


1993

Parallel hierarchical radiosity rendering

Michael Brannon Carter
Iowa State University

Follow this and additional works at: <https://lib.dr.iastate.edu/rtd>

 Part of the [Computer Sciences Commons](#), [Electrical and Electronics Commons](#), and the [Mechanical Engineering Commons](#)

Recommended Citation

Carter, Michael Brannon, "Parallel hierarchical radiosity rendering" (1993). *Retrospective Theses and Dissertations*. 10409.
<https://lib.dr.iastate.edu/rtd/10409>

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9321126

Parallel hierarchical radiosity rendering

Carter, Michael Brannon, Ph.D.

Iowa State University, 1993

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Parallel hierarchical radiosity rendering

by

Michael Brannon Carter

A Dissertation Submitted to the Graduate

Faculty in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Department: Electrical Engineering and Computer Engineering

Major: Computer Engineering

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University

Ames, Iowa

1993

DEDICATION

As with all accomplishments in one's life, recognition is due to those who helped make them possible. I am speaking primarily of my parents, Everett and Murrel Carter. This dissertation was wrought as much by their hands and hearts as by my own. Without their lifetime of patient nurturing, love, understanding, and wisdom, I would never have had the chance to learn. This one's for you, Mom and Dad!

TABLE OF CONTENTS

ACKNOWLEDGMENTS.....	xi
CHAPTER I INTRODUCTION	1
1.1 Realistic Image Synthesis.....	1
1.1.1 General.....	1
1.1.1.1 Geometric object modeling.....	1
1.1.1.2 Tessellation.....	2
1.1.1.3 Discretization	2
1.1.1.4 Light reflection models	2
1.1.2 The rendering equation.....	3
1.1.3 Ray tracing.....	4
1.1.4 Radiosity	5
1.1.5 Hybrid methods	7
1.2 Parallelism in Computer Graphics	8
1.2.1 Non-realistic.....	8
1.2.1.1 Pixel-Planes 5.....	8
1.2.1.2 The Pixel Machine.....	9
1.2.1.3 SGI Reality Engine	10
1.2.2 Ray tracing.....	11
1.2.2.1 LINKS-1.....	11
1.2.2.2 Hypercube Ray Tracer	11
1.2.2.3 Other work.....	12
1.2.3 Radiosity	12
1.3 Structure and Aim of This Dissertation	12
CHAPTER II SYMMETRIC RADIOSITY.....	14
2.1 Introduction.....	14
2.2 Existing Methods	15
2.3 Reformulating the Radiosity Equation.....	16
2.3.1 Coupling factors.....	16
2.4 Solution Techniques.....	17
2.4.1 Direct solution.....	17
2.4.2 Simple iterative techniques	17
2.4.2.1 Jacobi iteration.....	18
2.4.2.2 Gauss-Seidel iteration	18
2.4.2.3 Progressive radiosity (shooting).....	19

2.4.3 Other iterative techniques	19
2.4.4 Method of conjugate gradients.....	20
2.5 A Practical Comparison	21
2.6 Applicability to Hierarchical Methods.....	24
2.7 Summary	25
CHAPTER III HIERARCHICAL METHODS.....	26
3.1 Introduction to Hierarchical Methods	26
3.2 Hierarchical <i>N</i> -body Methods.....	26
3.2.1 Appel's <i>N</i> -body algorithm.....	26
3.2.1.1 Bounding the interaction error.....	26
3.2.1.2 The algorithm	28
3.2.1.3 Analysis of time complexity.....	28
3.2.2 Barnes and Hut's <i>N</i> -body algorithm.....	31
3.2.3 Greengard's fast multipole algorithm	31
3.2.3.1 Bounding the interaction error.....	32
3.2.3.2 The algorithm	34
3.2.3.3 Analysis of time complexity.....	36
3.3 Hierarchical Radiosity Methods.....	37
3.3.1 Patch couplings and link splitting.....	38
3.3.2 Hanrahan's method	42
3.3.3 Smits' method	43
3.4 Problems Amenable to Hierarchical Methods.....	44
CHAPTER IV HIERARCHICAL RADIOSITY ENHANCEMENTS.....	45
4.1 Introduction	45
4.2 Background and Definitions.....	46
4.3 Discussion.....	46
4.3.1 Alternation of error types.....	46
4.3.2 Rowsum correction	48
4.3.3 Clustering of polygons.....	50
4.3.4 Airtight occlusion testing	52
4.3.5 Binary vs. quadtree subdivision	55
4.3.6 Flaw in area/form factor threshold reasoning.....	55
4.3.7 Link subdivision	56
4.3.8 Unidirectional vs. bidirectional links	56
4.3.9 Coupling estimates	57

4.3.10 Estimation of error in coupling estimates.....	57
4.4 Results	58
4.5 Summary	58
CHAPTER V MAKING THE HIERARCHICAL METHOD PARALLEL.....	61
5.1 Elements of a Good Parallel Program.....	61
5.2 Statement of Algorithm	62
5.3 Observations.....	67
5.4 Identifying Sources of Parallelism	68
5.4.1 Data parallelism	68
5.4.2 Operational parallelism	68
5.5 Data Decomposition Strategy.....	69
5.5.1 Node hierarchy.....	69
5.5.1.1 Hierarchy decomposition method 1.....	71
5.5.1.2 Hierarchy decomposition method 2.....	72
5.5.1.3 Hierarchy decomposition method 3.....	74
5.5.1.4 Hierarchy decomposition method 4.....	75
5.5.2 Link heap	76
5.6 Critical Operations.....	76
5.6.1 Random all-to-all communication.....	77
5.6.2 Link refinement.....	77
5.6.3 Reheapifying	83
5.6.4 Hierarchical vector operations.....	83
5.6.5 Hierarchical matrix-vector multiply.....	84
5.6.6 Writing the answer file.....	90
5.7 Results and Analysis.....	91
5.7.1 A visit from reality.....	92
5.7.2 Revised algorithm.....	96
CHAPTER VI SUMMARY AND FURTHER RESEARCH	103
6.1 Summary	103
6.2 Further research	103
6.2.1 Optimizations to existing code.....	103
6.2.2 Other areas of investigation.....	103
6.2.2.1 Exact coupling factors.....	104
6.2.2.2 Discretization error.....	104
6.2.2.3 Specularity.....	105

BIBLIOGRAPHY	106
APPENDIX	114
Header file slal.h	115
Header file proto.h	117
Source file slal.c	120
Source file solver.c	129
Source file patch.c	135
Source file heap.c	155
Source file matvec.c	161

LIST OF FIGURES

Figure 1: Three-point transport geometry	4
Figure 2: Solver iteration counts	22
Figure 3: Solver time.....	23
Figure 4: Monopole approximation	27
Figure 5: Shell structure about X.....	28
Figure 6: Interaction list of a computational box	35
Figure 7: Interaction between two clumps of particles	36
Figure 8: Physical and hierarchical interpretation	39
Figure 9: Link refinement steps preceding Figure 8.....	40
Figure 10: Tartan artifact.....	49
Figure 11: Coupling estimate and actual coupling.....	50
Figure 12: Rowsum corrected scene	51
Figure 13: Construction of waist hull	53
Figure 14: Harpsichord practice room without rowsum correction.....	59
Figure 15: Harpsichord practice room with rowsum correction	60
Figure 16: Construction of composite hierarchy.....	64
Figure 17: Couplings in a patch hierarchy	64
Figure 18: Structure of coupling matrix	65
Figure 19: Cases of parallel link subdivision.....	70
Figure 20: Hierarchy decomposition method 1.....	71
Figure 21: Hierarchy decomposition method 2.....	73
Figure 22: Hierarchy decomposition method 3.....	74
Figure 23: Hierarchy decomposition method 4.....	75
Figure 24: Decomposition of example hierarchy	86
Figure 25: Loci of communication in Hprep	88
Figure 26: Locus of link contribution data.....	90
Figure 27: Link contribution phases vs. processor for original algorithm.....	93
Figure 28: Typical output from a 64 PE run.....	97
Figure 29: Time spent in link refinement phases vs. processor	98
Figure 30: Time spent in link contribution phases vs. processor	99
Figure 31: Histogram of link connectivity vs. processor number	100
Figure 32: Time spent in a single refine step vs. processor number	101

Figure 33: Performance vs. number of PEs 102

LIST OF TABLES

Table 1:	Opcount metrics of various solvers	22
Table 2:	Solver comparison for various geometries on 1000 patches	23
Table 3:	Operation count metrics for various hierarchical solvers.....	24
Table 4:	Program performance report	58
Table 5:	Situations in parallel link subdivision.....	69
Table 6:	Situations in splitting left link end in parallel.....	80

LIST OF ALGORITHMS

Algorithm 1: Jacobi iteration	18
Algorithm 2: Gauss-Seidel iteration	18
Algorithm 3: Shooting with sorting and ambient	20
Algorithm 4: Preconditioned conjugate gradients	21
Algorithm 5: Computing accelerations hierarchically	29
Algorithm 6: Greengard's fast multipole algorithm	37
Algorithm 7: Hierarchical radiosity	39
Algorithm 8: Alternation of error types	48
Algorithm 9: Airtight occlusion test	54
Algorithm 10: Waist plane construction	55
Algorithm 11: Improved hierarchical radiosity	63
Algorithm 12: All-to-all communications	78
Algorithm 13: Serial link refinement	79
Algorithm 14: Parallel link refinement	83
Algorithm 15: Parallel reheapify	84
Algorithm 16: Serial hierarchical matrix-vector multiply	87
Algorithm 17: Parallel hierarchical vector preparation	89
Algorithm 18: Parallel link contribution accumulation	91
Algorithm 19: Parallel partial product propagation	92
Algorithm 21: Revised parallel link contributions	94
Algorithm 20: Revised parallel link refinement	95
Algorithm 22: Revised parallel reheapify	96

ACKNOWLEDGMENTS

A project such as this necessarily cannot be the product of one heart, mind or hand. Many types of support came together from many sources to make it possible.

To John Gustafson, my friend and mentor, I extend my thanks for hours of excellent discussion. I believe we have learned much from one another, and are richer men for it.

To Denise Hayward, my soul-mate, goes my deepest gratitude for constant encouragement, moral support, and love. Through worse and better, you have been there with me; there's no way to go but ever upward, now.

To those who are seldom recognized and often overlooked, I reserve a special thanks. Peggy Pollock and Nan Ripley have made the wheels turn freely for two years now. Take your well-deserved bows.

This work was funded by Ames Laboratory which is operated for the U. S. Department of Energy by Iowa State University under contract No. W-7405-eng-82. This dissertation has been assigned DOE report number IS-T 1655.

CHAPTER I

INTRODUCTION

In this dissertation, the step-by-step development of a scalable parallel hierarchical radiosity renderer is documented. First, a new look is taken at the traditional radiosity equation, and a new form is presented in which the matrix of linear system coefficients is transformed into a symmetric matrix, thereby simplifying the problem and enabling a new solution technique to be applied. Next, the state-of-the-art hierarchical radiosity methods are examined for their suitability to parallel implementation, and scalability. Significant enhancements are also discovered which both improve their theoretical foundations and improve the images they generate. The resultant hierarchical radiosity algorithm is then examined for sources of parallelism, and for an architectural mapping. Several architectural mappings are discussed. A few key algorithmic changes are suggested during the process of making the algorithm parallel. Next, the performance, efficiency, and scalability of the algorithm are analyzed. The dissertation closes with a discussion of several ideas which have the potential to further enhance the hierarchical radiosity method, or provide an entirely new forum for the application of hierarchical methods.

1.1 Realistic Image Synthesis

1.1.1 General

Realistic image synthesis is a subdiscipline of computer graphics which deals specifically with producing, or *rendering*, images that look as realistic or true-to-life as possible. Applications of realistic image synthesis lie in cinematography, stagecraft, architectural design, simulation, and virtual reality.

On a conceptual level, these images are rendered by modeling the physics of light propagation in a scene as well as is either practical or well-understood. The key problem is to solve for the equilibrium light energy (or power) transfer between all surfaces of a collection of objects (*scene*). This is called a *global illumination* solution because light leaving any surface has the ability to affect the brightness of *all* other surfaces in a scene.

In nature, this problem is a continuous one. That is, a photon of light may arrive or depart from an (essentially) infinite number of positions on a surface. The solution to the continuous radiosity problem is not tractable for any but the simplest configurations of objects. In order to make the problem computationally tractable for complex scenes, a number of simplifying assumptions and approximations are made.

1.1.1.1 Geometric object modeling

In order to render a picture of a scene, one must be able to represent the scene in some way. This is usually done by representing the various objects in a scene with collections of simpler geo-

metric primitives such as polygons, spheres, cylinders, bivariate patches (such as spline patches or Bézier patches), implicit surfaces, etc. However, no physical objects can be accurately modeled with perfect geometric primitives; real objects always have scratches, dents, texture, creases, etc. which are difficult to model. Therefore, a certain permanent loss of realism happens during the modeling process. Much research has been done on how to effectively model physical objects and their properties. Broad areas include constructive solid geometry (CSG), primitives, modeling operations, and reflectance models.

1.1.1.2 Tessellation

Previously, mention was made of modeling a physical object with curved geometric primitives such as spheres and bivariate patches. Some rendering methods, such as ray tracing (to be discussed later), are able to directly render any curved surface so long as certain constraints are met. Other methods, such as radiosity (also to be discussed later), are presently unable to render curved surfaces of any kind. The approach that is generally taken to get around this limitation is to approximate a curved surface with a mesh of polygons. The process of generating a mesh of polygons to approximate a curved surface is called *tessellation*.

Tessellation represents a further degradation of realism in the modeling process. A tessellated sphere is an approximation to a real sphere, which is in turn an approximation to the physical object to be rendered. It is difficult to quantify the error introduced during either modeling process.

1.1.1.3 Discretization

In a physical scene, light propagates from surface to surface in a continuous manner. Every point on every surface may have a different brightness. An essentially infinite number of such points and brightnesses exist in a physical setting. Since a computer is incapable of representing an infinite number of brightnesses in a finite amount of memory, an approximation to the continuous brightness must be made. This is done by discretizing the scene into finite-sized areas, or patches, which will be assumed to be of constant brightness. In this way, rendering is transformed from a continuous problem into a discrete problem suitable for numerical solution on a computer. Again, it is difficult to quantify just how much realism is lost when the problem is discretized. It will become clear in Chapter IV that quantifying discretization error can potentially lead to greatly-improved rendering methods.

1.1.1.4 Light reflection models

Once the physical shapes of primitives in the scene have been modeled, one must then model the physics of light reflection from the surfaces. Physical surfaces have many different modes of light reflection. For example, the surface of a sheet of paper reflects light in a very different way than the surface of a mirror.

Surfaces like a sheet of paper, a pile of powder, velvet, and most other rough surfaces exhibit a light reflection mode called *diffuse* reflection. In this mode, the amount of light energy leaving a surface is independent of both the orientation at which the light arrives, and the orientation at

which it leaves. This is the simplest form of light reflection, and the easiest to model. Under this reflection mode, a surface's reflection properties are completely described by a single scalar value, called *reflectance*, for a given wavelength of incident light. Of course, no physical surface is a perfect diffuse reflector of energy, and thus, error is introduced into the rendering.

More complex is the directionally dependent reflection mode known as *specular* reflection. In specular reflection, the intensity of light leaving a surface depends upon the angle at which it impinges on the surface, and the angle at which it leaves the surface. Thus, the reflectance of a specular surface can be a function of up to four angular variables (azimuth and elevation for incoming and outgoing directions) plus a wavelength variable and a polarization angle. Most real surfaces have significant specular components. Although less error will be introduced into a rendering by attempting to model a surface's specularly, the calculations involved are made much more difficult. Ray tracing has made significant progress modeling arbitrary surface specularly, but only very limited progress has been made with scenes demanding accurate diffuse reflection models [Immel 86, Sillion 91].

Transparent or translucent surfaces not only reflect and absorb energy, but transmit it as well. Transmission, like reflection, can take the form of either diffuse or specular character. Once again, ray tracing accounts for transmission well. The radiosity method has yet to effectively deal with any form of transmission.

There are still other issues involving surface physics which have not been addressed, and can contribute significant error to a realistic rendering. Most obvious is the dependence of surface reflectivity upon the wavelength of the incident radiation. The reflectivity of physical surfaces depends strongly on the wavelength of incident light. Most current reflectance models allow the magnitude of reflected light to change with wavelength, but do not allow the specular angular dependence to vary with wavelength [Westin 92, and others].

