High Fidelity Radiosity Rendering at Interactive Rates

by

Stephen Lincoln Hardt

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1996

Author . . . .
Department of Electrical Engineering and Computer Science
May 23, 1996

Certified by . . . .
Seth J. Teller
Assistant Professor
visor

Accepted by . . . .
Morgenthaler
Chairman, Departmental Committee on Graduate Theses

High Fidelity Radiosity Rendering at Interactive Rates

by

Stephen Lincoln Hardt

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1996, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Existing radiosity rendering algorithms achieve interactivity or high fidelity, but not both. Most radiosity renderers optimize interactivity by converting to a polygonal representation and Gouraud interpolating shading samples, thus sacrificing visual fidelity. A few renderers achieve improved fidelity by performing a per-pixel irradiance "gather" operation, much as in ray-tracing. This approach does not achieve interactive frame rates on existing hardware.

This thesis bridges the gap, by describing a data structure and algorithm which enable interactive, high-fidelity rendering of radiosity solutions. In essence, our algorithm "factors" the radiosity rendering computation into two components: an *offline* phase, in which a per-surface representation of irradiance is constructed; and an *online* phase, in which this representation is rapidly queried to produce a radiosity value at each pixel. The key components of the offline phase are a conservative discontinuity ranking algorithm, which identifies only the strongest discontinuities, and a hybrid quadtree-mesh data structure which prevents combinatorial interactions between most discontinuities. The online phase involves a novel use of perspective-correct texture-mapping hardware to produce nonlinear, analytic shading effects.

Thesis Supervisor: Seth J. Teller
Title: Assistant Professor

Acknowledgments

I thank my advisor, Prof. Seth Teller, for his advice and ideas throughout all stages of my research. Satyan Coorg gave me plenty of good information relevant to my research and about graphics in general. Thanks to my officemate, Rebecca Xiong, for looking over the draft of this thesis. And thanks to my other officemate, George T. Chou, for his career advice and heated discussions.

And, of course, I thank MIT for everything it has done for (to) me.
Thank you sir, may I have another?

Contents

List of Figures

Chapter 1

Introduction

1.1 Some Limitations of Polygon Hardware

Modern graphics workstations, although extraordinarily powerful, can be ill-suited for viewing global illumination solutions. Polygon hardware typically linearizes both geometry and shading, whereas the illumination function over a surface will in general have discontinuities along curved contours, and vary in a non-polynomial (and certainly non-linear) fashion even where it is smooth. Moreover, screen-space interpolation is not invariant under rotation, causing shading artifacts during interactive viewing.

Polygon-rendering hardware has been successfully used in interactive walkthroughs of globally illuminated environments [1, 5, 13]. In these interactively rendered sequences, however, the surface geometry is a collection of polygons, and the surface shading is a screen-space linear interpolation of a function whose value is specified at three points (typically, the vertices of a triangle) [12, 2]. Although higher-order geometry primitives exist on some architectures [7, 26], even these polygonalize, then Gouraud interpolate over, the interiors of the resulting triangles. Graphics hardware architectures that perform higher-order shading have been built [12, 21], but are not widely available.

These facts partially explain why many of the beautiful images published in the global illumination literature (e.g., [23]) are produced by ray-casting algorithms which, at each pixel, identify surface points to be shaded, then compute analytical irradiance values there with an object-space algorithm. Given the solution data, this rendering process can consume tens of seconds or minutes per image, depending on scene complexity.

1.2 Towards Accurate Interactive Display

We discuss the generation of accurate irradiance and radiosity values for a polyhedral scene rendered from an interactively-controlled viewpoint. Our implementation uses standard rendering hardware as a massively parallel query engine operating upon a large, object-space, parallel spatial data structure.

We assume that a hierarchical radiosity solution method is in use, which produces as output a discretization of the input into *elements* annotated by radiosity values, and an organization of interactions between elements into *links* annotated by *blockers* (as in [17, 32]). From interactions among the blockers and light sources, we rank irradiance discontinuities by their strength on the receiver, and select the strongest. The selected discontinuities partition each solution element into disjoint regions, inside each of which none of the (selected) discontinuities can be present. We then approximate the irradiance function inside each region using a polynomial *interpolant*, whose domain is the surface itself.

In an interactive session, a synthetic eyepoint moves through the scene under user control. In a novel use of rendering hardware (normally used to display perspective-correct textured polygons), the radiance data structure is queried in object space, at every pixel. Next, a host-parallel pass through the query structure generates the radiosity at each pixel from the relevant irradiance interpolant and the surface's reflectance and emittance. Our prototype implementation is the first to simultaneously capture global lighting effects and evaluate superlinear radiosity interpolants at interactive rates.

1.3 Algorithmic Foundations

Our algorithms and implementation build upon several ideas and techniques from the literature:

Hierarchical radiosity. We reimplemented the algorithms of [17, 14, 15] and use the hierarchical radiosity and shaft-culling techniques to find a diffuse energy distribution. We assume that the algorithm converges to an accurate radiosity solution, but do not consider convergence or solver accuracy issues here.

Irradiance data structure. We use the idea of a per-surface data structure which approximates spatially varying irradiance [33, 23, 24, 3].

Discontinuity identification. Our algorithms explicitly identify irradiance discontinuities, in order to improve the visual fidelity of the computed solution, as have [4, 18, 27, 24, 31, 10].

Hardware acceleration of object-space computations. As did the "hemicube" [8], "two-pass" [29], and "3D painting" [16] algorithms, we use fast graphics hardware to discretize and accelerate object space computations.

Backprojection of occluders. We use the notion of "backprojection" for the computation of accurate source-to-point irradiance in the presence of occlusion [10].

This thesis introduces several new ideas and techniques, among them:

Discontinuity ordering. We give an algorithm for selection from a collection of irradiance discontinuities via a heuristic estimate of their relative strengths at the receiver. Although our irradiance gradient is heuristic, it is less computationally intricate than those in [20, 33, 3]. Moreover, both discontinuities caused by emitters and reflectors are handled.

Hybrid mesh structures. Quadtrees are fundamentally unable to model general domains, except by a sort of generalized (and aliasing-prone) binary subdivision. We show that a hybrid of quadtree and explicit meshing yields a meshing scheme more flexible than quadtrees, yet more efficient than explicit meshing.

Hardware acceleration of irradiance queries. We describe a novel use of the polygon-rendering and texture-mapping capabilities of a high-end graphics workstation to generate real-time irradiance queries, in parallel, for all pixels of an image. Our approach avoids both direct rendering of polygonalized elements, and the use of screen-space interpolation (i.e., Gouraud shading).

1.4 Notation

Actual C++ classes in the implementation will be capitalized in typewriter print, e.g. `Discontinuity`, while references to a concept represented by the C++ class will be in lower–case regular print, e.g. discontinuity. With a slight abuse of notation we use the plural of a class name, e.g. `Discontinuities`, to mean multiple instances of the class, e.g. several `Discontinuity` objects.

1.5 Organization

The remainder of this thesis is organized as follows: Chapter 2 gives a brief description of the global illumination problem and of the radiosity method. Chapter 3 describes the high-speed high-fidelity rendering algorithm as an abstract algorithm requiring three components. Chapters 4, 5, and 6 describe our implementation of the three components. Chapter 7 gives the results of running the system on a scene of medium complexity. Chapter 8 discusses limitations and future work. Appendix A gives a list of all adjustable parameters in the system. Appendix B describes the user interface and visualization tools, acting as a simplified instruction manual for the system.

All of this material can be found online at `http://graphics.lcs.mit.edu/~hardts`.

Chapter 2

The Radiosity Method

2.1 Global Illumination

Shading surfaces according to some lighting model is a fundamental problem in computer graphics. Illumination models can be divided into two classes, *local* and *global.* Local models compute shading based only on the position of the surface, light, and eye. Modern graphics workstations often provide hardware support for shading according to some local model. However, local models ignore illumination due to light reflected from other surfaces, as well as other global effects such as the shadowing of one surface by another. Global illumination models, on the other hand, generate images of greater realism by taking into account the interreflection of light between surfaces, i.e. light can make several "bounces" before arriving at a receiver [9].

2.2 The Radiosity Method

2.2.1 Description

The *radiosity* method for solving the global illumination problem models each surface as a Lambertian diffuse reflector, i.e. light reflects from the surface equally in all directions no matter what the distribution of incoming light. Each surface has an emittance E, the amount of energy generated by the surface itself, and a reflectance

ρ, the percentage of energy hitting the surface that is reflected by the surface. Given the scene geometry and values of E and ρ for each surface, the radiosity method solves for the radiosity B over each surface. Radiosity is the energy per unit area (W/m^2) leaving a surface, and is ultimately the quantity used to color each pixel in a radiosity rendered image. We mention here one other quantity of interest. Irradiance is the energy per unit area (W/m^2) impinging on a surface. Radiosity and irradiance are very closely related, radiosity = irradiance * reflectance + emittance. So, if one of radiosity and irradiance is known, the other can easily be computed. For the remainder of this paper we will often omit the explicit conversion between radiosity and irradiance.

The radiosity solution depends fundamentally on an energy balance equation which is approximated and solved by numerical means. We will call this the *Radiosity Equation.* Specifically, for each element i of a set of n elements:

$$B_i = E_i + \rho_i \sum_{j=1}^{n} B_j F_{ij} \tag{2.1}$$

where E_i, ρ_i, and B_i are the emittance, reflectance, and radiosity of element i, respectively. F_{ij} is the form factor from element i to element j. This represents the geometric relationship between element i and j and gives the amount to which the radiosity of j affects the irradiance of i. Specifically, F_{ij} is the "fraction of energy that leaves element i and arrives directly at element j" [9].

In general, these n elements are some *refinement* of the input geometry to improve the resolution of the radiosity solution. The refinement of input surfaces into smaller elements often occurs concurrently with the computation of the radiosity solution, producing greater subdivision where the radiosity function changes more rapidly.

2.2.2 View Independence

Radiosity algorithms are called *view-independent* solutions, since they operate in 3D object space, independent of the eye position from which the scene will eventually be rendered. This is as opposed to *view-dependent* solutions such as ray tracing, where the solution starts with the eye position and shoots rays back from pixel locations to compute the energy impinging on the eye along those rays. After creating an image from one viewpoint via a view-dependent method, work must be started again from scratch to create an image from a different viewpoint.

2.2.3 Radiosity Rendering

The view-independence of the radiosity solution allows subsequent rendering operations to be split into two phases. In the first *solution* phase, a radiosity solution is computed independently of the eye location. In the second *continuous rendering* phase, an eyepoint is specified and the radiosity solution is used to render an image from that eyepoint. The solution phase can be done once, offline, and may take a significant amount of time to compute. The rendering phase can be done many times, reusing the same radiosity solution. An interactive radiosity walkthrough system can be implemented if the rendering phase is performed sufficiently fast.

Most existing radiosity systems take one of two approaches to the rendering phase:

(1) Exact radiosity values are computed offline for the vertices of each element and then the elements are rendered online using hardware Gouraud shading.

(2) After the viewpoint is specified, rays are shot back from every pixel and an exact radiosity value is computed where the ray intersects the scene geometry.

Both methods require a means of computing an accurate radiosity value at a point in the scene. This is achieved by recomputing the form factors to the point from every surface visible to the point and then gathering the energy from all these surfaces. Method (1) makes this expensive computation in the solution phase, allowing it to achieve interactive rates in the rendering phase. However, A number of undesirable visual artifacts such as mach bands and color "swimming" can result from Gouraud shading, detracting from the image fidelity [11]. Method (2) produces beautiful, accurate radiosity images without shading artifacts, but the expensive per-pixel radiosity calculation in the rendering phase prevents interactive rates.

