $\Delta B_j = \Delta B_j + \rho_j B_i F_{ij}$

  // compute ambient term
▪ enable $ambientShader$
▪ for each surface $j$ in environment
    ❖ draw $F_{*j} \times \Delta B_j$ contribution to texture $ambient$

  // display environment with ambient effects
▪ if $currentTime - lastDisplayTime > 1/FPS$
    ❖ enable $displayShader$
    ❖ for each surface $j$ in environment
        ➢ $surfaceColor = B_j + (ambient \times \mathcal{R} \times \rho_j)$
```

APPENDIX B

The following are permanent links to videos captured for the demonstration of research results. All videos are stored at 1080p resolution.

Video 1	Progressive refinement radiosity, single-enclosure environment. http://www.youtube.com/watch?v=EHv8iCGMvRw
Video 2	Progressive refinement radiosity, single-enclosure environment using ambient term. http://www.youtube.com/watch?v=zz5Awdejp-8
Video 3	Progressive Refinement radiosity, single-enclosure environment with exaggerated colors. http://www.youtube.com/watch?v=MNUz8OnhLnc
Video 4	Progressive refinement radiosity, single-enclosure environment with exaggerated colors using ambient term. http://www.youtube.com/watch?v=oPXQUe5ZxaA
Video 5	Progressive refinement radiosity, multiple-enclosure environment. http://www.youtube.com/watch?v=tS1S8MdJSQs
Video 6	Importance-driven radiosity, multiple-enclosure environment. http://www.youtube.com/watch?v=QRZS9oMc4WM
Video 7	Importance-driven radiosity, multiple-enclosure environment, with importance visualized as intensity values. http://www.youtube.com/watch?v=G9ndxohtqAo

Video 8	Progressive refinement radiosity, subdivided single-enclosure environment. http://www.youtube.com/watch?v=_Sr80g7KYVE
Video 9	Importance-driven radiosity, subdivided single-enclosure environment. http://www.youtube.com/watch?v=3PH-hbDnFQ4
Video 10	Perspective-driven radiosity, subdivided single-enclosure environment, ten-second decay constant. http://www.youtube.com/watch?v=6S-gG3c_zkE
Video 11	Perspective-driven radiosity, subdivided single-enclosure environment with exaggerated colors, one second decay constant. http://www.youtube.com/watch?v=-Bq1jJJf7bk
Video 12	Progressive refinement radiosity, subdivided multiple-enclosure environment. http://www.youtube.com/watch?v=OcMGUfk9Hng
Video 13	Importance-driven radiosity, subdivided multiple-enclosure environment. http://www.youtube.com/watch?v=jRkXk-uGB6Q
Video 14	Perspective-driven radiosity, subdivided multiple-enclosure environment. http://www.youtube.com/watch?v=Qj1YwdVlplY

VITA

Name:	Justin Taylor Bozalina
Address:	8584 Katy Freeway, Suite 400 Houston, TX 77024
Email Address:	justin@bozalina.com
Education:	B.S., Computer Science, Texas A&M University, 2004 M.S., Computer Science, Texas A&M University, 2011