Less important effects which are not modeled by any existing reflectance models are fluorescence, polarization, interference, and diffraction. Fluorescence occurs when a surface emits light of a wavelength that is different from the incident light wavelength. This coupling between wavelength bands is completely ignored by all existing renderers. Interference, and diffraction effects are only relevant in the presence of a coherent light source, and are usually of negligible importance. It is likely that drastically different rendering methods will be required to properly account for these phase- and path-dependent effects.

1.1.2 The rendering equation

We will now discuss the mathematical foundations which have been developed for the field of realistic image synthesis. In some cases, such as light reflection models, the mathematics is relatively new [Whitted 80, Cook 82, Kajiya 85, du Montcel 85, He 91, Sillion 91, Westin 92, Ward 92]. In other cases, such as the basic radiosity equation, the theory dates back to radiative heat transfer literature of the 1950's [Sparrow 78, and earlier].

The "Rendering Equation" [Kajiya 86] is a unifying, high-level, continuous expression of the problem to be solved for a realistic or nonrealistic rendering. All known rendering methods, both realistic and nonrealistic, can be derived from this unifying equation via various special cases, and assumptions. The rendering equation is expressed as follows:

$$I(x, x') = g(x, x') \left[\varepsilon(x, x') + \int_S \rho(x, x', x'') I(x', x'') dx'' \right] \quad (1)$$

where:

- x, x', x'' are three separate points in the scene,
- $I(x, x')$ is the light intensity passing from point x' to x ,
- $g(x, x')$ is the visibility function between point x' and x ,
- $\varepsilon(x, x')$ is the light intensity emitted by point x' toward x ,
- $\rho(x, x', x'')$ is the fraction of light scattered by point x' from x'' toward x , and
- S is the hemisphere above x' from which light may arrive.

The integrand of (1) expresses the light transport from x'' to x' to x . This three-point form is necessary to account for the directional reflectivity dependence of specular surfaces. The setup for (1) may be expressed graphically, as shown in Figure 1. The function $g(x, x')$ requires further explanation. This function takes on the value 0 if there is not an unoccluded straight-line path between points x and x' , and the value 1 if the path between them is clear. The rendering equation is only valid for a single wavelength of light at a time. The rendering equation is not amenable to solution by computer in the form presented above, so various simplifications are made. The two most predominant ones are presented below.

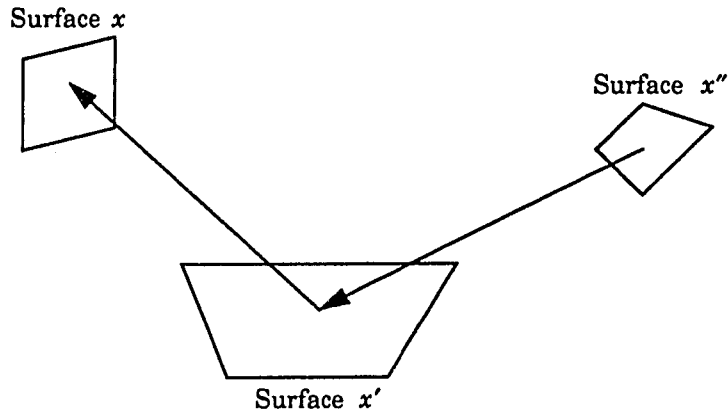


Figure 1: Three-point transport geometry

1.1.3 Ray tracing

Ray tracing is a technique developed by Whitted [Whitted 80] which traces light propagation paths backward from the eye to a light source. It accounts well for diffuse and specular reflection,

but only locally at a patch. Ray tracing is implemented such that all rays are completely independent, and the system has no memory from one ray to the next. Therefore, ray tracing as developed by Whitted does not solve the global illumination problem.

Mathematically, the rendering equation can be written in operator form [Courant 53] as:

$$I = g\epsilon + gMI, \quad (2)$$

where M is the linear operator given by the integral in (1). We may solve (2) in the following way:

$$\begin{aligned} (1 - gM)I &= g\epsilon \\ I &= (1 - gM)^{-1}g\epsilon \\ &= [1 + gM + (gM)^2 + (gM)^3 + \dots]g\epsilon \\ &= g\epsilon + gMg\epsilon + g(Mg)^2\epsilon + g(Mg)^3\epsilon + \dots \end{aligned} \quad (3)$$

Each term in (3) can be thought of as representing one “bounce” of a ray as traced through the scene. Furthermore, only one ray is followed after each bounce, so the operator M is modified to account for this:

$$I = g\epsilon + gM_0g\epsilon + g(M_0g)^2\epsilon + g(M_0g)^3\epsilon + \dots \quad (4)$$

where M_0 is the ray tracing scattering model.

A large body of literature exists for ray tracing. The technique has been extended in many ways since its introduction in 1980. Stochastic sampling techniques have been used to extend the range of optical effects possible with ray tracing. Some of these effects include fuzzy shadows, depth of field, fog, and area light sources [Cook 84, Cook 86, Lee 85]. A great number of geometrical primitives have been analyzed, and numerical methods created to ray trace them [Edwards 82, Kajiya 82, Hanrahan 83, Kajiya 83a, Fontes 84, Kajiya 83b, Kajiya 84, Sederberg 84, van Wijk 84a, van Wijk 84b, Bronsvort 85, Toth 85, Joy 86, Sweeny 86, Burger 89, Giger 89, Hart 89, Kalra 89, Lischinski 90, Nishita 90]. Innovative data structures have been developed to aid in the ray-object intersection operation [Rubin 80, Glassner 84, Coquillart 85, Fujimoto 86, Kay 86, Jansen 86, Naylor 86, Arvo 87, Goldsmith 87, Fussell 88, MacDonald 88, Devillers 89, Montani 90, Thirion 90]. Constructive solid geometry (CSG) has been explored as a way of combining basic primitive types into more complex objects [Cordonnier 85, Kunii 85, Wyvill 85, Gervautz 86, Naylor 86, Wyvill 86, Youssef 86, Arnaldi 87, Cottingham 89, Getto 89, Carter 89, Montani 90]. A method has been developed which “blends” two or more implicit surfaces into a smoothed version of the collection [Blinn 82, Filip 89]. For more general types of objects, deformations may be applied [Barr 84, Barr 86, Sederberg 86].

1.1.4 Radiosity

The method of radiosity rendering was developed from the field of radiative heat transfer [Sparrow 78, Siegel 81]. It attempts to solve for the global balance of energy transfer between objects in the scene. The radiosity technique is correct only for perfectly diffuse surfaces, although

recent improvements have extended the technique, in a crude way, to include some specular effects. The continuous radiosity equation [Cohen 92] is:

$$b(x') = e(x') + \rho(x') \int_x b(x) G(x, x') V(x, x') dA \quad (5)$$

$$G(x, x') = G(x', x) = \frac{\cos\theta_1 \cos\theta_2}{\pi \|x - x'\|^2}$$

where:

- $b(x)$ is the radiant intensity at point x in the scene and is given in units of watts per square meter,
- $e(x)$ is the radiant emissivity at point x in the scene in watts per square meter,
- $\rho(x)$ is the diffuse reflectivity of point x in the scene, is unitless, and represents the fraction of light reflected back into the hemisphere above x ,
- GdA is the differential form factor between x and x' , is unitless, and represents the fraction of light emitted from x that reaches x' ,
- θ_1 is the angle between the surface normal at x and $x' - x$,
- θ_2 is the angle between the surface normal at x' and $x - x'$, and
- $V(x, x')$ is the visibility between points x and x' and is 1 if the points are mutually visible and 0 otherwise.

Equation (5) is a continuous equation, and represents a problem of an infinite number of variables. In order to make it computationally tractable, the environment is discretized into N patches which are assumed to be of constant intensity. The discrete radiosity equation [Goral 84] is:

$$b_i = e_i + \rho_i \sum_{j=1}^N b_j F_{ij} \quad (6)$$

where:

- b_i is the radiant intensity of patch i given in units of watts per square meter,
- e_i is the radiant emissivity of patch i in watts per square meter,
- ρ_i is the reflectivity of patch i , is unitless, and is the fraction of incident light that is reflected back into the hemisphere above the patch, and
- F_{ij} is the "form factor" from patch i to patch j , is unitless, and represents the fraction of light leaving patch i that reaches patch j .

Equation (6) is applied at every patch j in a scene. Thus, a system of linear equations is produced that, when solved, gives a global illumination solution for the radiant intensity at every patch in the scene.

The earliest radiosity renderers formed the dense matrix F_{ij} , and solved it using conventional methods such as Gaussian Elimination with partial pivoting [Goral 84]. Such an approach is of $O(N^3)$ time complexity because of the solution process.

It was later noted that the linear system is diagonally dominant, and therefore amenable to iterative methods such as Jacobi iteration and Gauss-Seidel iteration [Nishita 85, Cohen 85]. Coupling iterative solution to the fact that a radiosity solution need be no more accurate than three or four decimals, the time complexity of the problem is reduced to $O(N^2)$. This reduction in complexity is because iterative solvers of this type are $O(N^2)$ per iteration for a dense system, and take $O(1)$ iterations to converge to the fixed precision criterion. Also, at this point, note that determination of the form factor matrix is of $O(N^2)$ time complexity because, in general, there are $N^2 - N$ nonzero matrix elements. At this point, a further reduction in the radiosity algorithm's time complexity is only possible if *both* the matrix setup and the system solution are improved.

In 1991, Hanrahan, Salzman and Aupperle [Hanrahan 91] applied a hierarchical method similar to the $O(N \log N)$ N -body algorithm [Appel 85] to the construction of the form factor matrix. The method takes advantage of the fact that since the final radiosity solution is only needed to a fixed precision, then the form factors may be *approximated*, in a hierarchical fashion, to a commensurate level of accuracy. Evidence is presented, although a convincing proof is not, that the algorithm approximates the form factor matrix with $O(N)$ blocks. Thus, matrix setup time is reduced in complexity to $O(N)$, and the matrix-vector multiply kernel of common iterative solvers is also reduced to $O(N)$. Thus, the overall time complexity of the hierarchical radiosity algorithm is $O(N)$.

Only one researcher has succeeded, so far, in extending the radiosity method to include specular reflection effects without resorting to ray tracing [Sillion 91]. This highly innovative method uses spherical harmonics to accumulate the directional light intensity variations at each vertex in the scene. The method, however, consumes a tremendous amount of memory relative to the standard diffuse radiosity implementation. This limits the *resolution* of specular effects severely.

1.1.5 Hybrid methods

Ray tracing accounts well for specular reflection, but does not solve for a correct global illumination solution. Radiosity solves for global illumination, but has not shown itself to be easily extended to handle directional lighting effects, including specularity. Several researchers have taken the logical next step, and attempted to create a rendering method which is a fusion of ray tracing and radiosity, combining the best aspects of both methods [Chen 90, Hermitage 90, Immel 86, Jessel 91, Shirley 90, Shirley 91, Sillion 89, Wallace 87, Ward 88].

This approach, however, is not as straightforward as it might seem. One's first instinct might be to use ray tracing to handle specular effects, and radiosity to handle diffuse effects, and provide the global illumination solution. Unfortunately, a global illumination solution is not merely a linear superposition of the two reflection modes. Some success has been enjoyed by these methods, but they remain physically incorrect though pleasing in appearance of results.

1.2 Parallelism in Computer Graphics

Parallelism has come slowly to the computer graphics community. Virtually every research paper done until the late 1980's was based on work done on the VAX 11/780 or similar serial systems. Parallelism was first applied to graphics rendering hardware. It was then applied to the more computationally intensive rendering methods such as ray tracing and radiosity.

1.2.1 Non-realistic

A wide variety of graphics hardware has been developed since the 1980's which uses parallel processing concepts to accelerate drawing of geometric primitives. Notable architectures among these are Pixel-Planes 5, Pixel Machine, and SGI Reality Engine.

1.2.1.1 Pixel-Planes 5

The Pixel-Planes 5 (Pxp15) graphics computer was developed at the University of North Carolina at Chapel Hill by Henry Fuchs et. al. [Fuchs 89] It is a heterogeneous multiprocessor which acts as an attached processor to a high-end workstation. It has applications for real-time simulations, volume rendering for medical imaging, scientific visualization, and realistic image synthesis via the radiosity method. Peak performance is 1 million Phong-shaded triangles per second, 39,000 Gouraud shaded polygons per second, 13,000 smooth shaded spheres per second, or 11,000 shadowed polygons per second. This puts Pxp15 well into the real-time environment for images consisting of a relatively small number of polygonally defined objects.

The Pxp15 system consists of five basic subsystems: graphics processors, renderers, frame buffer, a host interface, and a token ring interconnect. The graphics processors are the floating point math engines of the Pxp15 system. It is their job to perform the 3D geometrical transformations on primitives, and generate rendering requests to the renderers. There may be up to 32 graphics processors in a Pxp15 system. The array of graphics processors effectively comprises a MIMD computer.

Next come the renderers. A renderer is a SIMD array of 128x128 pixel processors, memory, and controller. Renderers are assigned to 128x128 blocks of pixels in the frame buffer to calculate their final contents from requests generated by the Graphics Processors. A SIMD array is a logical choice for pixel operations since such operations are simple, but spread over a large area. The 1280x1024 frame buffer is tiled into 128x128 patches, and a Renderer assigned to each patch. This way, a large number of Renderers may be actively rendering portions of the final image in their local buffers simultaneously. Renderers are built on the concept of logic-enhanced memory chips. A single Renderer chip contains 256 pixel processors, 208 bits of fast SRAM per processor, and one quadratic expression evaluator (QEE). The QEE evaluates the expression $Ax + By + C + Dx^2 + Exy + Fy^2$ with global inputs $A - F$. This is useful for shading curved surfaces and in calculating a spherical radiosity model. Furthermore, each pixel processor has access to an external 4K bits of additional backing store in the form of VRAM. This gives each processor a significant amount of memory to use for Z-buffering or Constructive Solid Geometry. Only when the Renderer is completely finished with its block of pixels are they written to the frame buffer. This

write-once strategy helps to reduce the I/O bottleneck that exists at the frame buffer. There may be up to 16 Renderers in a fully configured Pxp15 system.

The Pxp15 frame buffer is built in a conventional manner from VRAMs and supports a 1280x1024 display refreshed at 72 Hz with 24 bit true color, and a color lookup table. Two token ring nodes are allotted for the frame buffer. The host interface is via programmed I/O.

Connecting the four other components is the ring network. It is an eight channel token ring with an aggregate transfer rate of 160 MWord per second (4 byte words). Access nodes are provided for the Graphics Processors, Renderers, and frame buffer, each with a 20 MW per second bandwidth. In this ring network lies the primary bottleneck in the Pxp15 architecture: it will only support a limited number of graphics processors and renderers, and it cannot be expanded.

The Pxp15 system has many good features working in its favor. Graphics processors are flexible enough that they can implement virtually any graphics algorithm. This is in large part because the graphics processors are a plain vanilla MIMD computer. Also, since the renderers are programmable, a great amount of flexibility is maintained on the pixel level. Pxp15 also enjoys the considerable convenience of being expandable in units of one renderer or graphics processor.

Disadvantages include much degraded performance for shadowed polygons, a complex programming environment (heterogeneous parallel), and a serious scalability problem with the ring network. Although the fast radiosity technique is being developed for Pxp15 and shows promise, the ray-tracing method will gain no benefit from Pxp15's unique architecture.

1.2.1.2 The Pixel Machine

The Pixel Machine was developed at the AT&T Bell Laboratories in Holmdel, New Jersey by Michael Potmesil, Eric M. Hoffert, et. al. [Potmesil 89] It is a homogeneous MIMD image computer with a distributed frame buffer. Its applications lay in the areas of real-time simulations, volume rendering, ray-tracing, and scientific visualization.

DSP32 Digital Signal Processors are used as the computing elements in this novel approach to parallel image computing. The DSP's are organized into two groups: a group of nine DSP's in a pipeline configuration (pipe nodes), and a 2D mesh of 16 - 64 processors to actually operate on pixel data (pixel nodes). Each DSP32 is capable of a maximum floating point performance of 10 MFLOPS (5 MFLOPS of add plus 5 MFLOPS of multiply).

The front-end pipeline of nodes is meant to perform operations that are intrinsically sequential in nature. The input of the pipeline is fed by the Pixel Machine's host computer, and the last node in the pipeline may either send its data to all pixel nodes or back to the host. A second pipeline of nine nodes may be added to the system to form two parallel pipelines, or one 18 node pipeline.