2.2.4 Strengths and Limitations of the Radiosity Method

The radiosity method[1] models only Lambertian diffuse reflection. This accurately captures diffuse, eye-independent, lighting effects such as soft shadows (shadows with regions of penumbra and umbra), and smooth falloff of light across a surface. However, it does not account for eye-dependent effects such as specular highlights and mirror reflection. The radiosity method does not model absorption of light by the transmitting media, such as with smoke or fog. It can model translucency with a simple extension, but not full transmission effects as they require information about the eye location. Radiosity systems only handle static scenes, since the view-independent solution is only valid for the geometric configuration specified when the solution is computed.

[1] Here we refer to the straight–forward radiosity method described in this paper. There are extensions to radiosity that handle some of the described phenomena with various trade–offs [9].

Chapter 3

High Fidelity Rendering

3.1 Requirements

This chapter describes our interpolant data structure, construction scheme, and rendering algorithm. This assumes that we have:

1. a set of quadtrees produced by an HR algorithm;
2. an algorithm for triangulating quadtree leaves according to the strongest discontinuities impinging on them; and
3. a function `IrradianceAtPoint()` which computes the irradiance at any source point due to all receivers irradiating that point.

This chapter shows how to generate a data structure which accurately represents irradiance, and how to use this structure for rapid rendering. Solutions for 1, 2, and 3 follow in Chapters 4, 5, and 6, respectively.

3.2 Data Structure

The radiance data structure is a list of triangles, each annotated with a quadratic *interpolant* for irradiance (Figure 3-1); an expression in s and t which smoothly approximates irradiance over the domain region. We use quadratic interpolants, as we have found that constant and linear interpolants are inadequate to capture the radiosity function faithfully, even in regions where it varies smoothly, and that higher order interpolants do not significantly improve the interpolant fit.

A quadtree is generated via requirement 1 and triangulated via requirement 2. These triangles are then assembled into a list, and an interpolant constructed for each triangle (Figure 3-2).

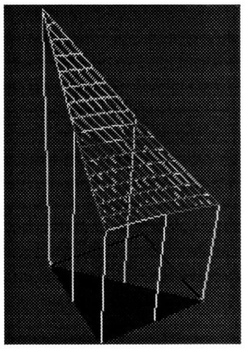

Figure 3-1: One triangle's interpolant (dark grey graph) from samples (white sticks) of analytic irradiance (light grey graph).

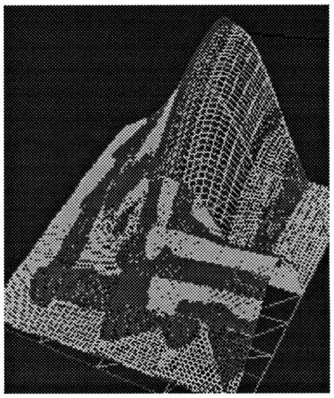

Figure 3-2: Contiguous triangle interpolants on a quadtree.

3.3 Constructing Interpolants

For each triangle `IrradianceAtPoint()`, requirement 3, is invoked at the triangle vertices and edge midpoints, to collect six irradiance values r_i. Using these six values, we construct an irradiance interpolant over the entire triangle, as a function of barycentric coordinates (s, t) over the triangle.

Given six barycentric sample locations $p_i = (s_i, t_i)$ and corresponding values r_i, the interpolant construction must determine coefficients $A...F$ of the function

$$R(s,t) = As^2 + 2Bst + 2Cs + Dt^2 + 2Et + F \tag{3.1}$$

so that

$$R(s_i, t_i) = r_i, \qquad 0 \le i < 6.$$

In general, this requires the solution of the quadratic form

$$\begin{pmatrix} s_i & t_i & 1 \end{pmatrix} \begin{pmatrix} A & B & C \\ B & D & E \\ C & E & F \end{pmatrix} \begin{pmatrix} s_i \\ t_i \\ 1 \end{pmatrix} = r_i, \qquad 0 \le i < 6.$$

However, judicious choice of a barycentric coordinate system and sample locations (the triangle vertices and edge midpoints) reduces the problem to solving a system of six linear equations. We write

$$\begin{pmatrix} R(0,0) \\ R(\frac{1}{2},0) \\ R(1,0) \\ R(\frac{1}{2},\frac{1}{2}) \\ R(0,1) \\ R(0,\frac{1}{2}) \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ \frac{1}{4} & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 2 & 0 & 0 & 1 \\ \frac{1}{4} & \frac{1}{2} & 1 & \frac{1}{4} & 1 & 1 \\ 0 & 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & \frac{1}{4} & 1 & 1 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \\ D \\ E \\ F \end{pmatrix} = \begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{pmatrix}.$$

Inverting the 6 × 6 matrix symbolically and multiplying by the sample vector yields the closed form solution for the coefficients:

$$\begin{pmatrix} A \\ B \\ C \\ D \\ E \\ F \end{pmatrix} = \begin{pmatrix} 2r_0 - 4r_1 + 2r_2 \\ 2r_0 - 2r_1 + 2r_3 - 2r_5 \\ -\frac{3}{2}r_0 + 2r_1 - \frac{1}{2}r_2 \\ 2r_0 + 2r_4 - 4r_5 \\ -\frac{3}{2}r_0 - \frac{1}{2}r_4 + r_5 \\ r_0 \end{pmatrix}.$$

The six resulting floating point numbers $(A \;\; 2B \;\; 2C \;\; D \;\; 2E \;\; F)$ are then stored as the interpolant for the triangle in question. (Storing $2B$, $2C$, and $2E$, rather than B, C, and E, avoids three multiplies during subsequent evaluation.)

3.4 Rendering

We now have a list L of triangles, each annotated with an irradiance interpolant expressed in terms of triangle barycentric coordinates. Our goal is to assign each pixel P in the output image the radiosity value for that point B (on the visible triangle T) that projects to P. We do so with a multi-pass algorithm on a graphics workstation.

To generate each frame of an interactively rendered sequence, we:

(a) Generate an image $\mathcal{I}_1$ in which every pixel P contains an encoding of which triangle T is visible at P;

(b) Generate an image $\mathcal{I}_2$ in which every pixel P contains an encoding of the barycentric coordinates B of the object-space point that projects to P;

(c) Using $\mathcal{I}_1$ and $\mathcal{I}_2$, generate the output image $\mathcal{O}$ by looping over all pixels P and evaluating T's irradiance interpolant at the object-space point corresponding to each P;

(d) Copy $\mathcal{O}$ to the framebuffer and swap it forward for display.

3.4.1 Visible Triangle Determination

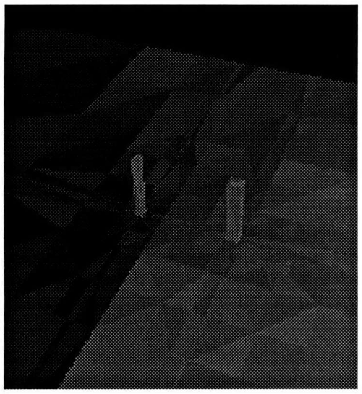

Figure 3-3: $\mathcal{I}_1$, Triangle identifiers.

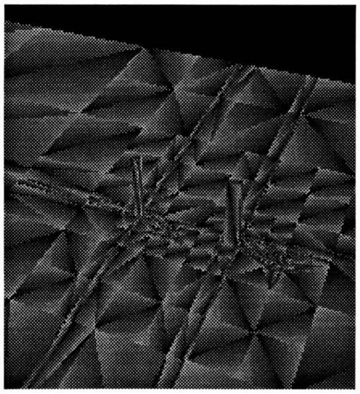

Figure 3-4: $\mathcal{I}_2$, Barycentric coordinates.

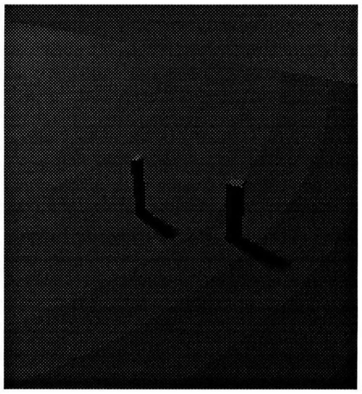

Figure 3-5: $\mathcal{O}$, Interpolant rendering.

Task **(a)**, visible triangle determination, is accomplished with the polygon fill and depth-buffering hardware. Each triangle T is rendered as a solid "color", the "color" being a 24-bit encoding of the index of T in the list of triangles L. This rendering is done in the hardware backbuffer to avoid erasing the previous image currently displayed in the frontbuffer. Occlusion is handled properly by the depth-buffering hardware. After all triangles have been rendered, the framebuffer is copied to host memory to create $\mathcal{I}_1$ (Figure 3-3).

3.4.2 Barycentric Coordinate Determination

Task **(b)** is accomplished with a $256 \times 256 \times 32$ bit texture map. This texture consists simply of an encoding of s and t at texel (s, t). We render each triangle, issuing it with (floating-point) texture coordinates $(0,0)$, $(1,0)$, and $(0,1)$. The texture-mapping hardware deposits, at each pixel, the correct texel and therefore the barycentric coordinates of the point to be shaded. Note that Gouraud-interpolation hardware is ill-suited for object-space interpolation, since it interpolates in screen space. However, perspective-correct texture mapping hardware is widely available, and interpolates in object space. Again, rendering is done in the backbuffer and occlusion is handled properly by the depth-buffering hardware. Finally, after all triangles have been rendered, the framebuffer is copied to host memory to create $\mathcal{I}_2$ (Figure 3-4).

3.4.3 Evaluating the Interpolant and Radiosity

To complete task **(c)**, we loop over every pixel P in the output image O. The corresponding pixels in the scratch images $\mathcal{I}_1$ and $\mathcal{I}_2$ give the visible triangle T (and its interpolant) at P and the barycentric coordinates B with respect to T of the point that projects to P, respectively. T's interpolant is evaluated at B to produce irradiance, which is then multiplied by the reflectivity of T and summed with $T's$ emissivity to produce radiosity.

Note that, given (s,t), evaluating

$$R(s,t) = As^2 + 2Bst + 2Cs + Dt^2 + 2Et + F$$

requires only eight multiplies and five adds (the factors of two are folded into the stored coefficients).

3.4.4 Depositing the Rendered Pixels

The final step, part **(d)**, simply copies $\mathcal{O}$ to the display backbuffer and swaps front and back buffers for an instantaneous update (Figure 3-5).

3.5 Costs and Parallelism

Steps **(a)** and **(b)** consist of flat-shaded polygon rendering, texture mapped polygon rendering, and copying pixels from the framebuffer to host memory. Step **(d)** consists of copying pixels from host memory to the framebuffer. These operations are all extremely fast (i.e. can easily be performed at interactive rates) on a high-end graphics workstation such as the Reality Engine [2].

The bottleneck of this algorithm is step **(c)**, taking 10 – 100 times as long as the other steps. Fortunately, this operation is highly parallelizable. The color of each pixel depends only on the triangle list and the scratch images $\mathcal{I}_1$ and $\mathcal{I}_2$. Since the color of every pixel in the output image is independent of the color of every other pixel, there is no data dependency problem. The time to evaluate the radiosity for any pixel is constant, so there is no load balancing problem.

On our host-parallel system, we create a separate evaluation process for each of the N physical processors and partition all pixels on the screen equally among them. Our implementation uses four processors (RISC R4400s running at 250 MHz), shared memory for communication between processes, and a system of locks to synchronize the processes.

3.6 T-vertices and Pixel Dropout

Our rendering hardware, a Silicon Graphics Reality Engine, guarantees that when two adjacent polygons that share vertices are rendered, every pixel along the common edge will be painted by exactly one of the two polygons.[1] However, when non-overlapping[2] polygons on either side of the edge do not have the exact same vertices, some pixels along the edge may be colored by both, or neither, of the polygons. We are concerned with a specific case of this, called a *t-vertex*. A t-vertex arises when an edge is shared by polygons that do not have the same vertices along that edge. In Figure 3-6 triangles t_1, t_2, and t_3 all have point a as a vertex, while the triangle t_4 on the other side of the edge does not.

If we restrict all quadtrees such that no neighboring nodes differ in depth by more than one, t-vertices occur in our system in only three cases.

A Adjacent quadtree nodes differ in depth by one. In Figure 3-6, there will be a t-vertex at a when q_1, q_2, and q_3 are triangulated.

B Neighboring quadtrees are triangulated such that one includes a discontinuity that crosses the common edge and the other does not.

C Quadtrees of adjacent top-level quadrilaterals do not have the same vertices along the shared edge (Figure 3-6, point c).

We detect case **A** by looking at the quadtree topology, and then employ a standard solution for t-vertex elimination. In Figure 3-6, the larger triangle t_4 is rendered as a quadrilateral, with an extra vertex at a. The polygons on both sides of the edge have the same vertices along the edge, so the requirements for the hardware to eliminate t-vertices are met.

We prevent case **B** from ever occurring by ensuring that if two quadtree nodes are adjacent, and a discontinuity from one node's triangulation crosses into the other

[1] In implementing this, we found a bug in the Reality Engine scan conversion hardware. This guarantee is not always met when zero-area triangles are rendered with backface culling turned on. We worked around this by doing all backface culling ourselves in software.

[2] By non-overlapping, we mean two polygons share, at most, an edge or vertex.

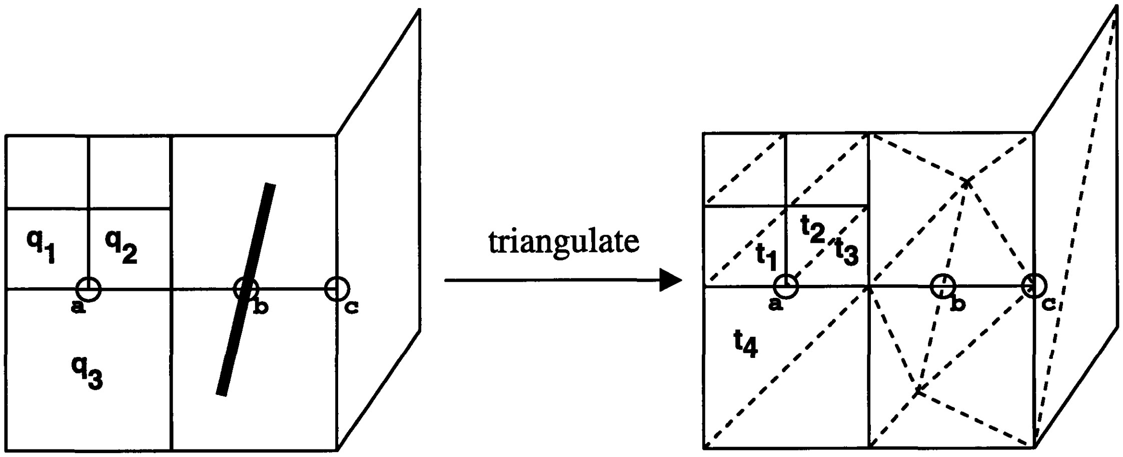

Figure 3-6: A quadtree node impinged upon by a discontinuity (dark line) on the left, with its triangulation on the right. The t-vertex at **a** is handled explicitly by rendering t_4 with an extra vertex at a. A t-vertex is prevented at b by ensuring that both leaves impinging on the discontinuity are triangulated according to the discontinuity. A t-vertex arises at **c** from separate top-level quadrilaterals.

node, then the other node will also have that discontinuity in its triangulation (Figure 3-6, point b). This is described in greater detail in Section 5.5.

We do not handle t-vertices arising from **C**, as they are less noticeable in the output images than those arising from **A** and **B**. T-vertices from **C** only occur on the boundary between different pieces of top-level geometry, whereas those from **A** or **B** can cause pixels to be lost in polygon interiors. If desired, type **C** t-vertices could be eliminated by incorporating inter-quadrilateral adjacency information into the quadtree leaf triangulation. Triangle edges would be marked with the locations of quadtree vertices of adjacent quadrilaterals that touch that triangle. Triangles would be rendered as a n-sided polygons, the triangle's three vertices plus extra vertices along the edges.

Chapter 4

Radiosity Solution

The underlying radiosity solver is a reimplementation of the hierarchical radiosity and wavelet radiosity algorithms described in [17, 14, 32]. The original code was developed in C++ and IrisGL by Seth Teller and Peter Schröder at Princeton. We ported the solver to OpenGL and modified it for our use.

Solving the global pass of the radiosity solution provides us with

- A quadtree subdivision of the input geometry;
- A radiosity value for each quadtree node; and
- A "link" representing each element-element interaction [32];
- A set of "blockers" for each link; That is, a set containing all possible elements that can occlude part of one element as seen from the other [32].

This computed solution satisfies the first requirement for the high fidelity rendering algorithm; that is, it is a suitable input for the interpolant construction (Section 3.1).

4.1 Representation

The bulk of the radiosity solver is implemented as the C++ classes `Quadrilateral`, `Tree`, `Link`, and `Tube`. The input geometry, a set of quadrilaterals, is represented with `Quadrilaterals`. Associated with each `Quadrilateral` is the root of a quadtree, a

`Tree` data structure. A quadtree is partition of two-dimensional space where each node is either a leaf or has four quadrilateral children. Each `Tree` node has a radiosity value represented as the amplitude of a constant basis function. Radiosity coefficients for nodes in the same quadtree are related such that a parent's coefficient is always the average of its children's values.

The `Link` structure represents energy interactions between quadtree nodes. For any two mutually visible points p and q on quadtrees P and Q, there will be exactly one `Link` between a node in P containing p and a node in Q containing q. In addition, a `Link` between a source j and receiver i contains the form factor F_{ij}.

The initial configuration of the solver comes from associating a `Tree` of zero depth with each input `Quadrilateral`. All possible interactions between pairs of `Quadrilaterals` are computed and a `Link` object annotated with a potential blocker list, `Tube`, is constructed for each [32]. Currently, this step is quite inefficient, requiring $O(n^3)$ time for n input quadrilaterals. For each of the n^2 pairs, check every one of the n `Quadrilaterals` to see if it is a potential blocker. This time could be greatly reduced by partioning the space with an octree and, for each pair, only considering quadrilaterals in the octree nodes overlapping the convex hull of the pair.

4.2 Element to Element Form Factors

In general, computing element to element form factors is one of the biggest problems in building a radiosity solver [9]. We experimented with three different methods:

1. Exact polygon to polygon form factor computation using Peter Schroeder's libff library [28].

2. Simple disk approximation [9].

3. Take the average of the point-to-polygon form factor computations for several points chosen randomly on the receiver. Use the same code and method described in Section 6.2.

In all three cases we clipped each of the source and receiver polygons to the plane of the other to deal with horizon effects, i.e. clip away the portion of one that cannot be visible to the front face of the other. By form factor algebra, when the source is split by the receiver plane we just clip away the invisible portion of the source. When the receiver is split by the source, we also clip away the portion of the receiver invisible to the source, but we must also adjust the resulting form factor by the ratio of the area of the clipped receiver to the area of the unclipped receiver [9].

We found cases in which both 1 and 2 gave results off by at least an order of magnitude. We chose method 3 because it gave the most stable results.

4.3 Iterative Solution

The iterative solution of the hierarchical radiosity system works as follows.

1. Each quadtree element's radiositiy coefficient is set to the quadtree's emittance.
2. A new set of radiosity values B_i^{new} is derived from the previous set B_i^{old} using the Radiosity equation (2.1). For all i, compute:

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

3. The quadtree representation of the radiosity function is refined (Section 4.4).
4. Steps 2 and 3 are repeated until the system converges, i.e. all radiosity values undergo a relative change less than a user specified amount, (Appendix A, 7).

4.4 Solution Refinement

The question now is how to "drive" the refinement process during the radiosity solution computation. After each pass "gathers" energy to elements in the quadtree, we refine the solution mesh by subdividing quadtree nodes and by splitting single links connecting nodes high in quadtrees into multiple links connecting nodes lower in these quadtrees [17]. We implemented two methods for choosing when to do this subdivision. The first is the standard method used by many existing hierarchical radiosity solvers, based on an estimate of error in the transport of energy across a link. The second is a new method which uses a more accurate and intuitive error metric based on the piece-wise quadratic representation of the radiosity function.

4.4.1 Transport-Based Refinement

Description

Traditional refinement schemes use a simple estimate of the error in transporting energy across a link. This error is relative to the change in the point-to-polygon form factor for different points on the link's receiver and to the source radiosity [17].

Refinement Step

For every `Link` in the system, if the the error estimate across it is greater than the specified error tolerance, `Gur::eps` (Appendix A, 1), then the link is split in one of two ways. The source quadtree node is subdivided and the old link is replaced by four new links from the new children of the source to the unchanged receiver. Or, the receiver quadtree is subdivided and the old link is replaced by four new links from the unchanged source to the new children of the receiver. Both will tend to reduce the error in energy transport across the link by reducing the size of the elements involved in the interaction. The choice is made based on the relative areas of source and receiver. To force interactions to be between elements of approximately equal size, thus minimizing error, the larger quadtree node is the one subdivided.

4.4.2 Representation-Based Refinement

Description

The transport-based error criteria is extremely conservative, and can do far more work on a surface than is required to capture the irradiance there faithfully (consider hundreds of strong sources arranged so as to produce nearly constant irradiance; a transport-based HR algorithm would subdivide to great depth to perform each source-receiver transport accurately). Clustering techniques such as [30] ameliorate this disadvantage somewhat, but still do not drive subdivision based on representation error, an estimate of the error between represented radiosity and that due to all sources irradiating the receiver.

Our representation of the radiosity function is a set of quadratic interpolants (Section 3.2). The error in an interpolant's fit can be accurately estimated by sampling using `IrradianceAtPoint()` (see Chapter 6) points in the triangle domain of the interpolant. For each point, compare the value given by `IrradianceAtPoint()` with that given by evaluating the interpolant. The representation error of a leaf is then the maximum error in interpolant fit over all interpolants on that leaf.

Refinement Step

Loop over all leaves of the quadtrees. For each:

1. Triangulate and build interpolants;

2. Compute the representation error; and

3. Subdivide the leaf if it does not meet the globally specified error bound, `Gur::eps` (Appendix A).

Advantages and Disadvantages

In practice this produces a coarser mesh than transport-based refinement does for solutions that give output images of equivalent quality. However, there are currently two significant drawbacks to this method.

1. It requires significantly more time to run, as interpolants must be constructed at every level of the quadtree. I.e. a quadtree node is only subdivided after interpolants have been built on that node. Transport-based refinement only requires interpolants to be built on the leaves of the tree for actual rendering.

Interpolant building, requiring several calls to `IrradianceAtPoint()`, is undoubtedly the bottleneck of the offline phase of the radiosity rendering system.

2. Representation-based refinement demands much greater accuracy and reliability from the `IrradianceAtPoint()` function. If the `IrradianceAtPoint()` function suffers from geometric aliasing or roundoff errors, the function it describes may have discontinuities in value where the actual irradiance function does not. Since these artificial discontinuities are not represented in the mesh, interpolants built across these discontinuities will generally fail to meet the error bound. In some cases, no matter how much the system refines the quadtree solution near such areas, the interpolant there will still fail to meet the error bounds. The refinement process gets stuck, refining elements down to the minimum allowable size in places where it is not truly necessary to do so.

The representation-based refinement scheme currently works well in practice for small input scenes, but does not scale well due to the cost of building interpolants. For larger scenes, such as in Figure 7-1, we used the transport-based refinement scheme where interpolants are only built once, on the leaves of the final quadtrees.