Pixel processors are connected to their four nearest neighbors in a closed torus network. These nodes are used for operations that are intrinsically parallel in nature. Each pixel node has an interleaved portion of the distributed frame buffer accessible to it. In other words, if we have processor (p, q) in an array of $m \times n$ pixel nodes, a processor-space pixel (i, j) is mapped to

screen pixel (x, y) by: $x = mi + p$, and $y = nj + q$. This interleaved scheme is a very effective load balancing mechanism for many classes of parallel graphics algorithms [Fuchs 77, Parke 80, Carter 90].

There are several types of communication paths in the Pixel Machine architecture. Each pixel and pipe node is connected to a global VMEbus. This implements the host-to-node communication path. Pipe nodes are connected by FIFOs. There is also a serial asynchronous link between pipe nodes in the reverse direction of the FIFOs. Finally, there is the already-described connection of pixel nodes to their four nearest neighbors.

One great advantage that the Pixel Machine enjoys is use of off-the-shelf components such as the DSP32. Furthermore, since it is a homogeneous parallel computer, the programmer enjoys a less complex programming environment. The Pixel Machine is remarkably flexible in its ability to implement new graphics algorithms due to its medium grain size. Finally, the distributed interleaved frame buffer approach chosen by the engineers scales by small increments.

Disadvantages include a serial bottleneck in the form of the pipeline. Also, the Pixel Machine does not achieve real-time performance for any application listed due to its considerable overhead to start up an operation. Programmability has been traded off against absolute speed in this architecture.

1.2.1.3 SGI Reality Engine

Recently, Silicon Graphics, Inc. (SGI), introduced their third generation Geometry Pipeline architecture [SGI 92]. It is a dedicated, special-purpose, real-time, non-realistic rendering system which utilizes both MIMD and SIMD parallel processing paradigms to achieve the highest performance of any contemporary system. It has a rich functional capability, including: simple lighting models, smooth polygon shading, Z-buffering, advanced anti-aliasing, fog effects, and the most advanced texture-mapping capabilities available.

The Reality Engine architecture has three major functional blocks. They are the geometry subsystem, the raster subsystem, and the display subsystem. The geometry subsystem utilizes eight advanced RISC microprocessors, operating in MIMD parallel fashion, to perform geometric coordinate transformations. Polygons with more than three vertices are decomposed into two or more triangles by the geometry subsystem, also. Only triangles are allowed because the next subsystem, the raster subsystem, is specifically engineered to render only triangles at high speed.

The raster subsystem is composed of a proprietary arrangement of custom VLSI processors and memory to scan-convert triangles into pixel data and process them into the frame buffer. Five parallel Pixel Generator processors perform the scan-conversion of triangles into pixel data. This pixel data is then optionally routed through the texture processors for texture-mapping. Final pixel values are then stored in the frame buffer.

Finally, the display subsystem takes pixel values from the frame buffer, and generates an analog video signal in any of several standard formats, including two HDTV formats.

1.2.2 Ray tracing

1.2.2.1 LINKS-1

The LINKS-1 system was developed by Hiroshi Deguchi et. al. at Osaka University, Suita, Japan [Deguchi 86]. It was developed specifically to perform high-speed ray tracing. LINKS-1 uses a loosely coupled set of microcomputers, divided into functional groups, to carry out ray tracing.

The LINKS-1 architecture is in the form of a binary tree of processors. These processors are divided into two logical groups: Node Computers and Leaf Computers. Node Computers make up the body of the tree, and the Leaf Computers perform the actual ray tracing. Node Computers and Leaf Computers are connected in a tree structure by the Intercomputer Memory Swapping Unit. This device is able to swap a block of memory between a pair of connected nodes. Node Computers and Leaf Computers are collectively called Unit Computers. Frame buffer data is taken from the Leaf Computers by the Data Collector mechanism and concentrated into the frame buffer for display. The Data Collector represents another potential serial bottleneck in the LINKS-1 system.

Input to the LINKS-1 system consists of a potentially large database of geometrical primitives which comprise a scene to be ray-traced. This database of objects is distributed to the Node Computers and Leaf Computers via the node computers. Note that for even modest object database sizes, each Unit Computer will be able to store only a portion of it. Thus, a large part of the overall architecture is devoted to efficient sharing of the object database among the Leaf Computers. Only the Leaf Computer perform the actual ray-tracing algorithm.

LINKS-1 could be programmed for an image synthesis algorithm other than ray-tracing due to the flexibility and programmability of its Unit Computers. One would expect it only to achieve results similar to that of the Pixel Machine for other graphics algorithms, however, due to the similarity in grain size and frame buffer characteristics between the two systems.

Again, in the LINKS-1 architecture, we see a serial bottleneck that ultimately limits the whole system's performance. LINKS-1 has the distinction of two serial bottlenecks; one at the root of the Data Distributor tree, and one at the frame buffer caused by the Data Collector. Furthermore, the LINKS-1 system does not nearly run at interactive speeds, although it does show significant speedup over previous ray-tracing implementations.

1.2.2.2 Hypercube Ray Tracer

The Hypercube Ray Tracer is a collection of programs developed for the Intel iPSC/2 parallel computer [Carter 89]. It provides common geometric primitives, and an easy-to-use scene description language. Its most important contribution is in the handling of very large object databases. When one wishes to render a scene containing many thousands of primitives at high speed, one encounters a problem early on when dealing with MIMD parallel computers—limited node memory. The naive approach to ray tracing on distributed memory parallel computers has been to duplicate the object database on all processors. With a large object database, there is insufficient memory on every node for this duplication. An object caching scheme is implemented in which the

object database is distributed to permanent “home” locations across the nodes, and copies of objects shuttled between nodes as needed. Since there is a high degree of spatial and temporal coherence in the object database access patterns, this scheme is very effective at dealing with large object databases.

Ray tracing is, at first blush, a trivially parallel application. However, the issues involved with large object databases and load balancing make the problem nontrivial. The hypercube ray tracer deals effectively with these problems and achieves high performance and reasonable scalability at the same time. Load balance is ensured by a master-slave arrangement where a single controlling processor assigns small portions of the image plane to worker processors. When a worker finishes rendering a block of pixels, they are shipped back to the master processor, and a new block of pixels on the image plane is assigned to the worker. These pixels blocks are assigned in a spatially coherent manner to improve the effectiveness of the object database caching scheme.

1.2.2.3 Other work

Others have also mapped the ray tracing algorithm to various existing general-purpose parallel architectures [Badouel 90, Priol 88, Priol 89, Hermitage 90].

1.2.3 Radiosity

Much less work has been done toward applying parallel processing to radiosity solutions. Efforts to date include [Chen 89, Drucker 92, Guitton 91, Purgathofer 91].

One parallel implementation of a radiosity solver on a parallel machine is the SLALOM benchmark [Gustafson 91]. SLALOM is an acronym for Scalable Language-Independent Ames Laboratory One-minute Measurement. It is a fixed-time benchmark which attempts to capture general salient features of scientific computing. A radiosity problem is solved in a right rectangular box to a specified precision. A computer is tasked with solving the largest problem it can, measured in patches, in less than one minute. The problem size, rather than time, is used as the figure of merit for the benchmark.

SLALOM has been ported to run on the following parallel and massively parallel systems: nCUBE 2, MasPar, Intel, Cray, SGI, Myrias, etc. Since the kernel operation of the first version of SLALOM was a dense matrix solver, speed was uniformly high on all parallel machines [Slalom 90]. We will later show this algorithm to be horribly wasteful in terms of the amount of work performed to arrive at a given solution.

1.3 Structure and Aim of This Dissertation

This dissertation tracks the research, development, and implementation of a state-of-the-art, parallel hierarchical radiosity renderer. An analysis of the standard radiosity equation, and methods for solving it, is performed in Chapter II. This analysis shows several interesting things about the development of radiosity solution techniques. It also discloses a previously unexploited symmetry in the linear system of equations. The implications of the new-found symmetry are discussed, and an algorithm is put forward to take advantage of it.

Next, in Chapter III, the new hierarchical methods are described as applied to computer graphics and other scientific disciplines. The two existing hierarchical radiosity methods are reviewed, and commentary made. An effort is made to identify the properties of a physical problem that make it amenable to a hierarchical method of attack.

Chapter IV discusses shortcomings and inconsistencies in the existing methods. Several new improvements to the hierarchical radiosity methods are laid out and discussed. Results from a serial implementation of the new algorithm are discussed and analyzed and compared against existing results.

The new method is examined for sources of parallelism in Chapter V. Sources of parallelism are identified, and decomposition strategies chosen for implementation on an nCUBE 2 parallel computer. Performance and efficiency of the parallel implementation is discussed, together with ways of further improving it.

During the course of this research, many ideas sprang to mind for which time was not available for further pursuit. Many of these future research possibilities are discussed, and a brief summary of work performed is given in Chapter VI.

CHAPTER II

SYMMETRIC RADIOSITY

2.1 Introduction

A new formulation of the radiosity equation is developed which has as its linear equation coefficients a symmetric matrix, rather than the nonsymmetric matrix in all other radiosity papers to date. Such a reformulation has considerable computational advantages, among which are that storage for the form factor matrix is reduced by half and that the more sophisticated Conjugate Gradient solution technique can be brought to bear on the problem. Jacobi, Gauss-Seidel, "Shooting", and Conjugate Gradient solvers are compared by operation count and experiment. The method of Conjugate Gradients is shown to be uniformly superior to the other solver types, with its advantage becoming overwhelming in highly reflective environments. One may observe that the historical evolution of iterative solvers for radiosity problems seems to have increased the solution time, not decreased it.

The formulation of a physical problem is often made to look as simple as possible from a symbolic viewpoint, without regard for computational issues. Since the theory of radiosity predates high-speed computing, its usual formulation is conceptually terse, but computationally wasteful. A major oversight in the analysis of the radiosity problem is pointed out in this chapter, and its implications with regard to storage requirements and the selection of a fast solution strategy.

To date, the methods used to *solve* the radiosity equation have been taken strictly from the backwaters of computational mathematics. Much more sophisticated methods exist than the antiquated Gauss-Seidel iteration (of which shooting is a subtle variant), and should be exploited. In most areas of computational mathematics, an effort is made to find a way of making a problem symmetric; the radiosity problem formulation has stayed nonsymmetric in the literature for *eight* years.

Nowhere has the performance of multiple solver types been objectively compared for the radiosity problem. Arguments are presented here based on operation counts and convergence rates. The conclusion is drawn that solvers appear to have actually gotten *slower* over time.

With the advent of new *hierarchical* radiosity methods [Hanrahan 91, Smits 92, Carter 93b], the time spent solving for patch radiosities has become more significant. Until recently, the majority of time in a radiosity rendering was spent calculating form factors. The new hierarchical methods approximate form factors only to the accuracy needed, thus saving time and magnifying the role of the solver.

2.2 Existing Methods

Since its introduction to the computer graphics community in 1984, the radiosity equation has been presented in a form similar to the following [Goral 84]:

$$b_j = e_j + \rho_j \sum_{i=1}^N b_i F_{ij} \quad (7)$$

where: b_j is the brightness (radiosity) of patch j in W/m^2 ,
 e_j is the emittance of patch j in W/m^2 ,
 ρ_j is the reflectance of patch j ,
 F_{ij} is the form factor from patch i to patch j , and
 N is the number of patches.

Equation (7) may be rewritten in vector-matrix notation in the following way:

$$\mathbf{b} = \mathbf{e} + \mathbf{P}\mathbf{F}\mathbf{b} \quad (8)$$

where: \mathbf{b} , \mathbf{e} are the brightness and emittance vectors,
 $\mathbf{P} = \text{diag}(\rho_j)$, and
 \mathbf{F} is the form factor matrix.

Equation (8) may be arranged into the form of a system of equations to be solved:

$$(\mathbf{I} - \mathbf{P}\mathbf{F})\mathbf{b} = \mathbf{e} \quad (9)$$

where \mathbf{I} is the identity matrix.

$\mathbf{I} - \mathbf{P}\mathbf{F}$ is nonsymmetric in general because $F_{ij} \neq F_{ji}$. Equation (9) was initially solved in a direct fashion using dense LU factorization and backsubstitution with partial pivoting [Goral 84]! It was later shown that $(\mathbf{I} - \mathbf{P}\mathbf{F})$ is diagonally dominant, and thus amenable to a number of iterative solution methods. Pivoting, certainly, is unnecessary unless patches exhibit fluorescence and thus have reflectivities greater than unity. The first iterative method to be applied to the radiosity equation was the Gauss-Seidel technique [Nishita 85, Cohen 85]. Gauss-Seidel and its close relative, Jacobi iteration, remained popular for a number of years [Immel 86, Cohen 86].

The method of "Shooting" was then developed to progress smoothly toward the final solution. In the following discussion, we will assume the shooting, sorting, and ambient version of "progressive radiosity" found in [Cohen 88]. Shooting offered the chance for quick gratification by producing an image of acceptable quality quickly, and then proceeding toward the final solution more slowly. The price one pays for this progressive radiosity approach, in its original form, is:

- Recalculation of form factors for a patch every time it is visited.
- $O(N)$ work between every update to find the patch with the largest unshot radiosity.
- 50% more memory references for the solution update itself.

Note that recalculating the form factors for each patch is only necessary if one does not wish to store the full coupling matrix. In the sequel, we will store the coupling matrix in the interest of

fairness. While this is appropriate if the user wishes to quickly see a rough rendering, it is not appropriate if only the final solution is needed.

2.3 Reformulating the Radiosity Equation

We begin with equation (9), and define the diagonal matrix $A = \text{diag}(A_i)$ where A_i is the area of patch i .

$$(I - PF)\mathbf{b} = \mathbf{e}$$

Now, multiply through on the left by the patch areas.

$$(A - APF)\mathbf{b} = A\mathbf{e} \quad (10)$$

Multiplication of diagonal matrices is commutative, so interchange A and P , giving

$$(A - PAF)\mathbf{b} = A\mathbf{e} \quad (11)$$

Finally, multiply through on the left by P^{-1} to yield the final form.

$$(P^{-1}A - AF)\mathbf{b} = P^{-1}A\mathbf{e} \quad (12)$$

Now, from the definition of form factors, we have the reciprocity relation, $A_i F_{ij} = A_j F_{ji}$. This relation says, among other things, that the product AF is a *symmetric matrix*. Furthermore, since P^{-1} is a diagonal matrix, the product $P^{-1}A$ is also symmetric. Therefore, the whole term $(P^{-1}A - AF)$ is symmetric. Equation (12) is the formulation used in SLALOM since its introduction as a benchmark in 1990 [Gustafson 91].

It has been shown that the radiosity problem can be reformulated as a symmetric system of equations. Discussion of appropriate solution strategies in light of this new observation follows.

2.3.1 Coupling factors

Equation (12) suggests a way to modify the definition of coupling between two patches such that the coupling from patch i to patch j is identical to the coupling from j to i . The traditional definition of the *form factor* from patch i to patch j is:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r_{ij}^2} dA_i dA_j \quad (13)$$

where: ϕ_i and ϕ_j are the angles between surface normals and r_{ij} , and r_{ij} is the vector from one differential area element to another.

The form factor was originally conceived this way because it was thought of as an area-average of differential point-to-area form factors. This quantity is not symmetric with respect to i and j . The product AF simply serves to undo the unneeded division by A_i in the form factor definition. We define:

$$C_{ij} \equiv A_i F_{ij} = \int_{A_i} \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r_{ij}^2} dA_i dA_j \quad (14)$$

Let us call C_{ij} the *coupling factor* between patches i and j . This quantity is conceptually simpler to deal with because it is symmetric with respect to i and j . Since a coupling factor has units of area, it also has the convenient property that when multiplied by a radiosity value, it gives energy impinging on the coupled-to patch directly. Thus, equations written using the coupling factor matrix C are energy balance equations, not energy density balance equations. Equation (12) can be simplified to:

$$(P^{-1}A - C)\mathbf{b} = P^{-1}A\mathbf{e} \quad (15)$$

2.4 Solution Techniques

Now that there is a symmetric system to solve, a number of new options are open. Direct solution methods may still be used, but iterative methods are more effective.

2.4.1 Direct solution

Cholesky factorization (LDL^T) rather than LU factorization may be used. This cuts the solution time, number of operations, memory references, and memory use in half when compared to LU factorization. Cholesky factorization, however, is still $O(N^3)$ in the number of equations (patches).