Chapter 5

Dynamic Discontinuity Meshing

5.1 Motivation

The irradiance function has discontinuities due to contact and occlusion. Since smooth interpolants do not perform well in the presence of discontinuities, researchers have proposed the construction of "discontinuity meshes," in which the solution elements (i.e., function domains) are explicitly meshed, in order to introduce boundary curves wherever discontinuities are detected [4, 23, 19, 24, 27, 31, 3, 10]. Once discontinuities have been banished from the interior of the element, a smooth interpolant can be fit, although for non-trivial domains this may require some fairly complex geometric and topological infrastructure [25, 27, 23, 22].

5.2 Quadtree with Discontinuity-Meshed Leaves

We assume, as in [17], that from every quadtree solution element all sources of irradiance there may be found, and that relevant blockers are associated with all source-receiver links. For each receiver element and its links, we identify the curves of irradiance discontinuity on the element. This is done by considering all edge-edge (EE) and vertex-edge (VE) pairs drawn from among the light source and blockers, as in [19, 24]. We also check all sources and receivers for horizon or contact discontinuities. The left two images in Figure 5-4 show VE discontinuities, Figure 5-3 shows an EV

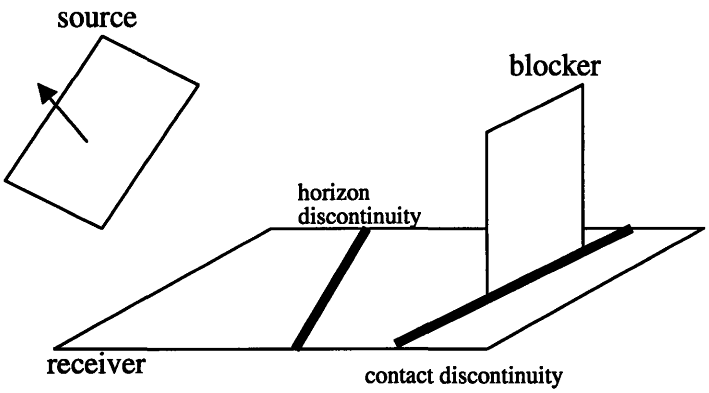

Figure 5-1: Horizon discontinuity caused by source plane splitting receiver and contact discontinuity caused by blocker touching receiver.

discontinuity, and Figure 5-1 shows a horizon and a contact discontinuity. Currently, we ignore triple-edge (EEE) critical surfaces as these have a generally weak visual effect. [19].

A general quadtree element, attempting to capture irradiance due to a multi-sided light source shining past some number of blockers, may intersect many discontinuity surfaces. However, quadtree subdivision of a node will tend to reduce the number of discontinuities impinging on the node's children.

For each quadtree leaf, we *rank* the discontinuities affecting the element and select the `Tree::maxLeafDisconts` strongest discontinuities with weight at least `Tree::minDiscontWeight`. `Tree::maxLeafDisconts` and `Tree::minDiscontWeight` are parameters to the algorithm (Appendix A). The minimum weight criteria avoids expending computational effort meshing solution elements which are impinged upon only by weak discontinuities. These discontinuity segments form the input to a *CDT*, constrained Delaunay triangulation, algorithm [23], which produces a triangulation containing the segments, along with adjacency information about the triangles. This triangulation satisfies the second requirement for the high fidelity rendering algorithm (Section 3.1).

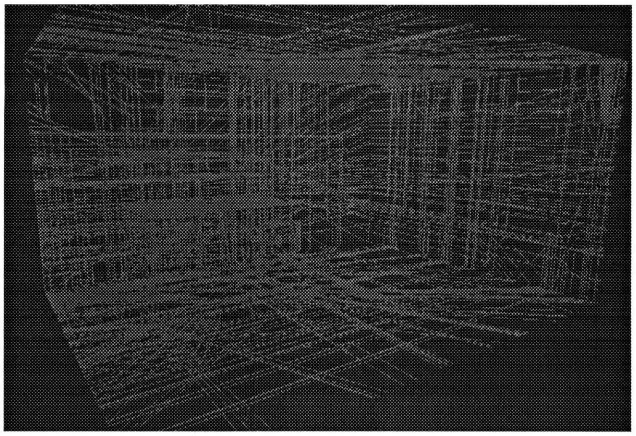

Figure 5-2: All the discontinuities in the office scene of Figures 7-1 and 7-2.

5.3 Discontinuity Ranking

Geometric interactions (horizons and occlusion) tend to produce an enormous number of discontinuity surfaces in a typical scene, and many of these surfaces will intersect a typical receiver surface (Figure 5-2). However, most of the geometric discontinuities will be quite weak radiometrically; consequently, they will have little or no visually discernible effect. We propose a method for ranking discontinuities by a heuristic "weight". The weight of a discontinuity produced by a specific source, blocker, receiver combination is derived by estimating the change that the discontinuity can cause in the radiosity function at the receiver.

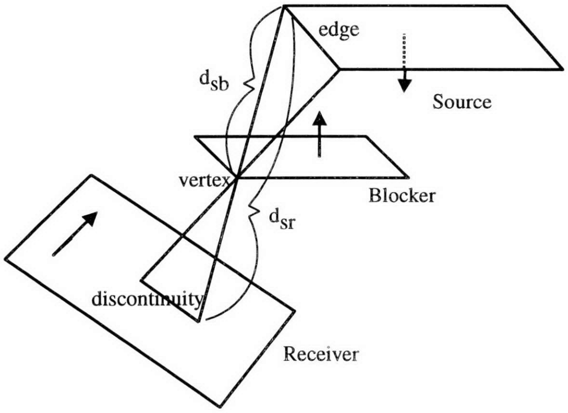

Figure 5-3: An EV(edge-vertex) discontinuity. d_{sb} is the distance from source to blocker. d_{sr} is the distance from source to receiver.

Define weight w for a VE or EV source-blocker-receiver combination. B_{src} is the radiosity of the source. d_{sb} and d_{sr} are the distances along a line in the VE or EV swath from the source to blocker and from the source to receiver, respectively. Figure 5-3 shows the configuration for an EV discontinuity. VE discontinuities are handled similarly.

$$w = \max\left(\frac{d_{sb}}{d_{br}}\right) \cdot B_{src}$$

This weight can viewed as the product of a purely geometric and a purely radiometric factor. The geometric factor $\max\left(\frac{d_{sb}}{d_{br}}\right)$ gives a measure of how close the blocker is to the receiver. The closer the blocker is to the receiver, the faster the source becomes visible (or obscured) as seen from a point moving from one side to the other across the discontinuity (Figure 5-3). The radiometric factor is B_{src}. The brighter the source, the stronger the discontinuity. If you take a second to look at the room around you, you will probably notice that all the shadows come only from the brightest or smallest light sources.

Horizon discontinuitiy weights are computed similarly, except with the geometric weight set to 1, the upper bound. In almost all cases, contact discontinuities give a strong visible effect, so they are treated specially and given “infinite” weight.

5.4 Discontinuity Weight Caching

`Discontinuity` structures, defined below, cache the discontinuity weight so that it is not recomputed every time it is needed.[1] The geometric factor of a weight never changes, but, since element radiosity values can change between iterations of the radiosity algorithm, the radiometric factor may become invalid. We handle this by explicitly invalidating cached radiometric weights in the radiosity algorithm, see Section 5.5.8.

[1] With such an inexpensive weight metric, it is a bit of overkill to worry too much about caching. However, we have been experimenting with more expensive and accurate weights for which caching does give a significant speedup (Section 8.2). We implemented a framework more general than what is strictly necessary for the weight metric described in this thesis.

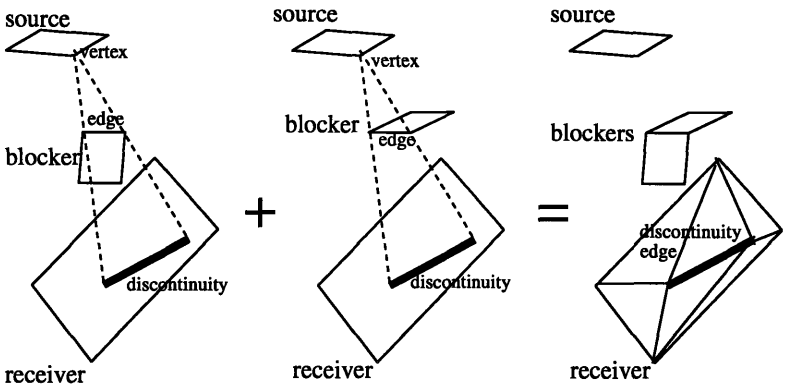

Figure 5-4: Two separate VE(vertex-edge) source-blocker-receiver combinations create two collinear (in this case, the same) discontinuities that are merged into one discontinuity edge.

5.5 Triangulation

5.5.1 Discontinuities and Discontinuity Edges

Until now we have glossed over the distinction between a what is computed by considering all source-blocker-receiver VE, EV, horizon, or contact combinations and what is actually inserted into the CDT. We call the former a *discontinuity* and the latter a *discontinuity edge.* This distinction is important because

1. a discontinuity edge can be composed of several collinear discontinuities. Figure 5-4 gives an example of how different source-blocker-receiver combinations can easily give rise to collinear discontinuities. In a scene of only medium complexity, Figure 7-1, the maximum number of discontinuities forming a discontinuity edge is 103. In this case the number is very high because a large number of the polygons are axis-aligned;

2. discontinuities that have insufficient weight will not be inserted into the mesh, so not all discontinuities lead to a discontinuity edge.

5.5.2 Constraints on Triangulation

Even with code to compute a CDT on an arbitrary set of 2-D segments, performing the triangulation of all quadtree leaves according to the strongest discontinuities is not trivial because we must work within several constraints:

1. For efficiency, we only want to compute all possible discontinuities once, at the top level, and we want this information to be propagated to children when quadtree subdivision occurs.

2. If a discontinuity becomes part of the CDT for one leaf of a tree, it must also be part of the CDT for all other leaves incident on that discontinuity. Otherwise a t-vertex may arise. E.g. in Figure 3-6 no t-vertex arises at point *b* because both of the quadtree leaves containing the discontinuity are triangulated according to the discontinuity.

3. The maximum number of discontinuity edges in the CDT on a leaf is bounded by a constant, `Tree::maxLeafDSets`. `DSet` is defined below.

4. At any step of the iteration, for any leaf, we wish to be able to quickly combine all collinear discontinuities of sufficient weight into a single discontinuity edge to be inserted into the CDT for that leaf. I.e. leaf triangulation may occur at any time and must be fast. Note that after every iteration, discontinuity weights may change, possibly changing the set of most powerful discontinuity edges, and requiring re-triangulation of the leaf.

We use two C++ classes, `Discontinuity` and `DSet`, to triangulate the leaves of the quadtree and update the triangulations over multiple iterations, while meeting these constraints.

5.5.3 Discontinuity

A `Discontinuity` is a non-zero length line segment annotated with some extra information. At program startup we loop over all combinations of sources, blockers, and receivers to compute all VE and EV `Discontinuities`, over all quadrilaterals whose plane splits another quadrilateral to compute all horizon `Discontinuities`, and over all quadrilaterals touching each other to compute all contact `Discontinuities`. This is much less expensive than it first appears because we use visibility preprocess information, i.e. Links. We consider only source and receiver pairs that are mutually visible, and for each pair consider only the list of potential blockers.

A Discontinuity contains:

1. two 3D endpoints, A and B, where A and B lie inside the receiver;

2. the type (VE, EV, horizon, or contact);

3. pointers to the source, blocker, and receiver quadtree nodes involved;

4. a geometric weight computed from the geometry that gave rise to the Discontinuity (Section 5.3);

5. a cached weight (geometric plus radiometric part) computed as needed (Section 5.3);

6. a list of all DSets containing the Discontinuity (Section 5.5.5);

7. an active flag (Section 5.5.7).

5.5.4 DSets

I introduce an algorithm using a data structure called a DSet (Discontinuity Set). A DSet for a Tree T is a nonempty set of collinear Discontinuities impinging on T. All collinear segments will be in the same DSet.

A DSet consists of:

1. a set of Discontinuities;

2. a pointer to the Tree containing the DSet;

3. an active flag (Section 5.5.7).

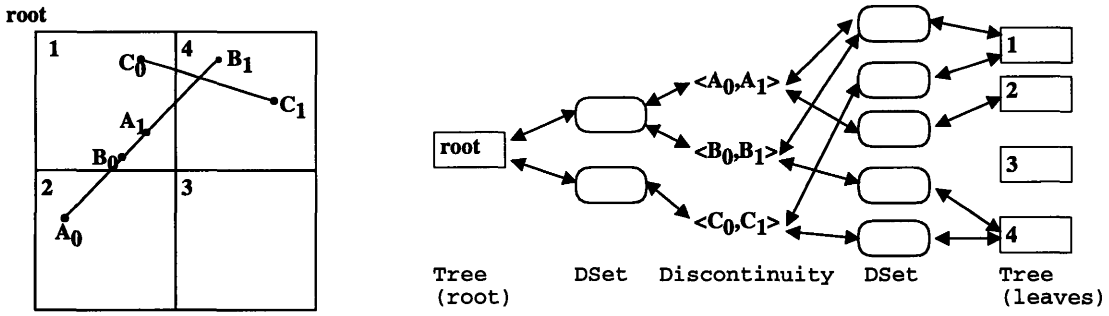

Figure 5-5: Depth 1 quadtree with `Discontinuities` and `DSets` for the **root** and four children of the quadtree. $< X_0, X_1 >$ means a `Discontinuity` from X_0 to X_1.

5.5.5 Discontinuity and DSet Relationships

The relationship between `Discontinuities`, `DSets`, and `Trees` is complex. An example is given in Figure 5-5. Note that all associations are two way. Given a `Tree`, we can find all `Discontinuities` on that tree, and given a `Discontinuity`, we can find all `Trees` containing that `Discontinuity`. The first direction is necessary for computing the discontinuity edges to be inserted in the CDT for a `Tree`. The second is needed to determine whether a `Discontinuity` can be activated without violating the the `Tree::maxNumDSets` requirement on any of the leaves containing it. Strictly speaking, a `Discontinuity` only needs pointers to all the `DSets` on leaf `Trees` which contain it.

5.5.6 Sharing Memory and Inheriting Discontinuities

C++ has an explicit memory management system. This means that we must be careful to free memory no longer in use and to make reference only to memory that is allocated. `Discontinuities` are shared by all quadtrees and all `DSets` as there is no reason for different structures to see different state. Plus, `Discontinuity` weight caching is more efficient, as after a `Discontinuity` internally computes and caches its weight, all structures referencing that `Discontinuity` can use the cached value. The root of a quadtree takes responsibility for the memory of all `Discontinuities` under it, deleting the memory when it is deleted. This is the right thing to do, as the only structures referencing these `Discontinuities` are quadtrees or structures on quadtrees under the root that are also deleted when the root is deleted.

DSets, on the other hand, cannot be shared. Only some of the Discontinuities in a parent Tree impinge on a given child of that Tree. The child needs to have its own DSet containing exactly the Discontinuities of the parent DSet that impinge on it. In Figure 5-5 **root** has a DSet containing the Discontinuities $< A_0, A_1 >$ and $< B_0, B_1 >$, while its child **2** has a clipped version of the DSet that only contains $< A_0, A_1 >$. If all the Discontinuities are clipped away, the child does not need that DSet at all. In Figure 5-5 child **3** does not have any DSets at all. The method DSet::Clip() performs this DSet clipping and copying. DSets are associated with the Tree they are contained in, a node of the quadtree. When a child Tree comes into existence from the subdivision of a parent Tree, new DSets are created for the child with DSet::Clip(). Similarly, when a Tree is deleted, it deletes all its DSets. This propagation of Discontinuities from root to children satisfies Constraint 1.

5.5.7 Active Discontinuities and Active DSets

Constraint 2, implies that if a Discontinuity is inserted as part of a discontinuity edge in one leaf, then it must be inserted in all other leaves incident on it. This prevents us from performing triangulation on an independent leaf-by-leaf basis. Instead, we start at the root of the quadtree and decide for all leaves at once which Discontinuities will be active. A Discontinuity has its active flag set iff it will be part of the CDT of every leaf that it intersects. Thus, a Discontinuity is either in or out of *every* leaf's triangulation, not in some and out of others, and Constraint 2 is met.

We also introduce an active flag in the DSet. A DSet is active iff it contains an active Discontinuity. Every DSet on a Tree leaf may contribute one or no discontinuity edges to the CDT of the leaf, depending on whether or not it is active. A DSet in a leaf is active iff it generates a discontinuity edge in the triangulation of the leaf.

To compute the active Discontinuities and DSets in a toplevel quadtree:

1. Sort all Discontinuities by weight (using the standard C library function qsort).

2. Traverse this sorted list from highest to lowest, trying to activate all Discontinuities. A Discontinuity can be activated if it is above the minimum weight and if activating it would not cause the number of active DSets on any leaf to go above Tree::maxLeafDisconts. Thus, constraint 3 is met.

The set of active DSets is always kept up to date, but leaves are only triangulated as needed. To triangulate a leaf, we ask each active DSet in the leaf to compute one segment that contains the union of its active Discontinuities. This is done quickly by searching for the extremal endpoints in the set of all the collinear line segments. Constraint 4 is met. We use these extremal segments (discontinuity edges) as the constraint edges in a call to the CDT algorithm.

5.5.8 Discontinuities in the Radiosity Algorithm

We want to make sure that whenever we use the interpolants on a leaf, the triangulation of the leaf exists and corresponds to the discontinuities that are currently most powerful. Note that triangulating leaves and creating interpolants is expensive, while computing the discontinuity weights and choosing the active DSets is cheap.

- At program startup compute all Discontinuities and the initial set of active DSets.

- For each iteration of the radiosity algorithm:

  - Run the gather/pushpull algorithm described in Section 4.3.

  - Invalidate all Discontinuity weights on all Discontinuities in all Trees.

  - Compute the active DSets in all toplevel quadtree. This will recompute and cache the Discontinuity weights.

  - Refine the quadtrees (Section 4.4). Any time the interpolants on a leaf are referenced either, check that the existing triangulation is consistent with current set of active `DSets`, or if there is no existing triangulation make a new triangulation and build interpolants on it.

Chapter 6

Accurate Point Irradiance

In the spirit of "two-pass" methods [29, 23], we use the coarse hierarchical radiosity solution to compute more accurate radiosity values at specific points on the geometry, i.e. to implement `IrradianceAtPoint()`, the third requirement for the high fidelity rendering algorithm (Section 3.1). For a point p, we compute the point-to-polygon form factors from p to all sources visible from p. To achieve point-to-source form factors with accurate visibility, we "backproject" potential blockers to the source [10], discounting the source fragments thereby obscured.

We found that, although backprojection computation is expensive, it is necessary to provide the desired level of accuracy in `IrradianceAtPoint()`. Initially in the partially visible case we tried subsampling the source, dividing it into a regular grid and summing the point to quadrilateral form factors for every grid square whose center is visible to p. However, this method resulted in aliasing problems so severe that quadratic interpolants could not be fit to the computed solution.

6.1 Fully Visible Point-to-Polygon Form Factor

Before going into the details of backprojection, we give the equation for the fully visible form factor from a differential surface element at p on quadtree R to an n-sided polygon S [6].

$$FF_{pS} = \frac{1}{2\pi} \sum_{g \in G_S} N_R \cdot \Gamma_g \tag{6.1}$$

where: G_S is the set of edges in S.

N_R is the surface normal for R.

Γ_g is a vector with magnitude equal to the angle gamma (in radians) illustrated in Figure 6-1 and direction given by the cross product of the vectors R_g and R_{g+1} as illustrated in Figure 6-1.

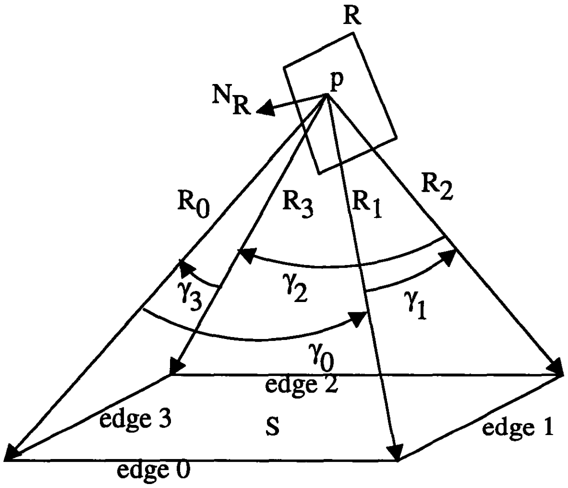

Figure 6-1: Geometry for fully visible analytic form factor from p to S.

6.2 Backprojection

To compute the irradiance at point p on quadtree R, traverse all `Links` contributing energy to R. For each link L add the contribution of L's source to p's irradiance. The contribution of a source quadrilateral S to p, is the source radiosity multiplied by the form factor from p to S.

The difficulty lies in computing the point-to-polygon form factor from p to S.

We cannot simply compute equation (6.1), as S may not be fully visible to p due to intervening blockers or horizon effects.

6.2.1 Horizon Effects

If the source quadtree node S is split by the plane of the receiver we clip S to the plane of R and continue with the clipped source. Note that this may turn the source into a three- or five-sided polygon. By simple form factor algebra [9], this gives the proper form factor from p to S. If R is split by the plane of S we check the sidedness of p relative to S. If p is behind, the form factor is 0, otherwise we continue.

6.2.2 Full Visibility

If L has no blockers, compute the fully visible point-to-polygon form factor equation (6.1) from from p to the (possibly clipped) source polygon and we are done. No backprojection needs to be constructed.

6.2.3 Partial Visibility

If the interaction is not fully visible, then we must compute the backprojection of p onto S, the portion of S visible to p (Figure 6-2). This is extremely similar to the problem of computing a discontinuity mesh (Chapter 5). In fact, we reuse the wedge intersect and CDT code already present in our system.

To compute the backprojection (Figure 6-2):

1. Project from point p to S all edges of all blocker quadrilaterals.
2. Use these projected line segments to form a CDT on the source.
3. Classify each triangle as visible or invisible by shooting a single ray from p to the center of the triangle.
4. Sum the point to triangle form factors from p to all visible triangles.

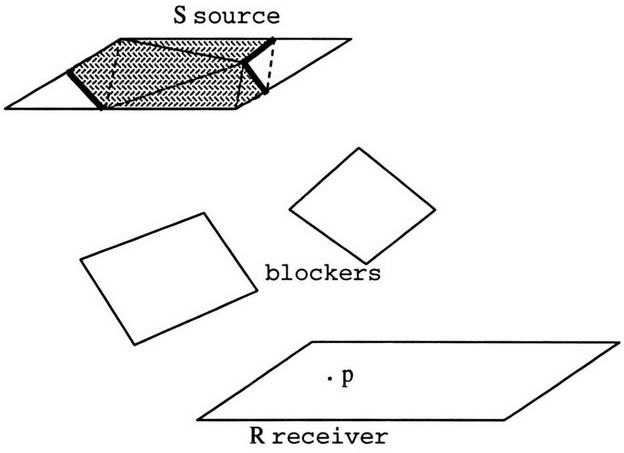

Figure 6-2: Backprojection of p onto S. Dark lines are backprojected blocker segments. Dotted lines are edges added in the triangulation of S. Shaded triangles represent the region of S visible to p.