Most researchers have recognized that the solution to a radiosity problem is seldom needed (or correct, in any realistic sense!) to more than three or four decimals. Even if the solution to a radiosity problem were needed to fifteen or more decimals, direct methods would lose for some value of N . This has been proved based on a condition number bound [Bjørstad 91b]. Even for highly-reflective scenes, iterative methods will win for only a few hundred patches. Cholesky factorization, like LU , is therefore of little more than academic interest for radiosity problems.

2.4.2 Simple iterative techniques

Jacobi iteration, Gauss-Seidel iteration, and Shooting are all still possible with the symmetric formulation. The advantages of the symmetric system, however, are more modest than for direct solution. Storage space for the matrix of coefficients is still cut in half, but the number of operations per iteration for these schemes remains the same. For the following analyses, the symmetric radiosity equation (12) is rewritten as:

$$(D + S)\tilde{\mathbf{x}} = \mathbf{v} \quad (16)$$

Where:

- D is the diagonal matrix $P^{-1}A$,
- S is the symmetric matrix $-AF$,
- $\tilde{\mathbf{x}}$ is the solution estimate to \mathbf{b} ,
- \mathbf{v} is the right-hand-side $P^{-1}A\mathbf{e}$.

Usually, the matrix S will have some degree of sparsity caused by coplanar patches, occluded patches, or patches facing away from one another which have zero coupling (patches which cannot "see" one another.)

2.4.2.1 Jacobi iteration

The simplest iterative scheme we consider here is the Jacobi iteration. This iteration updates all solution variables as a batch. It can be written as follows in algorithmic form:

```

k = 0, m = Sx̃₀
do {
  x̃ = -D⁻¹m + v
  m = Sx̃
  r = m + Dx̃ - v {Residual}
} while (||r||∞ > ε ||D + S||∞ ||x̃||∞)

```

Algorithm 1: Jacobi iteration

An examination of Algorithm 1 reveals that it takes $2N^2 + 4N$ floating-point operations per iteration plus $2N^2 - 3N$ floating-point operations for setup. Note that in the preceding operation counts, multiplication, addition, and subtractions is counted as one floating-point operation and divide and square root are counted as four. It is guaranteed to converge for diagonally dominant systems, and its convergence rate is related to the spectral radius of $\rho(D^{-1}S)$. The closer $\rho(D^{-1}S)$ is to zero, the more quickly Jacobi iteration converges. A small ρ corresponds to scenes with dark surfaces.

2.4.2.2 Gauss-Seidel iteration

Slightly more complicated is the Gauss-Seidel iteration. Instead of updating all variables as a batch, it updates one at a time, and then uses the value just computed when updating subsequent variables. It can be written as shown in Algorithm 2.

```

k = 0
do {
  for i = 1 to N
    α = xᵢ
    xᵢ = (vᵢ - ∑_{j=1}^{N, j≠i} sᵢⱼxⱼ) / dᵢ
    rᵢ = dᵢ(xᵢ - α) {Previous resid.}
} while (||r||∞ > ε ||D + S||∞ ||x̃||∞)

```

Algorithm 2: Gauss-Seidel iteration

Algorithm 2 is a slight variant on Gauss-Seidel that evaluates the residual from the previous iteration. Although this different residual check will cause Algorithm 2 to proceed one too many iterations, it cuts in half the amount of $O(N^2)$ work. Algorithm 2 requires $2N^2 + 3N$ floating-point

operations with no setup overhead. Gauss-Seidel has a convergence rate that is related to $\rho((D+L)^{-1}L^T)$, where L is the lower triangular part of the symmetric matrix S .

2.4.2.3 Progressive radiosity (shooting)

The method of “Shooting” is a variant on the Gauss-Seidel iteration proposed by Cohen [Cohen 88]. It progressively redistributes “unshot radiosity” through the scene according to which patch has the greatest amount of unshot radiosity accumulated.

At each step, the patch in the scene with the greatest amount of unshot radiosity is selected. The unshot radiosity belonging to this patch is redistributed to all other patches in the scene, updating their solutions and unshot radiosities. This process is repeated until convergence is reached.

An added optimization is also formulated to deal with ambient light in the scene. A constant radiosity is added to all patch radiosities in the scene in an attempt to reduce the RMS error in the answer at each step. The Shooting algorithm can be expressed as follows:

Algorithm 3 takes $6N^2 + 3N$ operations per iteration. Unlike Jacobi or Gauss-Seidel, there is no way to reuse the operations in the kernel to compute a residual. Therefore, a separate step involving $2N^2$ operations is necessary to calculate a true residual. Other convergence tests, such as the largest unshot radiosity or the difference between two consecutive solution estimates, could be used, but they are subject to catastrophic failure. In a highly reflective scene, the true solution is approached slowly, and an *ad hoc* convergence test might terminate the iteration prematurely while still far from the desired accuracy (even to the human eye!). Note that the human eye tends to forgive gross global illumination errors, while emphasizing small local errors. Since Shooting updates the solution variables in a data-dependent order, a convergence analysis is difficult. Experimental comparison will be presented in the section titled “A Practical Comparison” on page 21.

2.4.3 Other iterative techniques

Various schemes exist for accelerating the convergence of methods such as Gauss-Seidel, such as symmetric successive over-relaxation (SSOR), and the Chebyshev Semi-Iterative method. Both of these methods, however, assume that certain acceleration constants are available or computable at solution time. In general, the optimal values for these constants are difficult to obtain except for certain structured problems.

As there is little structure in the general radiosity matrix, determining optimal values for the acceleration constants is probably more expensive than a non-accelerated solution technique. This is because it would require an eigenvalue analysis which is as difficult a problem as the solution itself. If the values for the acceleration constants are sufficiently wrong, then, the “accelerated” solution may proceed more slowly than the unaccelerated version. A large literature exists for *estimating* the acceleration constants, but we have not chosen to pursue such an analysis. In the next section, we will show that the method of conjugate gradients is both simple and fast, and have concentrated our efforts there.

```

 $\tilde{x} = v$  (First guess solution)
 $\Delta x = v$  (Unshot radiosity)
 $\tau = \sum_{i=1}^N A_i$  (Total area)
 $\rho_{ave} = \frac{1}{\tau} \sum_{i=1}^N \rho_i A_i$  (Average reflectivity)
 $R = \frac{1}{1 - \rho_{ave}}$  (Interreflection factor)
do {
  for  $k = 1$  to  $N$ 
    Select  $i$  s.t.  $\Delta x_i$  is maximal
    for  $j = 1$  to  $N$ ,  $i \neq j$ .
       $\Delta rad = (\rho_j \Delta x_i s_{ij}) / A_j$ 
       $\Delta x_j = \Delta x_j + \Delta rad$ 
       $x_j = x_j + \Delta rad$ 
       $\Delta x_i = 0$ 
     $\Delta ambient = \frac{R}{\tau} \sum_{j=1}^N \Delta x_j A_j$  (Ambient light)
    for  $j = 1$  to  $N$  (Improved sol'n)
       $m_j = x_j + \rho_j \Delta ambient$ 
     $r = (D + S)m - v$  (Residual)
  } while ( $\|r\|_{\infty} > \epsilon \|D + S\|_{\infty} \|m\|_{\infty}$ )
  for  $j = 1$  to  $N$ 
     $x_j = x_j + \rho_j \Delta ambient$ 

```

Algorithm 3: Shooting with sorting and ambient

2.4.4 Method of conjugate gradients

An important implication of the symmetric radiosity equation is the ability to apply more sophisticated methods to its solution, such as the method of conjugate gradients [Golub 89]. This method was first applied to the SLALOM benchmark by Bjørstad and Boman [Bjørstad 91a]. They prove that, given bounds on the maximum reflectivity in the scene ($\rho \leq \rho_{max} < 1$), the CG solver will always be better than a direct solver as N grows large [Bjørstad 91b].

The method of conjugate gradients (CG) is not actually an iterative method, but a way of systematically constraining the solution in residual space. This powerful method exploits the direction in which the solution estimate changes in N -space to choose a better path toward the solution. Furthermore, it is guaranteed to converge to the *exact* solution in N iterations (assuming

exact arithmetic), where N is the number of unknowns. In practice, however, CG converges too quickly to be allowed to proceed for N iterations.

The convergence rate of the conjugate method is related to the condition number of the matrix being iterated upon. The closer the condition number is to unity, the more quickly the method converges. Preconditioning is a way of accelerating the conjugate gradient method by solving a similar problem whose condition number is closer to unity. We have chosen to use a simple diagonal preconditioner with our CG algorithm.

```

 $k = 0, \tilde{x} = 0, r = v$ 
do {
   $z = D^{-1}r$ 
   $\gamma_1 = r^T z$ 
  if  $k = 0$  then
     $\beta = 0$ 
  else
     $\beta = \gamma_1 / \gamma_0$ 
  end if
   $p = z + \beta p$ 
   $\alpha = \frac{\gamma_1}{p^T (D + S) p}$ 
   $\tilde{x} = \tilde{x} + \alpha p$ 
   $r = r - \alpha (D + S) p$ 
   $\gamma_0 = \gamma_1$ 
   $k = k + 1$ 
} while ( $\|r\|_\infty > \epsilon \|D + S\|_\infty \|\tilde{x}\|_\infty$ )

```

Algorithm 4: Preconditioned conjugate gradients

Algorithm 4 takes $2N^2 + 15N$ floating-point operations per iteration with no setup overhead. As with Jacobi iteration, only one matrix-vector multiply is needed per iteration.

2.5 A Practical Comparison

An early version of the SLALOM benchmark has been modified in order to objectively evaluate the effectiveness of various solution methods. The original SLALOM benchmark solves a radiosity problem in a six-sided, right, rectangular box. This type of simple enclosure is sometimes called a ‘‘Cornell box’’ in honor of the landmark paper [Goral 84]. Each face may have different reflectivity and emissivity, and is subdivided into a regular grid of subpatches in order to more accurately capture the change in light intensity across the face.

Jacobi, Gauss-Seidel, ‘‘Shooting,’’ and Conjugate Gradient solvers are implemented, and their results presented. The Cornell box used for this experiment is 13.5 by 9 by 8 units, with area-weighted average reflectivities ranging from 0.372 to 0.902. Some explanation is in order here for

exactly what is meant by an “iteration” with respect to Shooting. In CG, Jacobi, and Gauss-Seidel, an iteration means an update to each variable in the system. For purposes of comparison, one iteration of Shooting is defined to be N solution updates, whether they are different variables or not. This gives a uniform scale of comparison for all three methods, and does not penalize the Shooting method.

Table 1 summarizes the number of floating-point operations per iteration for each of the four solver types implemented in terms of the number of variables, N . Every effort was made to fairly assess each algorithm’s requirements, and tune them so that no operations were wasted.

Table 1: Opcount metrics of various solvers

Solver Type	Operations per Iteration
Jacobi	$2N^2 + 4N$
Gauss-Seidel	$2N^2 + 3N$
Shooting	$6N^2 + 3N$
Conjugate Gradient	$2N^2 + 15N$

Figure 2 presents the number of iterations each of the solver types required to converge for a range of problem sizes. The convergence criterion was four decimals of relative accuracy. Great care was taken to ensure that each solver used exactly the same convergence criteria so comparison of iteration counts would be fair. Convergence curves for four and eight decimals of accuracy are presented, although only the four-decimal plot is of practical interest.

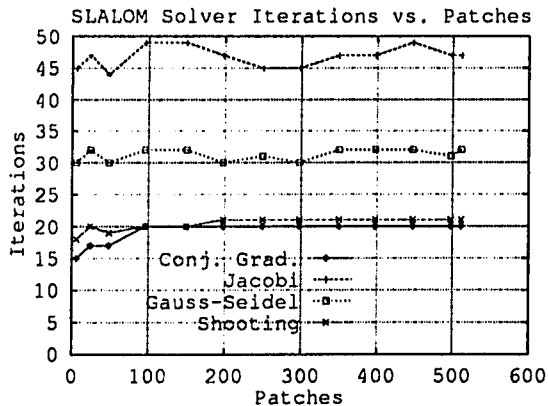


Figure 2: Solver iteration counts

One will immediately notice from Figure 2 that the numbers of iterations of Conjugate Gradient and Shooting required for four decimals of accuracy are comparable. Data from Table 1, however, puts this in a different light. A small number of iterations is not the measure of goodness for a solution technique, but rather the time it takes to solve a given system. The shooting method

requires *three times* the number of floating-point operations per iteration that conjugate gradient does! Thus, Shooting will take about three times as long to converge as CG even if their convergence rates are the same. Later, it will become apparent that CG has a much faster convergence rate than Shooting.

Total solution time for the three solvers on various problem sizes is shown in Figure 3. The CG solver can be seen to be approximately five times faster than Shooting for this problem. Collected in Table 2 are timing comparisons of Jacobi, Gauss-Seidel and Shooting against Conjugate Gradient for a variety of geometries and average reflectivities. The problem size is 1000 patches in each case. The results show that CG is superior to Jacobi, GS and Shooting for all configurations tested, with its advantage becoming overwhelming with higher average reflectivity. Thus, the convergence rate for CG is relatively unaffected by the average reflectivity of the scene, while GS and Shooting suffer badly with increasing reflectivity.

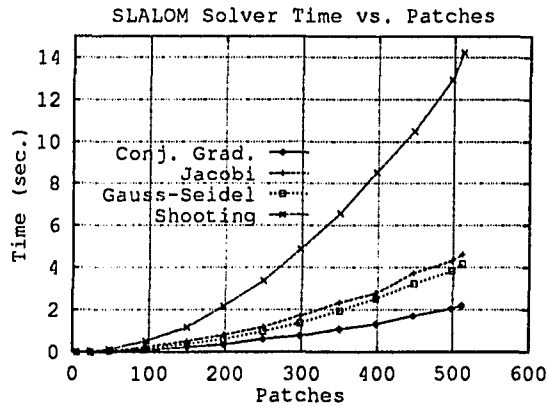


Figure 3: Solver time

Table 2: Solver comparison for various geometries on 1000 patches

Geometry (Cornell Box)	Average ρ	$\frac{T_{JAC}}{T_{CG}}$	$\frac{T_{GS}}{T_{CG}}$	$\frac{T_{Shoot}}{T_{CG}}$
10 x 10 x 10	0.411	1.21	1.21	3.84
	0.617	1.75	1.63	5.05
	0.738	3.08	2.56	10.67
	0.893	6.98	5.70	20.15
13.5 x 9 x 8	0.444	1.32	1.40	3.44
	0.659	1.91	1.81	6.19
	0.760	3.03	2.67	11.66
	0.895	7.06	5.79	25.49

Table 2: Solver comparison for various geometries on 1000 patches (cont'd)

Geometry (Cornell Box)	Average ρ	$\frac{T_{JAC}}{T_{CG}}$	$\frac{T_{GS}}{T_{CG}}$	$\frac{T_{Shoot}}{T_{CG}}$
2 x 2 x 10	0.372	1.06	1.24	1.86
	0.543	1.28	1.35	2.60
	0.690	2.10	1.95	5.74
	0.902	5.34	4.61	16.23

Another striking observation can be made from the data in Table 2. The performance of each solver type is exactly the opposite of that which would be implied by historical usage! Early radiosity papers used Jacobi iteration to solve their systems. Work then proceeded to Gauss-Seidel iteration, and Shooting methods. Apparently, the added complexity of the more sophisticated iterative schemes more than offset any gains that might have been made by improved convergence rates, thus *increasing* the solution time! Perhaps the choice was driven by apparent image quality instead of analysis of the error.

It is interesting to note that the original formulation of the Shooting technique attempts to take specific advantage of average reflectivity to compute a uniform, ambient illumination. This ambient term is added to the solution estimate at each iteration to give an improved solution. Even this specific optimization does not help the Shooting technique cope well with highly reflective environments. Indeed, the Shooting technique seems to work best for dim scenes with low average reflectivity—a situation where radiosity methods are not called for at all!

2.6 Applicability to Hierarchical Methods

With the advent of the hierarchical radiosity algorithm [Hanrahan 91], new areas of investigation are open. Hanrahan *et al.* promote the use of the Shooting method for solving a hierarchical system. They justify this by pointing out that each patch is linked to $O(1)$ other patches in the scene, and therefore a shooting step is very little work.