Each face of the graph G formed from the backprojection of all blocker edges is guaranteed to be either entirely visible or invisible to p [10]. The CDT is a refinement of G, so it also has this property. Thus, determining the visibility of a single point in each triangle is sufficient to determine the visibility of the entire triangle. By form factor algebra, the form factor from a point to an area is the sum of the form factors from the point to each face of a partition of that area. So, this method gives us the accurate form factor from p to S.

6.3 Point to Point Visibility

Visibility from a point p on a receiver R to a point q on a source S is computed using the precomputed blocker list as in [32]. However, there is one case not handled by Teller's algorithm that requires our attention. If p is on the edge of a blocker as in Figure 6-3 its visibility can only be defined in the limit approaching p from above or below the blocker plane.

In all cases where we call `IrradianceAtPoint()`, we are computing the irradiance of a point on a non-zero area triangle and we are concerned with the irradiance at that point as approached from inside the triangle. Since all contact discontinuities are inserted into the discontinuity mesh, the triangle cannot cross the trace of the blocker on the receiver. Thus, in this special case, the center of the triangle can be used to define the visibility of q with respect to p. q is visible to p iff it is visible in the limit approaching p from the triangle center. To deal with this case in the implementation, we simply nudge p a small fixed amount in the direction of the appropriate triangle center and call the point-to-point visibility code (Appendix A).

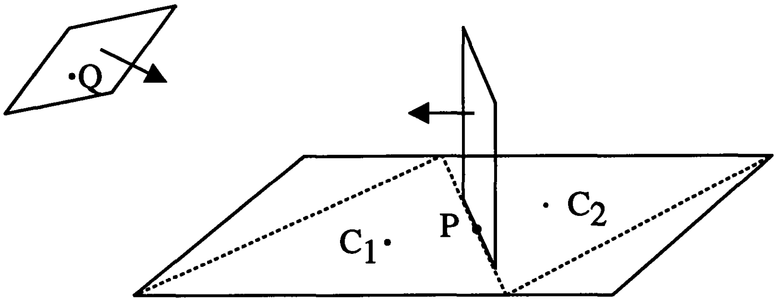

Figure 6-3: q is visible from p when p is approached from the triangle centered at c_1, but invisible when p is approached from the triangle centered at c_2.

6.4 Source-Receiver Contact

We run into trouble when S is adjacent to R and S ends on the inside of one of R's edges (Figure 6-4). Equation (6.1) becomes highly nonlinear for points on R in the neighborhood $\mathcal{N}$. So nonlinear, in fact, that quadratic interpolants around $\mathcal{N}$ cannot be fit within error tolerance no matter how much quadtree subdivision occurs.

We solve the problem by computing the irradiance function as if S were not actually touching R. When computing the contribution of a source S to p, we first shift S a small amount (Appendix A) before using the analytic point-to-polygon form factor equation. This banishes the "bad" neighborhood $\mathcal{N}$ from R.

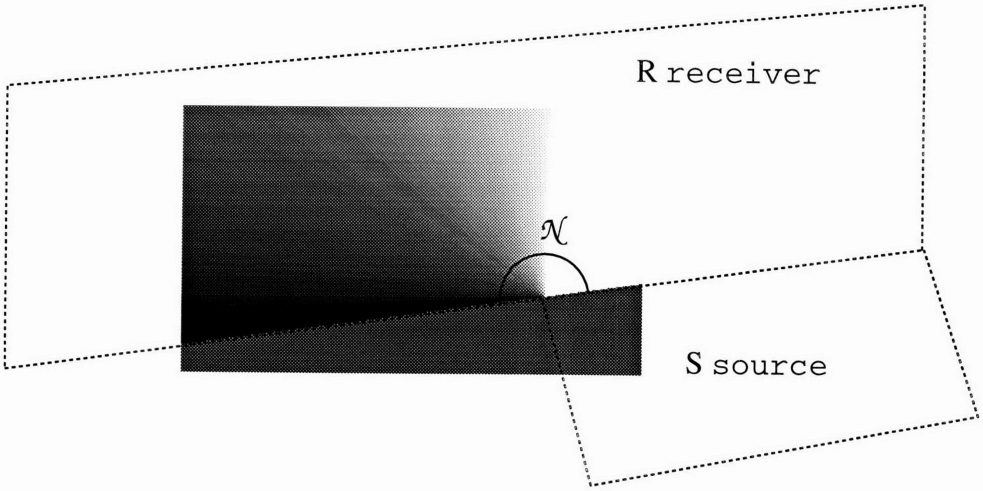

Figure 6-4: Analytic form factor equation is highly multivalued and nonlinear in the neighborhood $\mathcal{N}$.

Chapter 7

Results

We implemented these algorithms on a Silicon Graphics Reality Engine with four 250 MHz MIPS RISC R4400 CPUs and 512Mb of memory. The underlying radiosity solver is a reimplementation, in C++, of the hierarchical radiosity and wavelet radiosity algorithms described in [17, 14, 32]. The system components and code complexity are as follows:

- Form factor and radiosity solver (18,000 lines of C++);
- Interpolant module (1500 lines of C++);
- User interface (4500 lines of C++);
- Rendering module (3000 lines of C++);
- Basic computational geometry and math modules (3000 lines of C++).

7.1 Test Scene

Our test scene, comprised of about sixty quadrilaterals, is shown in Figure 7-1. The hierarchical radiosity algorithm, with the allowable error `Gur::eps` set to $0.1W/m^2$, the maximum number of discontinuity edges per quadtree leaf `Tree::maxLeafDSets` set to 10, and the minimum discontinuity weight `Tree::minDiscontWeight` set to 100, ran to convergence on this input in less than two minutes, and meshed the input surfaces into 1622 quadtree (leaf) elements. After triangulation, there were 6576 triangles (interpolants), an average of about 4 triangles per element. Figure 7-3 shows the resulting quadtrees and triangulations. The discontinuity edges actually used in the triangulations are a subset of those in Figure 5-2.

The interpolant construction was clearly the bottleneck, requiring two and a half hours of CPU time (running on a single processor), about eighty times the cost of computing the radiosity solution.

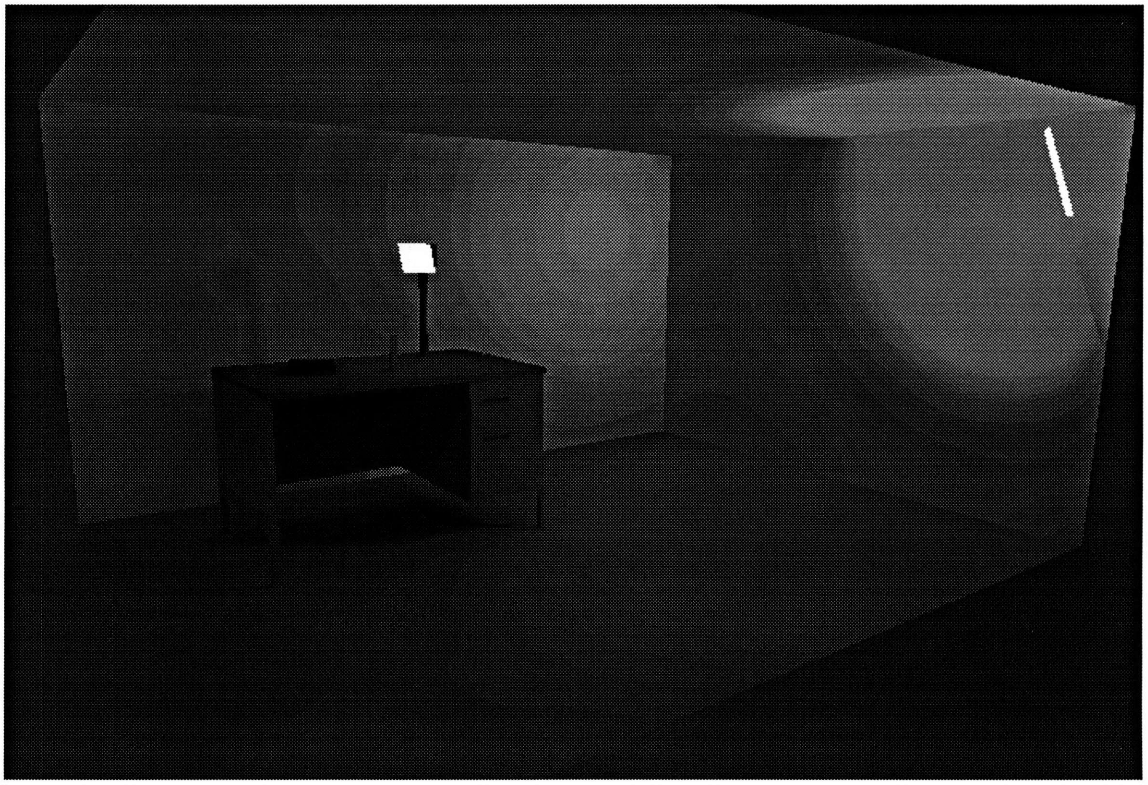

Figure 7-1: One frame of an interactive viewing session.

Figures 7-1 and 7-2 are screen snapshots taken from an interactive session viewing the office model at NTSC (640 × 480) resolution. Our real-time rendering algorithm achieved an average of 2.3 updates per second for this model. This simple office scene serves to highlight the discontinuity resolution and shading abilities of the techniques described here. Note that although the underlying mesh is relatively coarse, it still yields a crisp image due to the use of irradiance interpolants.

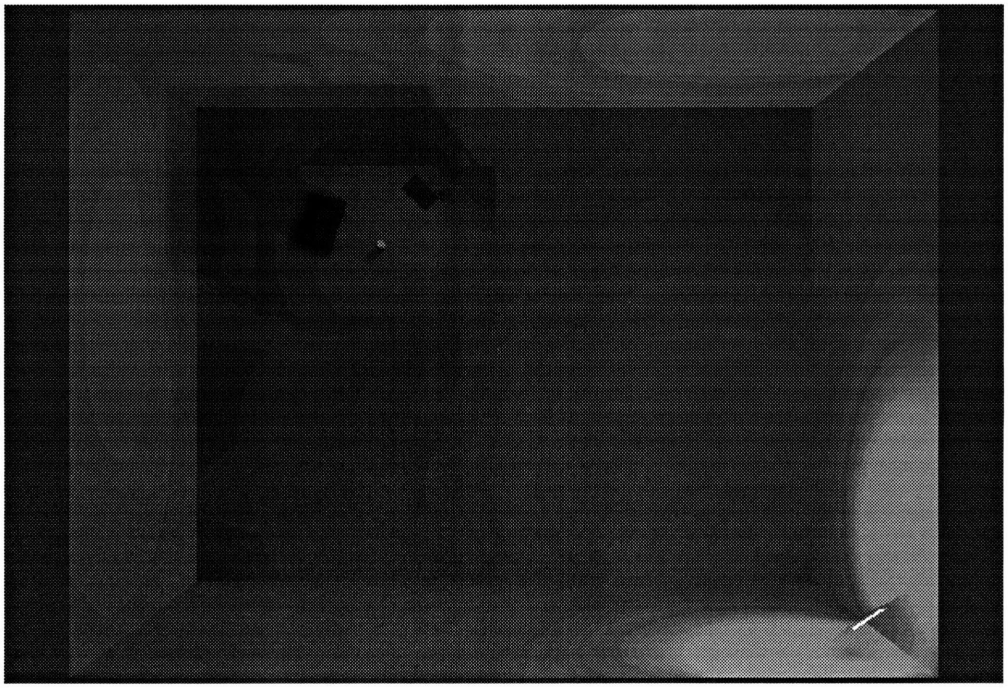

Figure 7-2: A second frame, bird's eye view.

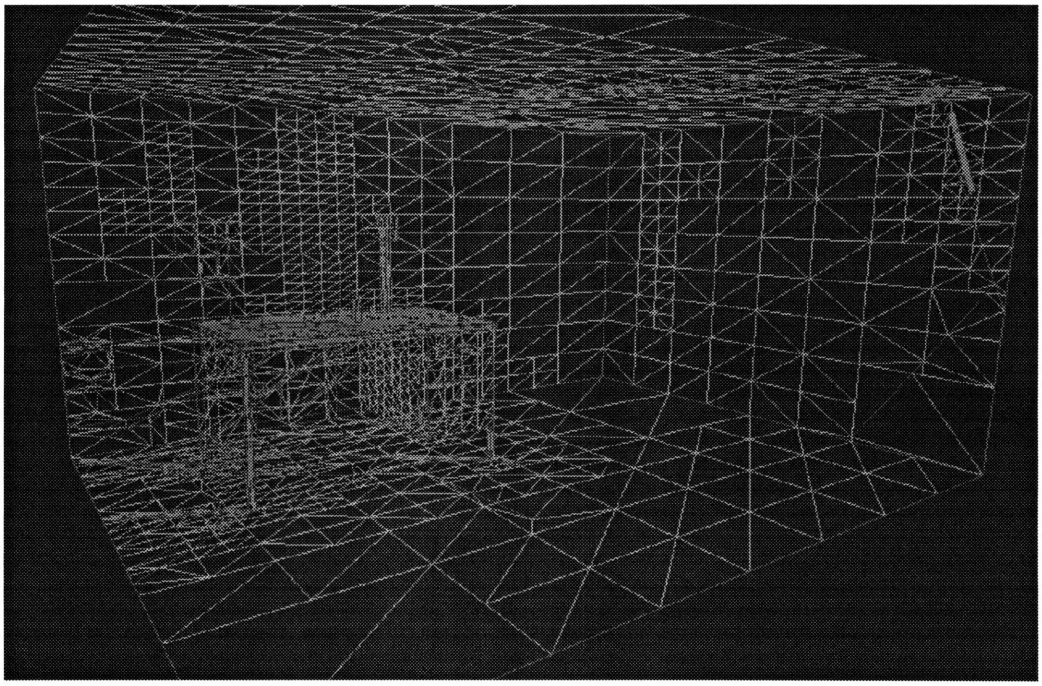

Figure 7-3: Mesh used for Figures 7-1 and 7-2.

Figure 7-4 shows a detail view of the corner near the light source, which contains a strong horizon discontinuity. The difference between interpolant rendering and Gouraud-shaded rendering is particularly evident in this region.

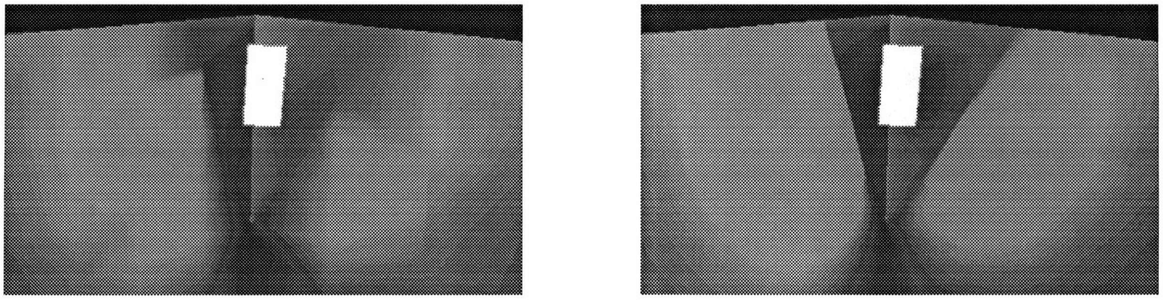

Figure 7-4: Gouraud shading (left) vs. quadratic interpolant rendering (right).

Chapter 8

Limitations and Future Work

8.1 Limitations

Our implementation has at least two limitations, namely 1) it processes only polyhedral scenes, and 2) since our techniques rely on graphics hardware, they operate with relatively low numerical precision. However, we expect the latter concern to be ameliorated by the increasing precision of next-generation software and hardware architectures. Generalizing to non-polyhedral geometries remains a difficult problem.

8.2 Improved Weighting Metric

We are experimenting with an improved method for ranking discontinuities by their worst-case radiometric "weight." The weight of a discontinuity produced by a specific source, blocker, receiver combination is derived by bounding the maximum change in the radiosity function, per unit length on the receiver. This weight will be proportional to the source radiosity, and to the change in solid angle subtended by the source as viewed from the receiver, as a result of motion on the receiver across the discontinuity. The discontinuity meshing scheme could then be tied into the global error bound `Gur::eps` (Appendix A, 1).

8.3 Representation-Based Refinement

Currently, representation-based refinement (Section 4.4.2) is too slow to be practical because it requires interpolant construction at every stage of the iteration algorithm. We are looking into methods for accelerating calls to `IrradianceAtPoint()` and for bounding representation error without actually constructing all interpolants.

8.4 Extension to Radiance

Radiosity algorithms are giving way to those for radiance, a much more complex, 4-dimensional quantity associated with each surface point's position, and a direction from which the point is viewed. We plan to extend our algorithms to operate on a radiance data structure, by rendering (a discretized representation of) the direction (θ, ϕ) from which each surface point is viewed, in object space. The assembled values (s, t, θ, ϕ) will then be used to evaluate a suitable 4-dimensional quadratic interpolant.

Chapter 9

Conclusion

Generating high-quality imagery that precisely captures diffuse irradiance is a computationally expensive proposition, and is arguably unachievable using polygon-based linear-shading hardware alone. We presented a scheme in which fast integer and floating point units, and a substantial amount of general purpose memory, were used to capture a representation of irradiance for every point on every surface in a scene. Later, in combination with a fast massively parallel graphics hardware rendering architecture, the data structure is queried to produce quadratically interpolated radiosity renderings at interactive rates.

One of the scheme's strengths, its use of rendering hardware, can also be considered a limitation due to that hardware's limited precision. However, software advances (e.g., OpenGL) and hardware augmentations (e.g., higher iterator precision and framebuffer resolution, larger textures, and object-space "Gouraud" interpolation) should make these techniques both more efficient and accurate.

The realization of this technique required advances at both theoretical and practical levels. The theoretical advances of this paper were the ranking of discontinuities by relative strengths, and a "factoring" of radiosity rendering into online and offline components. The practical advances were the use of texture-mapping hardware for barycentric coordinate generation, and the introduction of a hybrid quadtree-mesh, a quadtree with discontinuity-meshed leaves.

Appendix A

List of all Parameters in System

1. A global error, `Gur::eps`, in W/m^2 for the transport-based and representation-based refinement schemes.

2. The minimum allowed area, `Gur::aeps`, for a quadtree node in m^2.

3. A floating point roundoff error `__m_eps_f` for computing equality. Floats a and b are considered equal if $\mid a - b \mid <$ `__m_eps_f`.

4. The minimum weight for a `Discontinuity` to be introduced into the quadtree triangulation, `Tree::minDiscontWeight`.

5. The maximum number of discontinuity segments used to triangulate a quadtree leaf, `Tree::maxLeafDSets`.

6. The minimum length of a backprojected segment introduced into the backprojection triangulation, `Backprojection::minBPSegment`. Necessary because the CDT code is not stable when the inserted segments are too small. This is a larger value than `__m_eps_f`.

7. The relative error cutoff in the iterative algorithm, `Basis::relErrorCutoff`. This determines when the solution has converged.

8. The number of extra sample points computed to determine the representation error of an interpolant, `SAMPLES` × `SAMPLES`.

9. The distance to push the source when computing `IrradianceAtPoint()`, `Polygon::ffPushOff`.
10. The distance to nudge a point on the edge of a blocker to resolve visibility, `nudgeAmt`.

Appendix B

User Interface

This describes the Motif interface to the system along with the OpenGL 3D visualization tools.

B.1 Command Line

I only mention the most important command line options. Run "gur -h" to see all options. "-N" sets the number of draw slave processes to use in interpolant rendering mode. E.g. You might use "gur -N 4 scene.lrad" on a four processor machine. "-D" disables double buffering. For machines with less than 48 bits per pixel, e.g. Indy or Indigo II, this is the only way to see the full-resolution radiosity mode.

B.2 Layout

Upon program startup, the main window will appear. The menubar is at the top with a small status indicator in the upper-right. The drawing area takes up the rest of the window. There is one other top-level window, the Drawing Control Panel, which is initially hidden. Press **d** or click the **Drawing Control Panel** button under the **Drawing** menu item to hide/unhide the panel.

B.3 User Interface

B.3.1 Viewing Modes

Arcball

The default viewing mode, arcball, comes from some IrisGL code written by Ken Shoemake that I ported to OpenGL. This mode can be entered by selecting **Arcball** in the **Viewing** menu. Hold down the left mouse button and drag the virtual trackball to rotate the model. If the **shift** or **ctrl** key is held down while dragging with the left button, motion will be restricted to rotation around the coordinate axes of the viewer or of the object, respectively. Hold down the middle mouse button and drag to translate the model in screen space. Press **shift** and the middle button and drag the cursor up/down to zoom in/out. Press **ctrl** and the middle button and drag the cursor up/down to change the perspective warp.

Frustum

Select **Frustum** in the **Viewing** menu to enter the fly-through (frustum) mode. Left mouse button moves the eyepoint forward, right moves it back. Pressing middle button will rotate the eyepoint in the direction corresponding to the location of the cursor on the screen. **X Up**, **Y Up**, and **Z Up**, in the **Viewing** menu specify the up vector of the viewing frustum. **Frustum Speed** activates a dialog to specify the speed at which the eyepoint moves through the scene.

B.3.2 Drawing Modes

The polygon drawing mode can be set from the **Viewing** menubar item. Modes can be switched at any time and will always display the current state of the radiosity solver. Certain modes require structures to be built or information to be computed, so when switching drawing modes the system may take time to compute before displaying anything.

In all four modes the **Irradiance-2-Pixel** slider in the Drawing Control Panel controls the mapping between irradiance values in W/m^2 and pixel colors. This is useful if the image appears too dark or too bright. Non-emitters are colored with the product of irradiance, reflectance of the surface, and this mapping. Emitters are always drawn white.

Flat Shading

This mode can be used to render the scene using only the constant basis functions, but its primary purpose is to display the scene geometry and the visualization tools

Gouraud Shading

The scene is rendered as quadrilaterals corresponding to the leaves of the quadtree. The system must first calculate and cache radiosity values on all quadtree leaf vertices.

Gouraud Shading with Discontinuities

The scene is rendered as a list of triangles, the triangulation of the quadtree leaves. Each triangle is gouraud shaded from the evaluation of its interpolant at the vertices.

Interpolant Rendering

High fidelity interpolant rendering mode as described in Chapter 3. Use the **Full Window Radiosity** button on the Drawing Control Panel to toggle between rendering scratch data in the backbuffer (default) or splitting the viewport in four and rendering scratch and final images to different sections.

B.3.3 Per-pixel Form Factors

The system provides two methods of creating a raytraced image. The **RayShade** button in the **File** menu shoots several rays through each pixel and evaluates `IrradianceAtPoint(` where the rays hit the geometry, i.e. a per-pixel form-factor evaluation. **LerpShade** is similar except that the underlying interpolant is evaluated at every pixel to produce the color. This produces an image similar to that produced by the **Interpolant Rendering** mode.

B.3.4 Computing Radiosity Solution

In the **Iteration** menu, the **Iterate(Refine)** button triggers one iteration of the transport-based refinement scheme. The **Iterate(Refine Interpolants)** button triggers the representation-based refinement scheme. The representation-based iteration will iterate once or repeatedly until convergence depending on the setting of the **Iterate to Convergence** toggle.