Although this is true, the conjugate gradient algorithm can benefit just as much from the hierarchical nature of the problem; the matrix-vector multiply kernel of CG can be performed in $O(N)$ time instead of $O(N^2)$ time. The convergence properties of these two methods will be the same regardless of whether the matrix is represented densely or hierarchically.

Hierarchical matrix-vector multiply takes $4kN + 12N$ floating-point operations, where n is the number of leaf patches in the hierarchy, and k is the average number of links per patch. Experience has shown us that k is typically about 10. Reanalyzing operation counts for Algorithm 1 through Algorithm 4 gives us Table 3.

Table 3: Operation count metrics for various hierarchical solvers

Solver Type	Operations per Iteration
Jacobi	$4kN + 19N$

Table 3: Operation count metrics for various hierarchical solvers

Solver Type	Operations per Iteration
Shooting	$8kN + 15N$
Conjugate Gradient	$4kN + 25N$

Once again, we see CG and Jacobi tied for the least number of operations per iteration. Knowing that the convergence rates of the various solvers are unchanged, we may conclude that CG would once again be the method of choice.

2.7 Summary

We have shown how the diffuse radiosity problem can be reformulated in terms of a symmetric system of linear equations. All previous formulations of the radiosity problem have been non-symmetric, and thus, could not benefit from decreased storage requirements, decreased memory references, and advanced solution techniques.

The method of Conjugate Gradients has been applied to a Cornell box radiosity problem, and its performance compared to that of the progressive radiosity Shooting technique, Gauss-Seidel iteration, and Jacobi iteration for a variety of geometric configurations and average reflectivities. The Conjugate Gradient technique is uniformly superior to all three other methods, with its advantage becoming very pronounced in scenes with high average reflectivity.

CHAPTER III

HIERARCHICAL METHODS

3.1 Introduction to Hierarchical Methods

A hierarchical algorithm is one which exploits a pre-specified accuracy criterion to reduce the amount of calculation in the solution of a problem. The method will solve at multiple resolutions to avoid excess work on coarse resolutions, and thus reduce the total amount of computational work.

This chapter's purpose is twofold. First, the overall philosophy of hierarchical methods is explored and discussed. Then, in order to motivate the previous discussion, five examples of hierarchical methods are discussed in modest detail. Two of these algorithms are hierarchical radioactivity algorithms—the only two in the literature. The other three are astrophysical N -body simulation programs.

3.2 Hierarchical N -body Methods

Hierarchical methods, as defined here, started with an improved algorithm for calculating the total forces acting on a set of mutually gravitating bodies or particles [Appel 85]. This type of problem, called the N -body problem, is of intense interest in cosmology where there exist many open questions about the state of the universe such as, "Is the universe open or closed?" To conduct meaningful theoretical experiments, simulations of thousands or even millions of gravitating bodies (particles) must be modeled. A brute-force algorithm for calculating the exact force on each particle consumes $O(N^2)$ time. This is because all N particles in the system interact with the other $N - 1$ particles in a non-trivial manner.

3.2.1 Appel's N -body algorithm

Appel proposed a method which approximates the gravitational interaction between two particles, a single particle and a distant clump of particles, or between two distant clumps of particles [Appel 85]. By only computing the force on a particle or clump to a specified level of accuracy, and applying this clumping recursively, the time complexity of the calculation was reduced. A proof of this time complexity was not provided, but a conservative argument was given by Appel to support a time complexity of $O(N \log N)$.

3.2.1.1 Bounding the interaction error

The gravitational interaction between a particle and a clump of particles can be approximated by regarding the clump of particles as a single point mass. This is called the *monopole approximation*. Consider the arrangement shown in Figure 4. Two point masses m_1 and m_2 are shown together with an "observing" particle, o . The two point masses are no further than $|dr|$ from their center of mass, c . The acceleration on the observing particle o may be written as

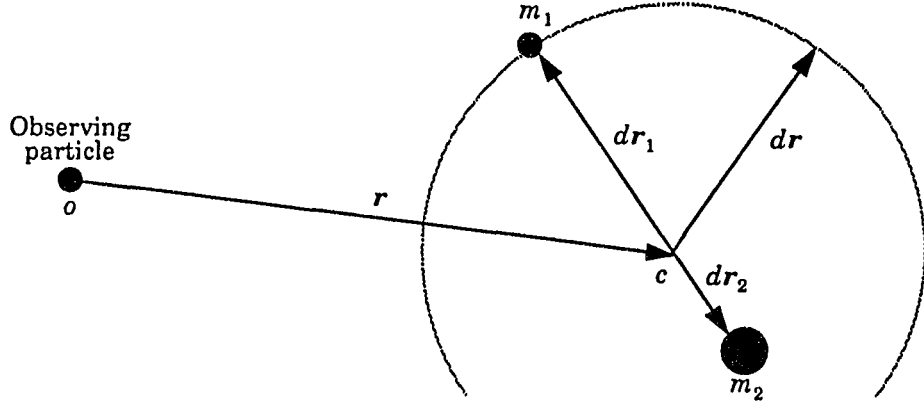


Figure 4: Monopole approximation

Theorem: The monopole approximation, (17), correctly estimates the force on a single observing particle due to two other particles to within $O(|dr|^2)$.

$$\mathbf{a} = \frac{Gm_1(\mathbf{r} + d\mathbf{r}_1)}{|\mathbf{r} + d\mathbf{r}_1|^3} + \frac{Gm_2(\mathbf{r} + d\mathbf{r}_2)}{|\mathbf{r} + d\mathbf{r}_2|^3} = \frac{G(m_1 + m_2)\mathbf{r}}{|\mathbf{r}|^3} + O(|dr|^2). \quad (17)$$

Proof: The center-of-mass of the system in Figure 4 satisfies,

$$m_1 d\mathbf{r}_1 + m_2 d\mathbf{r}_2 = 0. \quad (18)$$

We will use the vector form of Taylor's formula,

$$f(\mathbf{x}_0 + \mathbf{h}) = f(\mathbf{x}_0) + \mathbf{h} \cdot \nabla f(\mathbf{x}_0) + O(|\mathbf{h}|^2), \quad (19)$$

in the sequel. First, we form the Taylor series expansion,

$$f(\mathbf{r} + \mathbf{h}) = \frac{1}{|\mathbf{r} + \mathbf{h}|^3} = \frac{1}{|\mathbf{r}|^3} - \frac{3(\mathbf{h} \cdot \mathbf{r})}{|\mathbf{r}|^5} + O(|\mathbf{h}|^2). \quad (20)$$

We now use (20) to expand the middle term of (17),

$$\begin{aligned} \frac{\mathbf{a}}{G} &= \frac{m_1 \mathbf{r}}{|\mathbf{r}|^3} - \frac{3m_1 \mathbf{r} (d\mathbf{r}_1 \cdot \mathbf{r})}{|\mathbf{r}|^5} + \frac{m_1 d\mathbf{r}_1}{|\mathbf{r}|^3} - \frac{3m_1 d\mathbf{r} (d\mathbf{r}_1 \cdot \mathbf{r})}{|\mathbf{r}|^5} + O(|d\mathbf{r}_1|^2) \\ &+ \frac{m_2 \mathbf{r}}{|\mathbf{r}|^3} - \frac{3m_2 \mathbf{r} (d\mathbf{r}_2 \cdot \mathbf{r})}{|\mathbf{r}|^5} + \frac{m_2 d\mathbf{r}_2}{|\mathbf{r}|^3} - \frac{3m_2 d\mathbf{r} (d\mathbf{r}_2 \cdot \mathbf{r})}{|\mathbf{r}|^5} + O(|d\mathbf{r}_2|^2). \end{aligned} \quad (21)$$

By (18), we may eliminate terms 3 and 8 from (21). Terms 2 and 7 may also be eliminated using (18). Also, terms 4 and 9 are of $O(|d\mathbf{r}|^2)$, therefore,

$$\ddot{\mathbf{a}} = \frac{G(m_1 + m_2) \mathbf{r}}{|\mathbf{r}|^3} + O(|d\mathbf{r}|^2) \blacksquare$$

Thus, the monopole approximation can be used to approximate the acceleration on a particle due to a clump to within order $|d\mathbf{r}|^2$. Similarly, the acceleration on every particle in a clump due to another clump may be approximated to within order $(|d\mathbf{r}|/|r_{min}|)^2$, where $|r_{min}|$ is the minimum distance between the two clumps.

3.2.1.2 The algorithm

The first step in executing Appel's fast N -body algorithm is to construct a binary k - d tree above the N given particles. This has the effect of spatially clumping nearby particles into adjacent subtrees. Interior nodes are tagged with the center of mass for all particles in the subtree. Interior nodes also contain the radius of a sphere which will enclose all particles in the subtree. Thus, clump-to-clump interactions are equivalent to applying the monopole approximation between two interior nodes in the tree. Appel gives the following algorithm for computing the acceleration on all particles in the system.

Algorithm 5 traverses the hierarchy of particle clumps, and evaluates acceleration contributions at the first place where the error criterion is satisfied. Note that if δ is set to zero, the `TWO NODE` procedure recurs all the way down to the leaf level, and calculates all N^2 interactions. If δ is sufficiently greater than zero, then recursion will terminate before reaching the leaf level, and calculations will be saved. The approximation made at this higher level will be accurate to a relative accuracy of $O(\delta)$.

3.2.1.3 Analysis of time complexity

Suppose a particle X is surrounded with a series of spherical shells as shown in Figure 5. A shell of inside radius r is defined to have a thickness of $\delta \cdot r$. Consider one of these shells of radius r and thickness $\delta \cdot r$. The shell is filled with clumps of diameter $\delta \cdot r$. All these clumps satisfy the error criterion set forth above. The number of such clumps that may be placed in the shell is

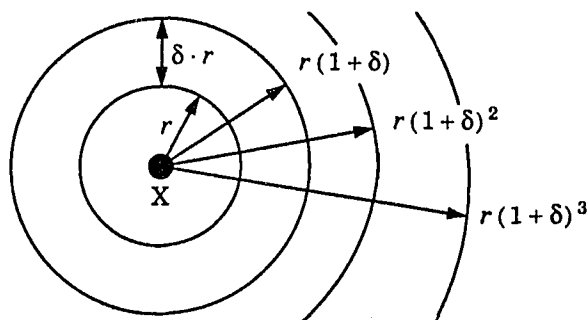


Figure 5: Shell structure about X

```

{ Compute all acceleration contributions for }
{ all nodes in the tree rooted at clump B. }
ComputeAccel(B)
{
  if B is a nontrivial clump {
    ComputeAccel(Bleft)
    ComputeAccel(Bright)
    TwoNode(Bleft, Bright)
  }
}

{ Compute the gravitational interaction between }
{ clumps A and B. If A and B do not satisfy the }
{ error criterion, proceed further down tree. }
TwoNode(A, B)
{
  d = vector from A to B
  d = magnitude of d
  drA = diameter of sphere around clump A
  drB = diameter of sphere around clump B
  if (drA/d > δ) and (drA > drB) {
    TwoNode(Aleft, B)
    TwoNode(Aright, B)
  }
  else if (drB/d > δ) {
    TwoNode(A, Bleft)
    TwoNode(A, Bright)
  }
  else {
    AccA = AccA + GmBd/d3
    AccB = AccB - GmAd/d3
  }
}

```

Algorithm 5: Computing accelerations hierarchically

$$\frac{\text{area of shell}}{\text{area of clump}} = \frac{4\pi r^2}{4\pi \left(\frac{\delta \cdot r}{2}\right)^2} = \frac{4}{\delta^2} \text{ clumps per shell.} \quad (22)$$

This follows by projecting the clumps' silhouettes onto the surface of the shell. The shells about X are arranged so that the expected number of particles inside the smallest sphere is 1, and the expected number of particles inside the largest sphere is N , the number of particles in the system. The radius of the smallest sphere is defined to be r . Therefore, the radius of the largest sphere will be $r(1 + \delta)^k$ for some $k \in I$. Assuming a uniform distribution of particles through space, the expected number of particles that a given sphere will enclose is directly proportional to its volume. Therefore, the ratio of the radii of the largest and smallest spheres will be

$$\frac{r_{\text{largest}}}{r_{\text{smallest}}} = \left(\frac{\left(\frac{3V_{\text{largest}}}{4\pi} \right)}{\left(\frac{3V_{\text{smallest}}}{4\pi} \right)} \right)^{1/3} = \left(\frac{3N}{3} \right)^{1/3} = N^{1/3}. \quad (23)$$

This ratio of radii may be derived another way by using the sizes of the shells:

$$\frac{r_{\text{largest}}}{r_{\text{smallest}}} = \frac{r(1+\delta)^k}{r} = (1+\delta)^k. \quad (24)$$

Equating the right hand sides of (23) and (24), we obtain the following:

$$\begin{aligned} N^{1/3} &= (1+\delta)^k \\ \frac{1}{3} \log N &= k \log(1+\delta) \\ k &= \left\lceil \frac{\log N}{3 \log(1+\delta)} \right\rceil, \end{aligned} \quad (25)$$

the number of shells. Next, the number of floating-point operations (flops) necessary to update particle X is proportional to the number of shells times the number of clumps per shell times the number of flops for one clump, which is (25) times (22):

$$\frac{\text{flops}}{\text{update}} \propto \frac{\log N}{3 \log(1+\delta)} \frac{4}{\delta^2} = \frac{4 \log N}{3 \delta^2 \log(1+\delta)}. \quad (26)$$

Note that the number of clumps per shell in (22) is independent of the radius of the shell. Finally, all N particles in the system must be updated in accordance with (26), so the total number of operations necessary to update all particles in the system is bounded from above by,

$$N \frac{\text{flops}}{\text{update}} = \frac{4N \log N}{3 \delta^2 \log(1+\delta)} = O(N \log N), \quad (27)$$

assuming a constant δ , the measure of precision.

The complexity bound given in (27) is conservative because in reality, Appel's algorithm does not evaluate the acceleration on all particles one at a time. Rather, there are clump-to-clump interactions which take the place of many particle-to-clump interactions (refer to Algorithm 5).

A less conservative argument would involve analyzing the update of all N particles at once, not just one at a time. Consider all the interactions computed in Algorithm 5. An interaction is computed between two clumps only when the ratio dr/r is of order δ . If the ratio were larger, further recursion would have taken place in the `TwoNode` procedure, and two or more interactions with smaller ratios would have been computed. The ratio will never be smaller than $O(\delta)$ because the interaction would have been computed at a higher level, with a correspondingly larger ratio of dr/r . Thus, by (22), there are a *constant* number of interactions between a given clump, and clumps of the same size. Stated another way, there are a *constant* number of links to other clumps or particles at every node in the hierarchy. The total number of nodes in a binary tree of N parti-

cles is $2N - 1$. Therefore, the total amount of computation involved in updating all particles is $k(2N - 1)$, or $O(N)$. It is startling that this argument was not given in either [Appel 85] or [Barnes 86]. It appears Appel was first to discover the $O(N)$ method, but was not so credited because his proof was conservative. Greengard later proved $O(N)$ behavior for a slightly different approach.

3.2.2 Barnes and Hut's N -body algorithm

A variation on Appel's algorithm uses a spatial octree, rather than Appel's k - d tree, to partition particles into clumps [Barnes 86]. This less flexible spatial partitioning has the benefit of simplicity, and allows a more rigorous error analysis. Again, an argument is given to support a time complexity of $O(N \log N)$.

3.2.3 Greengard's fast multipole algorithm

By further examining this strategy, Greengard and Rokhlin [Greengard 87, Greengard 88] lowered the time complexity of the N -body problem to $O(N)$. Although their error analysis and theoretical development is mostly concerned with potential fields, the underlying methodology transfers readily to other application areas.

Greengard has one major discovery set forth in his dissertation. He develops the theory of multipole expansions for potential fields. This discovery allows a preset error criterion to dictate how accurately to approximate the interaction between two clusters of particles. The algorithms of Appel and Barnes obtain higher accuracy by restricting which clumps may interact with one another. Greengard's algorithm, on the other hand, maintains a fixed rule for which clumps may interact, but uses the so-called multipole expansion to approximate the interactions to the desired precision.

Instead of dealing directly with accelerations or forces, Greengard instead chooses to approximate the *potential*, rather than the *force* field. Potential is a scalar field whose value at any point in space is the relative potential energy elicited by all other masses in the system. The gradient of the potential field is a vector field called the force (or acceleration) field. Its direction at any point in space points in the direction of steepest decrease in the potential field. Mathematically, this is expressed as follows:

$$\mathbf{F}(\mathbf{x}) = -\nabla \Phi(\mathbf{x}), \quad (28)$$

where: $\mathbf{F}(\mathbf{x})$ is the vector force field at point \mathbf{x} , and
 $\Phi(\mathbf{x})$ is the potential at point \mathbf{x} .