Note: When **Iterate(Refine Interpolants)** is pressed, the **Compute Rep. Error** option will automatically be selected. This means that when an interpolant is built, extra samples will be taken to compute the representation error of the interpolant.

B.3.5 Setting Parameters

Important parameters for the system:

1. **Max Error** in the **Iteration** menu sets the error for the representation or transport-based refinement algorithms, `Gur::eps` (Section 4.4, Appendix A).

2. **Min Area** in the **Geometry** menu sets the quadtree leaf area at which subdivision bottoms out `Gur::aeps` (Appendix A).

3. **Max Leaf Discont Segments** in the **Geometry** menu sets the maximum number of discontinuity segments that will be used to triangulate a quadtree leaf (Section 5.2, Appendix A).

4. **Minimum Discontinuity Weight** in the **Geometry** menu sets the minimum weight for a discontinuity to be considered in the triangulation of a quadtree leaf (Section 5.2, Appendix A).

B.4 Visualization Tools

We built a number of 3D visualization tools directly into the application. These tools visually display the internal state of the radiosity solver and renderer, giving valuable debugging and sanity-checking information to the developers and providing a means of demonstrating the algorithms to others.

These tools must all be used in the **Flat Shading** drawing mode.

B.4.1 Quadtree Face Color

By default, quadtree faces will be colored with the value of the underlying basis coefficients. If **Draw Trees Using Rho** is set, trees will be drawn with their reflectance.

B.4.2 Quadtree Mesh

The mesh of the HR radiosity solution can be displayed with **Element Mesh** in the Drawing Control Panel. Sometimes it is useful to turn off **Draw Faces** to better see the mesh. The depth to which the mesh is drawn is controlled by the **Mesh Depth** text field.

B.4.3 Links

Links between quadtree nodes are drawn as white, pink, or green arrows. White represents a fully visible interaction. Pink represents an interaction where either the source or receiver tree node splits the plane of the other. Green represents partially visible interactions. Display of these links can be toggled with the Drawing Control Panel **Visible Links** and **Partial Links** buttons.

B.4.4 Discontinuities

If **Draw Discontinuities** is selected in the Drawing Control Panel, all the potential discontinuities in the scene will be drawn as violet line segments. This does not take into account discontinuity weight.

B.4.5 Triangulation

The actual triangulation being used as a basis for the interpolants is viewed by selecting **Draw Triangles** in the Drawing Control Panel. Violet triangle edges represent discontinuity segments in the mesh, green segments represent other segments inserted into the CDT to satisfy the constraints imposed by the discontinuity segments. Note that this is showing exactly which discontinuities have a large enough weight to be inserted into the discontinuity mesh. It is sometimes easier to see the triangulation if **Draw Faces** and **Draw Discontinuities** in the Drawing Control Panel are turned off.

B.4.6 Selection

By clicking the left mouse button on certain structures, the user can select an item of interest so that only the selected item will be drawn. Double clicking on the background or choosing **Unselect All** in the **Geometry** menu will unselect all items.

A quadtree node can be selected by clicking on a top-level quad, or, if a quadtree node is already selected, by clicking on one of its children.

Clicking a link once will select that link, clicking on it twice will select the link's tube, displaying the blockers and information about the link.

If **Draw Triangles** is set in the Drawing Control Panel, clicking on a triangle in a quadtree leaf will select that triangle.

If **Draw Discontinuities** or **Draw Triangles** is set in the Drawing Control Panel, discontinuities can be selected. The source, blocker, receiver triple will be displayed, along with information printed to the command line.

B.4.7 Triangle Graphs

When a tree (or triangle) is selected, you can display a graph of the irradiance over the tree (triangle) as given by `IrradianceAtPoint()` (drawn in white) or as given by the interpolants (drawn in blue) by turning on **Analytic Irradiance** or **Interpolated Irradiance** in the Drawing Control Panel, respectively. Also, a graph of the difference (drawn in red) between the two can be displayed by turning on **Error Surface**.

The scale of the graphs, the arbitrary mapping from irradiance in W/m^2 to linear distance in m, can be controlled with the **Graph height scale** slider bar. The grid density of the graphs can be controlled in a crude way with the **Dynamic Grid Density** toggle. If turned on (default), the density of the grid over a triangle will be proportional to the ratio of the area of that triangle to the area of the quadrilateral it is in. Thus, the sum number of points in all the graphs over a quadrilateral will be approximately the same no matter how finely triangulated the quadrilateral is. If turned off, every triangle, no matter how small will be drawn with the same density grid.

A caveat: the existing X/Motif interface is not interrupt-driven, so the system cannot be interrupted when it is drawing or performing calculations. You must be careful not to activate any time-consuming function if you are not willing to wait for it to finish. Drawing the analytic irradiance or error graphs may be slow as the system must call **IrradianceAtPoint()** at each point on the graph. Be especially careful if a tree with many triangles is selected and **Dynamic Grid Density** is turned off. Also, if interpolants have not already been built for the selected tree or triangle, turning on **Interpolated Irradiance** will take time to first build the interpolants.

B.4.8 Sample Points

When a tree or triangle is selected, turning on **Draw Sample Points** in the Drawing Control Panel will show (in yellow) the sample points and values used for interpolant construction on each triangle.

B.4.9 Irradiance Probe

The right mouse button activates the general purpose irradiance probe. Clicking a point on some quad will draw (in white) the analytic irradiance at that point. Also, the coordinates, the analytic irradiance, the interpolant value, and the difference between analytic and interpolant values at that point will be printed to standard out.

B.4.10 Backprojection Probe

To view the backprojection of a point back onto all its sources, hold the shift key and click the right mouse button on any point in the scene. On each potential source, the backprojection graph (Section 6.2) is drawn as violet and green line segments with faces that are solid white or transparent. Violet segments are backprojected blocker segments and green segments are other segments inserted to make the constrained Delauney triangulation. Regions of the source visible to the point are drawn white and invisible regions are left transparent. It is helpful to turn off **Draw Faces**, **Draw Triangles**, and **Draw Discontinuities** when viewing a backprojection.

Bibliography

[1] John M. Airey, John H. Rohlf, and Frederick P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. *ACM Siggraph Special Issue on 1990 Symposium on Interactive 3D Graphics*, 24(2):41–50, 1990.

[2] Kurt Akeley. RealityEngine graphics. *Computer Graphics (Proc. Siggraph '93)*, pages 109–116, 1993.

[3] James Arvo. The irradiance Jacobian for partially occluded polyhedral sources. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 343–350. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[4] D. Baum, S. Mann, K. Smith, and J. Winget. Making radiosity usable: Automatic preprocessing and meshing techniques for the generation of accurate radiosity solutions. *Computer Graphics (Proc. Siggraph '91)*, 25(4):51–60, 1991.

[5] D. Baum and J. Winget. Real time radiosity through parallel processing and hardware acceleration. *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, 24(2):67–75, March 1990.

[6] Daniel R. Baum, Holly E. Rushmeier, and James M. Winget. Improving radiosity solutions through the use of analytically determined form factors. *Computer Graphics (Proc. Siggraph '89)*, 23(3):325–334, 1989.

[7] Derrick Burns. *Dynamic Trimmed Surface Rendering.* PhD thesis, Stanford University, 1993.

[8] M. Cohen and D. Greenberg. The hemi-cube: A radiosity solution for complex environments. *Computer Graphics (Proc. Siggraph '85)*, 19(3):31–40, 1985.

[9] M. F. Cohen and J. R. Wallace. *Radiosity and Realistic Image Synthesis.* Academic Press Professional, 1993.

[10] George Drettakis and Eugene Fiume. A fast shadow algorithm for area light sources using backprojection. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 223–230. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[11] J.D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, 1990.

[12] Henry Fuchs, J. Poulton, J. Eyles, T. Greer, J. GoldFeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel. Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. *Computer Graphics (Proc. Siggraph '89)*, 23(3):79–88, 1989.

[13] Benjamin Garlick, Daniel R. Baum, and James M. Winget. Interactive viewing of large geometric databases using multiprocessor graphics workstations. *Siggraph '90 Course Notes (Parallel Algorithms and Architectures for 3D Image Generation)*, 1990.

[14] S. Gortler, P. Schröder, M. Cohen, and P. Hanrahan. Wavelet radiosity. *Computer Graphics (Proc. Siggraph '93)*, pages 221–230, August 1993.

[15] E. Haines and J. Wallace. Shaft culling for efficient ray-traced radiosity. In *Proc. 2^{nd} Eurographics Workshop on Rendering*, May 1991.

[16] Pat Hanrahan and Paul E. Haeberli. Direct WYSIWYG painting and texturing on 3D shapes. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 215–223, August 1990.

[17] Pat Hanrahan, David Salzman, and Larry Aupperle. A rapid hierarchical radiosity algorithm. *Computer Graphics (Proc. Siggraph '91)*, 25(4):197–206, 1991.

[18] P. Heckbert. *Simulating Global Illumination Using Adaptive Meshing.* PhD thesis, CS Department, UC Berkeley, June 1991.

[19] Paul Heckbert. Discontinuity meshing for radiosity. *Third Eurographics Workshop on Rendering*, pages 203–226, May 1992.

[20] Nicolas Holzschuch and Francois Sillion. Accurate computation of the radiosity gradient for constant and linear emitters. In P. M. Hanrahan and W. Purgathofer, editors, *Rendering Techniques '95 (Proceedings of the Sixth Eurographics Workshop on Rendering)*, pages 186–195, New York, 1995. Springer-Verlag.

[21] David Kirk and Douglas Voorhies. The rendering architecture of the DN10000VS. *Computer Graphics (Proc. Siggraph '90)*, 24(4):299–307, 1990.

[22] Dani Lischinski. Incremental delaunay triangulation. In Paul Heckbert, editor, *Graphics Gems IV*, pages 47–59. AP Professional, 1994.

[23] Dani Lischinski, Filippo Tampieri, and Donald P. Greenberg. Discontinuity meshing for accurate radiosity. *IEEE Computer Graphics and Applications*, 12(6):25–39, 1992.

[24] Dani Lischinski, Filippo Tampieri, and Donald P. Greenberg. Combining hierarchical radiosity and discontinuity meshing. *Computer Graphics (Proc. Siggraph '93)*, 27, 1993.

[25] Charles Loop and Tony DeRose. Generalized B-spline surfaces of arbitrary topology. *Computer Graphics (Proc. Siggraph '90)*, 24(4):347–356, 1990.

[26] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide.* Addison-Wesley, 1993.

[27] David Salesin, Dani Lischinski, and Tony DeRose. Reconstructing illumination functions with selected discontinuities. In *Proc. 3^{rd} Eurographics Workshop on Rendering*, pages 99–112, May 1992.

[28] Peter Schröder. http://csvax.cs.caltech.edu/~ps/.

[29] Francois X. Sillion and Claude Puech. A general two-pass method integrating specular and diffuse reflection. In Jeffrey Lane, editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 335–344, July 1989.

[30] Brian Smits, James Arvo, and Donald Greenberg. A clustering algorithm for radiosity in complex environments. *Computer Graphics (Proc. Siggraph '94)*, 28, 1994.

[31] Seth Teller. Computing the antipenumbra cast by an area light source. *Computer Graphics (Proc. Siggraph '92)*, 26(2):139–148, 1992.

[32] Seth Teller and Pat Hanrahan. Global visibility algorithms for illumination computations. *Computer Graphics (Proc. Siggraph '93)*, 27:239–246, 1993.

[33] Gregory J. Ward and Paul Heckbert. Irradiance gradients. *Third Eurographics Workshop on Rendering*, pages 85–98, May 1992.