For problems in two dimensions, the potential at point \mathbf{x} due to a unit mass or charge at point \mathbf{x}_0 is given by,

$$\phi_{\mathbf{x}_0}(\mathbf{x}) = -\log(|\mathbf{x} - \mathbf{x}_0|). \quad (29)$$

Substituting (29) into (28), we have the expression for the gravitational force field in two dimensions,

$$F_{x_0}(\mathbf{x}) = \frac{1}{|\mathbf{x} - \mathbf{x}_0|} \nabla |\mathbf{x} - \mathbf{x}_0| = \frac{\mathbf{x} - \mathbf{x}_0}{|\mathbf{x} - \mathbf{x}_0|^2}. \quad (30)$$

For problems in three dimensions, the potential for a unit mass is given by,

$$\phi_{x_0}(\mathbf{x}) = \frac{1}{|\mathbf{x} - \mathbf{x}_0|}. \quad (31)$$

Substituting (31) into (28), we have the expression for the gravitational force field in three dimensions,

$$F_{x_0}(\mathbf{x}) = \frac{1}{|\mathbf{x} - \mathbf{x}_0|^2} \nabla |\mathbf{x} - \mathbf{x}_0| = \frac{\mathbf{x} - \mathbf{x}_0}{|\mathbf{x} - \mathbf{x}_0|^3}, \quad (32)$$

which gives us the familiar inverse square relationship between gravitational force, and distance between the masses of interest. For purposes of illustration, we shall use the two-dimensional potential function in the following analyses, which are due to Greengard and Rokhlin. Also, for ease of expression, we shall represent the two-dimensional vector $\mathbf{x} = x\hat{\mathbf{a}}_x + y\hat{\mathbf{a}}_y$, as the complex number $z = x + iy$. This way, we may take logs of z directly without the inconvenience of using vector magnitude notation.

3.2.3.1 Bounding the interaction error

From the theory of logarithms, we have the identity

$$\log(z - z_0) = \log(z) + \log\left(1 - \frac{z_0}{z}\right), \quad (33)$$

and the series expansion

$$\log(1 - w) = -\sum_{k=1}^{\infty} \frac{w^k}{k}, \quad |w| < 1. \quad (34)$$

We substitute (33) and (34) into (31) to obtain

$$\phi_{z_0}(z) = \log(z) - \sum_{k=1}^{\infty} \frac{1}{k} \left(\frac{z_0}{z}\right)^k. \quad (35)$$

Now, if we suppose that there are particles of mass $\{m_i, 1 \leq i \leq n\}$ located at points $\{z_i, 1 \leq i \leq n\}$, with $|z_i| < r$, then for any $z \geq r$, we may perform the following derivation for the global potential field.

$$\begin{aligned} \phi(z) &= \sum_{i=1}^n \phi_{z_i}(z) = \sum_{i=1}^n m_i \left(\log(z) - \sum_{k=1}^{\infty} \frac{1}{k} \left(\frac{z_i}{z}\right)^k \right) \\ &= \log(z) \sum_{i=1}^n m_i - \sum_{i=1}^n \sum_{k=1}^{\infty} \frac{m_i}{k} \left(\frac{z_i}{z}\right)^k \end{aligned}$$

$$\begin{aligned}
&= \log(z) \sum_{i=1}^n m_i + \sum_{k=1}^{\infty} \left(\frac{1}{z^k} \sum_{i=1}^n \frac{-m_i z_i^k}{k} \right) \\
&= M \log(z) + \sum_{k=1}^{\infty} \frac{a_k}{z^k}
\end{aligned} \tag{36}$$

where:

$$M = \sum_{i=1}^n m_i, \text{ and} \tag{37}$$

$$a_k = \sum_{i=1}^n \frac{-m_i z_i^k}{k}. \tag{38}$$

Equation (36) is called the *multipole expansion* of the potential field due to n particles. M is simply the total mass in the system, and the a_k are the *multipole expansion coefficients*. Now, we may approximate the potential by truncating the infinite series to p terms where $p \geq 1$:

$$\phi(z) \approx M \log z + \sum_{k=1}^p \frac{a_k}{z^k}. \tag{39}$$

In order to derive the error bound, we begin by rearranging (36) in the following way:

$$\left| \phi(z) - M \log z - \sum_{k=1}^p \frac{a_k}{z^k} \right| = \left| \sum_{k=p+1}^{\infty} \frac{a_k}{z^k} \right|. \tag{40}$$

From (38) and (40), we may obtain the following inequalities bounding the error:

$$\left| \sum_{k=p+1}^{\infty} \frac{a_k}{z^k} \right| \leq M \sum_{k=p+1}^{\infty} \frac{r^k}{k |z|^k} \leq M \sum_{k=p+1}^{\infty} \left| \frac{r}{z} \right|^k = \frac{M}{1 - \left| \frac{r}{z} \right|} \left| \frac{r}{z} \right|^{p+1} = \left(\frac{M}{c-1} \right) \left(\frac{1}{c} \right)^p \tag{41}$$

where $c = \left| \frac{z}{r} \right|$.

Thus, for points sufficiently far away from the particles at z_i , the p -term multipole expansion has a simple error bound dependent upon the geometric relationship between the set of particles and the point at which the potential is evaluated, and the number of terms in the multipole expansion. Note that there are now two parameters that affect the error bound: c and p . c is the ratio of the separation between the particle cluster and the observing point to the cluster size. p controls the amount of work spent evaluating the multipole expansion.

Greengard fixes c at 2, and selects p to obtain the desired accuracy. Note by this method, it is relatively easy to prove that the time complexity of the fast multipole algorithm is $O(N)$. Appel uses a monopole approximation, which is equivalent to fixing p at 1. Thus, Appel's algorithm must use larger separation ratios c between clumps to maintain a constant error. A critical observation about these two approaches may now be made. In (41), we may observe that increasing the num-

ber of multipole expansion coefficients by one will reduce the error in the approximation by a factor of two. To achieve the same reduction in error by only evaluating clumps that are farther apart, one must double the distance between clumps. This has the effect of decreasing the number of interactions that may be approximated to within the error bound, and forcing more of the elementary particle-to-particle interactions to be computed.

It is clear that using more terms in a multipole approximation will increase the computational work required to evaluate the potential field by a constant factor. Increasing the minimum distance between clumps in Appel's algorithm will force the refinement of a constant number of interactions, and thus also increase the computational work by a constant factor. Just which constant factor is smaller will be implementation-dependent.

3.2.3.2 The algorithm

Greengard proposes two versions of his algorithm: one that uses a uniform spatial subdivision, and one that uses an adaptive spatial subdivision. A uniform spatial subdivision is appropriate for systems with a uniform spatial distribution of particles. An adaptive spatial subdivision is appropriate for systems with localized clumps of particles. For purposes of simplicity, we shall consider only the uniform spatial subdivision version here.

Before giving the fast multipole algorithm, several terms must be defined. First, we define the *computational box hierarchy*. Consider a square box which contains all of the particles in our simulation. This box will form the *root* of the box hierarchy (level 0). We now divide the root box into four equal sized boxes to form level 1 of the box hierarchy. By recursively applying this subdivision, we form a hierarchy of boxes that become smaller as we proceed down the hierarchy. Subdivision continues until the smallest boxes (the ones at the leaf level) each contain fewer than a pre-specified number of particles.

In order to satisfy the error bound in (41) sufficient separation must exist between computational boxes if the multipole expansion is to converge properly. With $c = 2$, a sufficient condition for satisfying the error bound is that interactions only be computed between computational boxes at the same level, and that the two interacting boxes not be neighbors. Figure 6 illustrates a four-level hierarchy. A box at the leaf level is labeled as box j . All boxes in j 's *interaction list* are shaded. Note that the eight boxes immediately touching box j are not part of the interaction list because they are too close for the multipole expansion to be valid. Potential interactions between particles in j and particles in these eight boxes must be calculated directly. Interactions between j and boxes outside the interaction list are not considered because they can be handled at a higher (coarser) level in the hierarchy.

Finally, there is a subtle difference between a multipole expansion and a *local expansion*. Note that the multipole expansion in (36) is only valid outside a certain radius, r , about the center of the expansion. If we replace z in (36) with $z - z_0$, where z_0 is the center of the multipole expansion, we may expand in a Taylor series to obtain $\phi(z)$, the multipole expansion about the *origin*, rather than z_0 . Thus, we are able to translate the center of a multipole expansion. This translation comes at a price, however. The original multipole expansion about z_0 is valid for all points

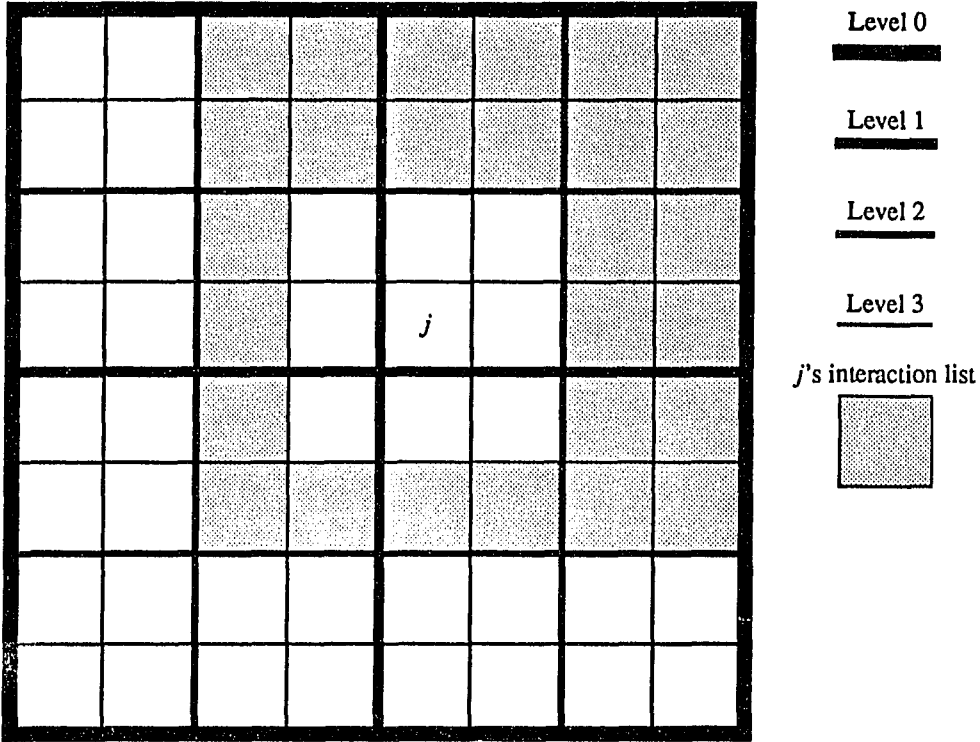


Figure 6: Interaction list of a computational box

lying outside a circle of radius r , centered about z_0 . The shifted expansion is valid for all points lying outside a circle of radius $|z_0| + r$, centered about the origin. Thus, in translating the center of a multipole expansion, we greatly expand the region in which it will not converge. For a detailed description of why this is so, the reader is referred to [Greengard 88 pp. 9-10]. In order to make the fast multipole algorithm work, a method is needed to move the center of a multipole expansion without incurring this convergence penalty. The solution is called a *local expansion*. If we expand (36) in a MacLaurin series, we obtain

$$\phi(z) = \sum_{l=0}^{\infty} b_l z^l, \quad (42)$$

where:

$$b_0 = a_0 \log(-z_0) + \sum_{k=1}^{\infty} \frac{a_k}{z_0^k} (-1)^k,$$

$$b_l = -\frac{a_0}{l z_0^l} + \frac{1}{z_0^l} \sum_{k=1}^{\infty} \frac{a_k}{z_0^k} \binom{l+k-1}{k-1} (-1)^k, \text{ for } l \geq 1, \text{ and}$$

$\binom{l}{k}$ are the binomial coefficients.

If the multipole expansion upon which (42) is based converges *outside* a circle of radius R centered at z_0 , then the local expansion in (42) converges *inside* a circle of radius R centered at the origin iff $|z_0| > 2R$. An error bound for the conversion of a multipole expansion into a local expansion exists, and is similar in nature to (41). For a detailed derivation of this error bound, the reader is referred to [Greengard 88], pages 12 and 13.

The local expansion gives us a way of adding up the contributions from multiple well-separated multipole expansions about a central point. This is possible because *all* of the local expansions will converge in an area of analyticity near the origin, whereas, *no* shifted multipole expansions would converge near the origin. For purposes of clarity, Greengard's fast multipole algorithm is presented in Algorithm 6 as a series of high-level steps rather than in algorithmic notation.

3.2.3.3 Analysis of time complexity

As a basis for analyzing the time complexity of the fast multipole algorithm, we will first analyze how the potential interaction may be computed between two clumps of particles. Figure 7

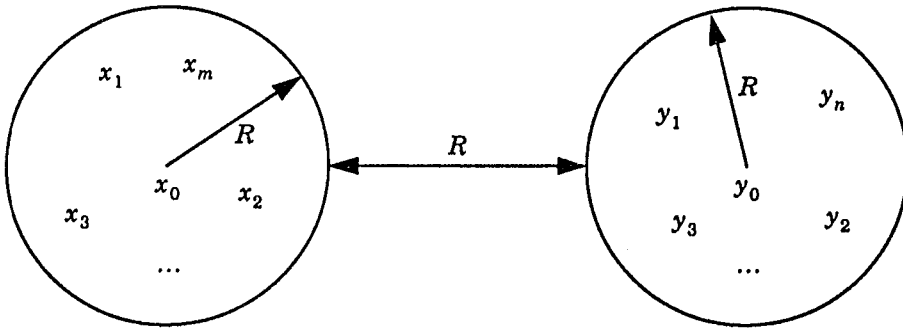


Figure 7: Interaction between two clumps of particles

shows two clumps of particles centered at x_0 and y_0 . In each case, all particles in a clump lie inside a circle of radius R of the center of the clump. In order to calculate the effect of all particles y_j due to all particles x_i , we could simply calculate

$$\phi(y_j) = \sum_{i=1}^m \phi_{x_i}(y_j) \quad (43)$$

for each the n particles y_j . This clearly requires $O(mn)$ work. Instead, suppose we form a p -term multipole expansion of the gravitational potential due to m masses at x_i , requiring $O(m)$ work. We may then evaluate the multipole expansion at each of the n points y_j , requiring $O(n)$ work. Thus, by using the multipole expansion, and settling for a bounded error in a potential interaction, the work may be reduced from $O(mn)$ to $O(m)+O(n)$. But how many interactions are there? It is obvious

1. For each box j at the leaf level of the hierarchy, do the following: Form the multipole expansion of the potential field due to all particles in box j about the box center of box j .
2. Work up the hierarchy, from the level above the leaves towards the root, and do the following for each box, j : Form the multipole expansion about the center of box j by shifting the center of j 's child's expansions. Add all these shifted multipole expansions together to form the composite multipole expansion for box j .
3. Work down the hierarchy, from the root towards the leaves, and do the following for each box, j : Form local expansions about the center of box j due to the composite multipole expansions of all boxes in j 's interaction list. Accumulate these local expansions into a composite local expansion for box j . If j is not a leaf box, then shift the composite local expansion to the center of each of j 's children, and propagate it down to them. After this step is complete, the local expansions at the leaf level are available to evaluate the potential due to all particles other than those in box j and its nearest neighbors.
4. For each box j in the leaf level, do the following: Evaluate the composite local expansion at each particle position to obtain the potential due to all distant particles. Store these potentials into each particle's aggregate potential.
5. For each box j in the leaf level, do the following: Evaluate the potential directly due to all other particles in box j , and j 's nearest neighbors. Accumulate these potentials into each particle's aggregate potential.

Algorithm 6: Greengard's fast multipole algorithm

from Figure 6 that each box at each level in the hierarchy has a constant number of interactions with other boxes. Thus, there are $O(N)$ interactions to be computed. Since multipole and local expansions can be formed, translated, and accumulated in constant time, the total amount of work required to execute Algorithm 6 is $O(N)$.

3.3 Hierarchical Radiosity Methods

The hierarchical method was first applied to radiosity by Hanrahan *et. al.* [Hanrahan 91]. In this algorithm, the coupling between clumps of patches (but not initial polygons) is approximated to within a constant error estimate. There is the added complication that the coupling between single patches is analytically unwieldy, and must therefore be approximated. Furthermore, no

reasonable hard bound yet exists for the error in the coupling approximations, so it, too, must be approximated.

In stellar dynamics codes, physicists have the luxury of being able to accurately model particles as point masses. In the radiosity milieu, patches cannot be modeled as point areas due to their extremely close relative proximity. In summary, the differences between the N -body problem and the hierarchical radiosity problem are as follows:

1. The solution to an N -body problem is the final position and velocities of the particles, or how the particles move. Particle forces and positions are alternately updated until the desired span of time has been simulated. The solution to a radiosity problem is an approximation to the continuous brightness across the surfaces in the scene. Patches never move; instead, they are refined into smaller patches which will better approximate the brightness gradient across a surface. Particles in an N -body problem are indivisible units.
2. Patches cannot be modeled as point areas in the same way that particles can be modeled as point masses. In the realm of N -body problems, the relative separation between particles is very large. It therefore suffices to represent a particle as a point mass. Given a point mass representation, it is trivial to compute the exact gravitational interaction between two particles. In a radiosity environment, closed systems or rooms are always modeled. Thus, there will always be patches which are adjacent to one another. The ratio of their separation to their size will not be large, and thus, a coupling estimate based on point areas will be grossly in error.

3.3.1 Patch couplings and link splitting

Exact couplings can be determined for simple arrangements of patches. Usually, these “simple” arrangements are in terms of axis-aligned rectangles or disks [Sparrow 85, Siegel 81]. Methods exist for approximating the coupling between surface patches which do not fit one of these nice arrangements [Goral 84, Cohen 85, Hanrahan 90, Smits 91]. These methods, however, are plagued by a number of drawbacks. The hemi-cube approximation proposed by Cohen, *et. al*, [Cohen 85] requires a very large amount of work to form its coupling estimate relative to other methods. The other methods are much quicker, but are prone to gross error if the patches are close to one another, or if the support plane of one patch splits the other patch. Also, no tight bound on the error in any of these coupling approximations exists.

At this point, it becomes clear that there are many different sources of error in the hierarchical radiosity problem. There is *coupling estimate error* caused by the inexact nature of the equations used to approximate the coupling between patches. There is *solution error* caused by the inexact (iterative) numerical solution of the linear system of light transport equations. There is *spatial discretization error* brought about by approximating the continuous radiosity solution with a set of constant-intensity patches (These topics will be covered in more detail in the section titled “Alternation of error types” on page 46). In the N -body problem, there are only two sources of error: *interaction error* between clumps of particles, and *temporal discretization error* caused by the discrete time stepping nature of the algorithm.

The method of hierarchical radiosity has a general form similar to that shown in Algorithm 7. The major differences between the two previously existing hierarchical radiosity methods lie mainly in how they define the error in a link, and the error in the solution. In neither case are link and solution error in commensurate units

```

Read in the scene description
Create initial link or links
Initialize brightness of each patch
While the error in the solution is too large
    Refine some links, doing those with the largest error first
    Solve for new patch brightnesses
End while
Write the final patch geometries and brightnesses

```

Algorithm 7: Hierarchical radiosity

A point in Algorithm 7 needs to be further elucidated. Just what “refining a link” means has not been defined. We do so now. In Algorithm 5 on page 29, we see that if an interaction does not meet the error criterion, then the procedure *TwoNode* recurs, and examines two interactions with subclumps. This replacement of one interaction with two interactions at a finer level of resolution is what is meant by *link refinement*. But what if a link is already at the leaf level in the hierarchy? In the N -body problem, such an interaction cannot be further refined because a particle is indivisible. In the radiosity setting, we may *subdivide* the patch into two daughter patches. Patches may always be subdivided if necessary, and new nodes added to the bottom of the hierarchy.

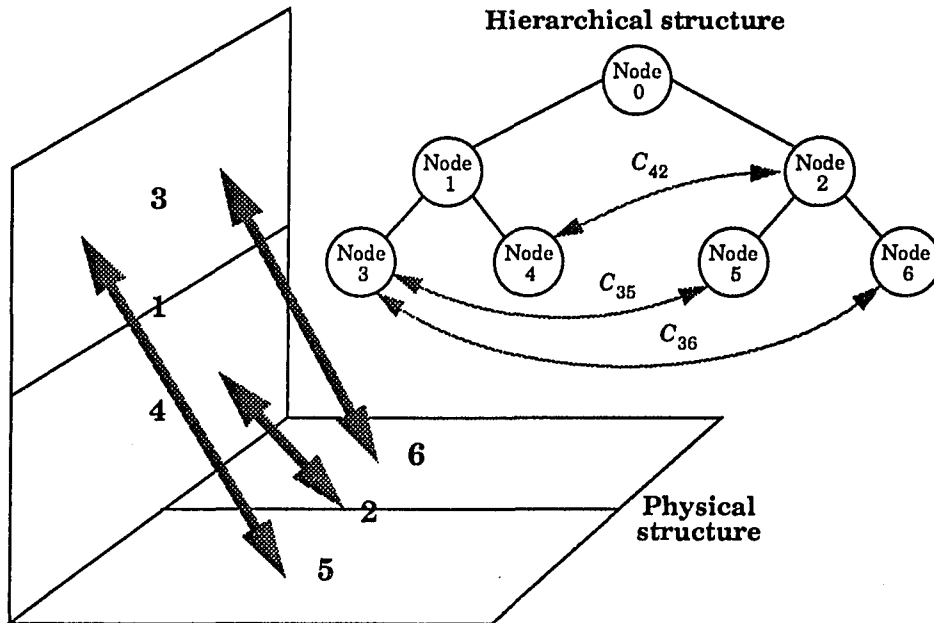


Figure 8: Physical and hierarchical interpretation

Figure 8 is a depiction of the subdivision and coupling of two patches together with its hierarchical representation. Note that patch 1 has been split down the middle to form two subpatches, 3 and 4. Patch 2 has been similarly split into patches 5 and 6. Figure 9 shows the link refinement steps which led to the arrangement shown in Figure 8. The refinement process begins with a single link from the hierarchy root node to itself. Since this link is a self-link, it is refined into three links: C_{11} , C_{12} , and C_{22} . Since both patches 1 and 2 are flat and can have no self-coupling, C_{11} and C_{22} are 0 and we see only the C_{12} coupling in step 2. The C_{12} coupling is then refined into C_{32} and C_{42} , and finally C_{32} into C_{35} and C_{36} .

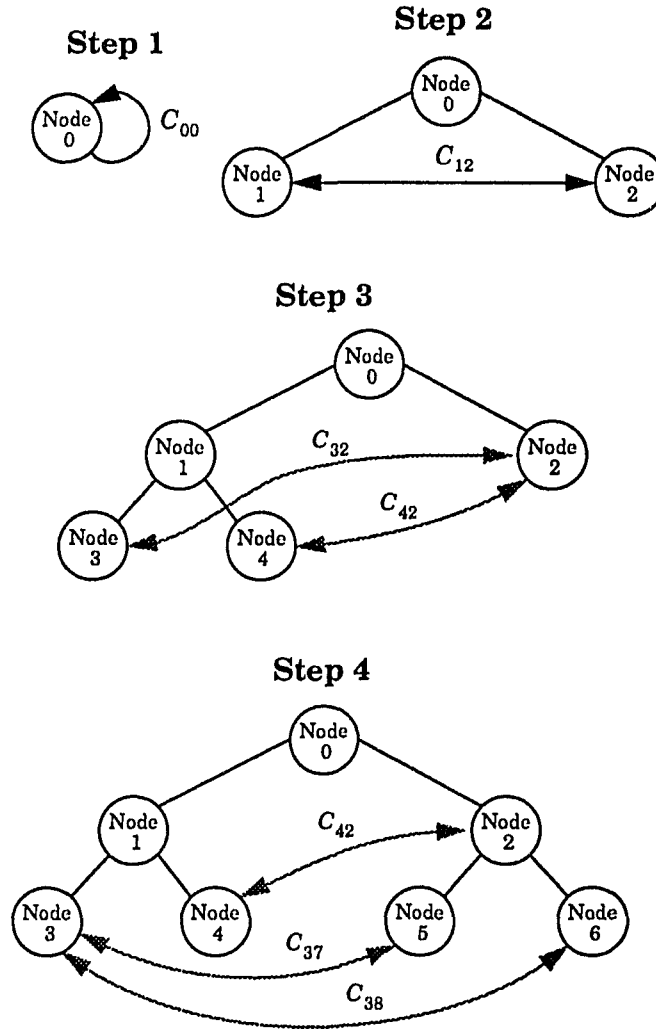


Figure 9: Link refinement steps preceding Figure 8

For the following derivation, we shall use form factors rather than coupling factors. Recall that F_{pq} is the fraction of power (or power per unit area) leaving patch q and arriving at patch p . Thus, if x_q denotes power per unit area (this quantity is called *irradiance*) being emitted by patch q , then $F_{pq}x_q$ is the amount of power per unit area emitted by q that impinges on p . From a physical standpoint, we may calculate the total irradiance of any patch p by summing $F_{pq}x_q$ for all F_{pq} at all nodes above, on, and below node p in the hierarchy.

We will now derive rules for splitting a link C_{pq} on the left. Consider patch p where p has daughters l and r . The expression for the total irradiance arriving at p is

$$b_p = F_{pq}x_q + \alpha_l + \alpha_r + \beta, \quad (44)$$

where C_{pq} is the link to be split, α_l is the contribution from all links at or below patch l to the irradiance of patch l ; α_r is the contribution from all links at or below patch r to the irradiance of patch r ; and β is the contribution from all links at or above patch p , except for link C_{pq} , to the irradiance of patch p . The expressions for the total irradiance arriving at l and r are:

$$\begin{aligned} b_l &= F_{lq}x_q + \alpha_l + \beta \\ b_r &= F_{rq}x_q + \alpha_r + \beta. \end{aligned} \quad (45)$$

Now, suppose we replace link C_{pq} with the two links C_{lq} and C_{rq} . Let us now rewrite (44) and (45) in light of this link splitting. The expression for the irradiance at p now becomes

$$b_p = F_{lq}x_q + F_{rq}x_q + \alpha_l + \alpha_r + \beta. \quad (46)$$

The expressions for the irradiances of patches l and r become

$$\begin{aligned} b_l &= F_{lq}x_q + \alpha_l + \beta \\ b_r &= F_{rq}x_q + \alpha_r + \beta. \end{aligned} \quad (47)$$

Thus, splitting a link on the left affects only the expressions for the irradiance of patches p , l , and r . Such a splitting will presumably yield a more accurate irradiance for patches l and r .

Now, suppose we wish to split link C_{pq} on the right. Again, we will consider patch p , but this time patch q will have daughters l and r . The expression for the total irradiance arriving at p is:

$$b_p = F_{pq}x_q + \alpha + \beta, \quad (48)$$

where C_{pq} is the link to be split, α is the contribution from all links below patch p to the irradiance of patch p ; and β is the contribution from all links at or above patch p , except for link C_{pq} , to the irradiance of patch p . Now, suppose we replace link C_{pq} with the two links C_{pl} and C_{pr} . The expression for the irradiance of patch p now becomes

$$b_p = F_{pl}x_l + F_{pr}x_r + \alpha + \beta. \quad (49)$$

Thus, splitting a link on the right affects only the expression for the irradiance of patch p .

An important addition to the above splitting will be discussed in the section titled “Unidirectional vs. bidirectional links” on page 56. In that section, splitting the link C_{pq} on the left implies a splitting of C_{qp} on the right, and *vice versa*.

3.3.2 Hanrahan’s method

Hanrahan, *et. al*, choose to approximate the form factor from one patch to another using a point-to-disk coupling estimate. In this method, the form factor from a patch p to a patch q is approximated by the following formula:

$$F_{pq} \approx \cos \theta_1 \cos \theta_2 \left(\frac{r_q^2}{R_{pq}^2 + r_q^2} \right), \quad (50)$$

where r_q is the radius of the disk at q ,
 R_{pq} is the distance from the center of p to the center of q ,
 θ_1 is the angle between R_{pq} and the normal to patch p , and
 θ_2 is the angle between R_{pq} and the normal to patch q .

Actually, (50) is derived from the equation for the form factor between a differential area and a disk, not two areas. Hanrahan argues that the differential area to area form factor will be a good estimate of the true form factor as the separation between the area increases. Furthermore, the magnitude of (50) is a good estimate of the error in the form factor itself.

When determining the coupling between two patches, one must be cognizant of the possibility that another patch lies between them. Thus, a *visibility test* must be performed to see if the patches of interest are obscured with respect to one another. Hanrahan’s method fires a fixed number of test rays between the two patches and notes how many are blocked by another patch. The estimated form factor between the patches is then attenuated by the fraction of rays which were obscured. If all rays are obscured, then the candidate link is thrown out completely because no light can be propagated between the two patches. This strategy has serious flaws which are discussed at length in the section titled “Airtight occlusion testing” on page 52.

Hanrahan proposes that a better measure of the error in a link is the amount of *energy* that it propagates, rather than just the form factor between the two patches. The reasoning behind this assertion is that links between dark patches don’t matter because there is little light there to transport in the first place. This alternative criterion for refining links is called “BF refinement” because link error is calculated by multiplying patch *Brightness* by link *Form factor*. From a physical point of view, BF refinement asserts that a correct radiosity solution minimizes the error in the total amount of energy being transported in the scene.

The link refinement process is driven by a decreasing $(BF)_\epsilon$ criterion. A value for $(BF)_\epsilon$ is chosen, and links are refined to this precision. The system is then solved, $(BF)_\epsilon$ lowered, and the process repeated. No specifics are given regarding how $(BF)_\epsilon$ is chosen initially or changed during the course of the algorithm.

Next, there is the subject of solving for patch radiosities. The method in question essentially uses the Jacobi iteration, which was discussed in the section titled “Jacobi iteration” on page 18. The convergence test, however, is not specified in the published work. From this omission, we must assume that little attention was paid to the relationship between the error estimates for links, and the degree of accuracy to which it is appropriate to solve for patch radiosities. This relationship is discussed at length in the section titled “Alternation of error types” on page 46.

Since form factor estimates are not approximated to within a preset error tolerance, Hanrahan’s algorithm is not driven by an *a priori* error criterion like Appel’s and Greengard’s algorithms; rather, it produces a bound on the error as a result of subdivision. This is not necessarily bad, but its appropriateness depends upon the application to which it is applied.

3.3.3 Smits’ method

The hierarchical radiosity method proposed by Smits, Arvo, and Salesin [Smits 91] is an extension to that of Hanrahan, *et. al.* Its major contribution is the introduction of importance into the link refinement process. Until Smits’ importance-driven radiosity algorithm, a radiosity solution was *view independent*; that is, the solution for patch brightnesses and the link refinement process did not depend on from where the scene was viewed. Once the system was solved, it could be viewed from any location in space with equal fidelity. Smits argues that in a scene containing many objects which are not directly viewed, a solution may be reached much more quickly if a viewing point and viewing direction are specified. In other words, patch radiosities are calculated to a high accuracy only for the surfaces which are directly visible to the viewer, or contribute significantly to their illumination.

In order to determine which patches in a scene are “important,” a transport equation is solved which is dual to the light transport itself. Whereas the light sources in a scene emit *light*, the viewpoint emits *importance*. Thus, two simultaneous transport systems are solved: the usual radiosity transport of light from light emitting patches toward the viewer, and the transport of importance from the viewer toward the light sources.

The definition of link error is modified so that only links between *bright and important* patches are refined rather than just links between bright patches. Smits further modified the definition of link error in two ways which are coincident with this research. The first way has to do with taking into account the reflectivity of the patches at either end of a link. Details are discussed in the section titled “Flaw in area/form factor threshold reasoning” on page 55. The second modification has to do with the way coupling error is estimated between two patches. Recall that Hanrahan used the *magnitude* of the form factor between two patches in his estimate of the error in the coupling. Smits proposes taking several estimate samples across the surface of each patch. These samples are averaged together to form the actual coupling estimate, and their *range* is used as an estimate of the error in the coupling estimate. More discussion is given in the section titled “Coupling estimates” on page 57.

Exactly like Hanrahan’s algorithm, Smits’ link refinement process is driven by a decreasing $(BF)_\epsilon$ criterion. A value for $(BF)_\epsilon$ is chosen, and links are refined to this precision. The system

is then solved, $(BF)_\epsilon$ lowered, and the process repeated. No specifics are given regarding how $(BF)_\epsilon$ is chosen initially or changed during the course of the algorithm.

3.4 Problems Amenable to Hierarchical Methods

At this point, we have enough data to make some preliminary observations about what kinds of physical problems are amenable to solution with hierarchical methods. First, we borrow a characterization of potential problems from [Greengard 88]:

$$\Phi = \Phi_{near} + \Phi_{external} + \Phi_{far} \quad (51)$$

where

- Φ_{near} is a very localized potential which decays rapidly with distance,
- $\Phi_{external}$ is an externally imposed potential and is independent of the number or size of particles in the system, and
- Φ_{far} is a far-field potential for which contributions from all particles in the system are significant.

The term Φ_{near} is too localized to benefit from hierarchical methods, and $\Phi_{external}$ is independent of the number and size of particles in the system. It is the far-field potential term, Φ_{far} , which is of interest because of its dependence on all particles in the system. In a 3D potential system, the far-field interaction falls off as the square of the distance between interacting particles. Analogously, the light intensity emitted from a patch falls off as the square of the distance from the patch. This decreasing interaction with distance is precisely the property which allows us to cluster particles and patches together, and estimate interactions between them to a specified level of accuracy.

In order for a hierarchical method to be useful, it must be faster than existing methods. In the case of a highly clustered N -body problem, hierarchical methods have reduced the time complexity of the problem from $O(N^2)$ to $O(N)$ for a given level of accuracy. In the case of the radiosity problem, hierarchical methods have reduced the time complexity of the problem from $O(N^2)$ to $O(N)$ for a given level of accuracy.

More generally, a problem must obey the principle of superposition (or at least have a bounded error for a superposition) for present hierarchical methods to be applicable. Furthermore, the system must be stable so that a small error made solving the system of equations implied by the interactions does not produce a disastrously wrong final answer.

CHAPTER IV

HIERARCHICAL RADIOSITY ENHANCEMENTS

4.1 Introduction

The new hierarchical radiosity method has provided a great leap in performance for radiosity renderings. In its original form, however, several serious issues were left unresolved. We present several major improvements to the hierarchical radiosity algorithm. Among them are: a better accounting of the error in link estimates; a mathematically sound basis for trading off solver error against link error; improved occlusion testing which does not involve ray tracing; and a novel self-consistency check called “rowsum correction” that removes many of the image artifacts associated with hierarchical radiosity.

The method of hierarchical radiosity [Hanrahan 91, Smits 92] has provided a powerful new framework in which to solve the radiosity problem. Form factors are now approximated to only the accuracy demanded by the calculation. Clustering has reduced the complexity of radiosity from $O(n^2)$ to $O(n)$. BF refinement adaptively subdivides polygons only where the error in transported energy becomes too large. Hemi-cubes are replaced by simple coupling estimates and a realization that inaccuracy in the coupling estimate is acceptable so long as it is reducible with patch refinement.

Several drawbacks and deficiencies still exist, however. Error in link estimates and error in system solution are not handled in a consistent fashion. Artifacts caused by the method of estimating coupling factors exist, and have not been acknowledged or mitigated. Currently, no method for clustering initial polygons exists for purposes of creating fewer than n^2 initial links, where n is the number of initial polygons. Also, ray tracing is currently used to determine whether two patches are visible with respect to one another. This method is prone to catastrophic error, is very costly, and does not obey a consistent error criterion. If enough rays are cast to make this method consistent with an error criterion, it completely dominates the execution time.

We address these problems, and put forth techniques and suggestions for dealing with them. Error consistency is supplied by defining solution error, link error, and discretization error in terms of *power*, and alternating between refining and solving to a matching accuracy tolerance. A type of artifact, dubbed the “tartan” artifact, is shown to exist in all hierarchical renderings. A self-consistency correction factor is applied to operations involving the hierarchical matrix-vector multiply, and is shown to deal effectively with the tartan artifact.

A method is proposed for building a hierarchy above the initial polygons so a single unified data structure is seen by the link refinement algorithm, rather than a forest of hierarchies. A method for estimating the coupling between groups of polygons is also presented. This gives the

algorithm the unique ability to start with a *single* link from the hierarchy root node to itself, and refine it into as many links as are needed.

A method of classifying the state of occlusion between two polygons with respect to a single third polygon is presented. This new method classifies the visibility between the two test polygons as either totally visible, partially visible, or totally occluded, and does not have a catastrophic failure mode like the ray tracing method.

4.2 Background and Definitions

Some terms related to the hierarchical radiosity method are defined here. As the subdiscipline is very young, the terminology is not widely used.

The hierarchical radiosity method takes as input, a set of *initial polygons*. *Interactions* or *links* are formed between the initial polygons, representing all possible light transport paths. A link consists of a *coupling factor estimate*, an estimate of the *coupling factor error*, references to the two *patches* between which the link is transporting light, and the *visibility* of the two patches. Visibility is an indication of how much of each patch is visible from the other patch. Once the initial links are set up, the links are placed into a priority queue which is keyed to their link errors. We call this priority queue of links the *link heap*. Links are then *refined* by taking the link with the largest error from the link heap, and splitting one of the patches it couples. Usually, the larger patch will be split or *subdivided*. The link is discarded, and two (or more, depending on how many subpatches are created during subdivision) new links are created between the newly created subpatches and the original patch that was not split. New couplings, coupling errors, and visibilities are determined. These new links are placed back into the link heap.

As subdivision proceeds, a *hierarchy* of subpatches is created below the initial polygons. As refinement proceeds, link error is smoothly reduced. Periodically, a *solution* pass is made to update patch radiosities.

4.3 Discussion

In the following sections, we will discuss some weaknesses in the existing hierarchical radiosity method, and propose enhancements that strengthen the method.

4.3.1 Alternation of error types

The diffuse radiosity equation has several types of error that may be present in a computed solution. The integral form of the diffuse radiosity equation is:

$$b(x) = e(x) + \rho(x) \int_{x'} b(x') g(x, x') f(x, x') dA \quad (52)$$

where

- $b(x)$ is the radiosity at point x ,
- $e(x)$ is the emittance at point x ,
- $\rho(x)$ is the reflectivity at point x ,
- $g(x, x')$ is the visibility between x and x' , and
- $f(x, x')$ is the differential form factor between x and x' .

Equation (52) is typically discretized into a form such as equation (53). The rough loci of four sources of error are pointed out below. A fifth source of error, machine representation error, exists but is not localized.

$$b_i = e_i + \rho_i \sum_{j=1}^N b_j F_{ij} \quad (53)$$

1. Numerical error in the computed solution. This is caused by an inexact numerical solution to the system of linear equations. It is measured in units of power per unit area (typically watts/meter²). We measure it in terms of the residual.
2. Error in modeling patch emittance and reflectance. These quantities are generally assumed to be exact for purposes of computer graphics.
3. Discretization error. This is a measure of how well the patches we have selected approximate the underlying continuous solution to (52) in a piecewise constant manner.
4. Patch coupling error. Error here arises from the approximation of the coupling factor between two patches.
5. Machine precision. A computer can represent real numbers to only a finite degree of precision. Fortunately, 32-bit floating-point representation is usually more than adequate for purposes of radiosity calculations.

As we attempt to solve the radiosity equation, there is no reason to waste time minimizing only one or two of these sources of error. If our patch coupling estimates are only good to within 10%, then solving to more than one decimal of accuracy is meaningless.

Discretization error can be thought of as how much error we introduce by approximating a curve by a constant. Thus, in terms of the radiosity problem, discretization error is related to the change in brightness across a patch, which is, in turn, related to the change in coupling factor across a patch. This change in coupling factor across a patch is one of two things our estimate in coupling factor error seeks to quantify.

It is not clear in the current literature to what criterion solving is done. There is no reason to solve the system to an accuracy greater than that in the coupling estimates. In order to trade off these sources of error against one another, they must be measured in consistent units, such as power. The error in the solution estimate is defined as:

$$\|Ar\|_{\infty} = \|(A - PAF)b - Ae\|_{\infty} \quad (54)$$

where r is the residual vector.

Equation (54) gives the residual in units of power. For the error in a link, we define the following:

$$E_{pq} = \max(\rho_p (C_{pq}^+ - C_{pq}^-) b_q, \rho_q (C_{pq}^+ - C_{pq}^-) b_p) \quad (55)$$

where E_{pq} is the error estimate in coupling of p and q ,
 C_{pq}^+ is the largest sampled coupling of p and q , and
 C_{pq}^- is the smallest sampled coupling of p and q .

Equation (55) gives the estimated error in a link in units of power.

A consistent set of error measures for our calculated solution and couplings is now available. One can now solve only to the accuracy of the refinement, and refine only to the accuracy of the solution. Algorithm 8 illustrates this alternation scheme.

```

Δsolution = ∞
Δrefine = ∞
While numlinks < desired_links
  If (Δsolution < Δrefine)
    { Solve to below Δsolution. }
    { Return new Δrefine.      }
    Δrefine = Refine(Δsolution)
  Else
    { Solve to below Δrefine.  }
    { Return new Δsolution    }
    Δsolution = Solve(Δrefine)
  End if
End while

```

Algorithm 8: Alternation of error types

Experimentation has been done with Conjugate Gradient, Gauss-Seidel, and Jacobi-based solvers. For purposes of this chapter, Jacobi iteration is used as a simple batch-oriented method. This algorithm wastes time neither oversolving nor overrefining. The error in both the solution and the refinement is lowered until the desired terminating condition is reached.

4.3.2 Rowsum correction

One may regard the set of refined links as representing a matrix of coupling factors. The rows of this matrix must, by the definition of coupling factors, add up to the area of the patch which the row represents. Since the coupling factors have only been estimated, the row sum will deviate from the actual patch area by some amount. This error tends to alternate spatially, giving rise to what is dubbed the “tartan artifact.” Figure 10 shows a “Cornell box” rendered with 10,000 links. Notice the dark bands near the box corners which form a plaid pattern.

The reason for this artifact follows directly from the way links are formed near any internal corner, and the nature of the coupling estimate used. Near a corner, links from the patches on one wall to patches on the other wall tend to make two sets of angles with respect to the normal of each patch. Figure 11 shows an illustration of the coupling estimate versus link angle. Also shown is the exact coupling for the same configuration. Marked along the horizontal axis are the two

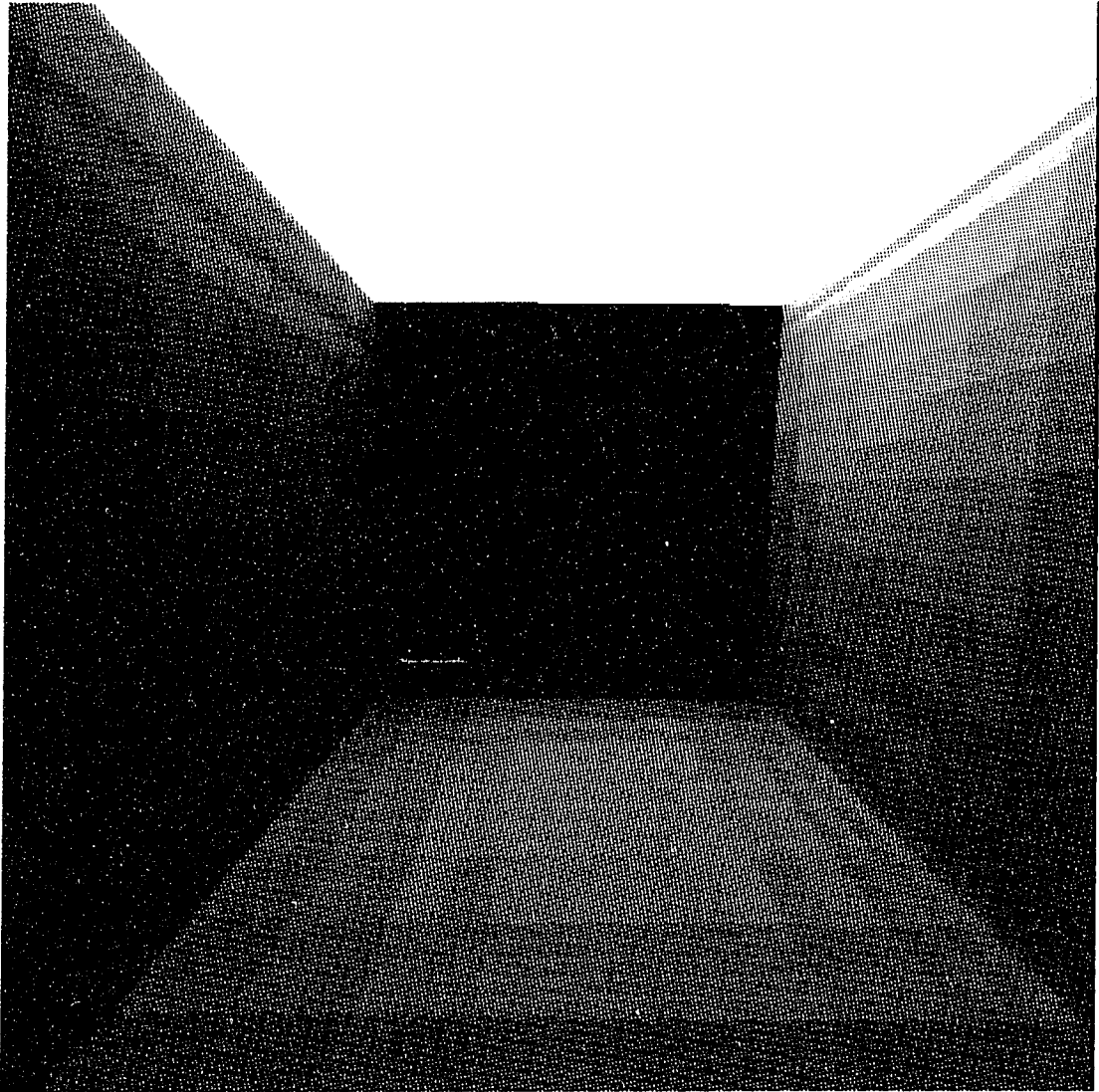


Figure 10: Tartan artifact

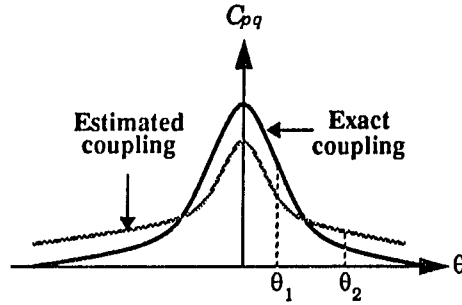


Figure 11: Coupling estimate and actual coupling

angles at which links are formed. In general, the error is different for each angle cluster. The links formed to the rows of patches near the corner in Figure 10 alternate between these two angles. Thus, alternately, too much and too little light will be transported between them, and dark bands will appear.

We know that the rows of the induced coupling factor matrix must add up to the patch areas in a closed scene. Therefore, in operations involving multiplication by the coupling matrix, we may use the following correction: Consider the matrix-vector product $x = \hat{C}v$, where \hat{C} is the coupling matrix perturbed by errors in coupling estimates. If we calculate the rowsums of \hat{C} in a manner similar to matrix-vector multiply, we can construct a correction vector, $r = (\hat{C}i)^{-1}$ where i is a vector of 1's. This correction vector may be used to scale x . Thus we use $x = (\hat{C}v) \text{diag}(r)$ as a better approximation to Cv .

Figure 12 shows the same scene in Figure 10, but with rowsum correction applied during the solution. Note the more uniform color of the walls, and the near-disappearance of the bands characteristic of the tartan artifact.

4.3.3 Clustering of polygons

With complicated geometries containing many initial polygons, the number of initial links created by existing methods in the hierarchy may be prohibitively high. Other hierarchical methods treat initial polygons as the root of their own tree. Thus, one grows a forest of hierarchies as the refinement proceeds.

Here, these independent hierarchies are merged into a single unified hierarchy. The initial forest is merged recursively, as a preprocess, by joining pairs of sub-hierarchies into *composite* nodes. In order to be successful, the composition must capture enough of the salient features of its constituents to produce a reasonable coupling factor estimate. An added convenience is that a single link from the root hierarchy node to itself provides a good "ambient" light approximation.

A simple area-to-area coupling factor estimate is found to work well for calculating composite couplings. The visibility for any link involving a composite node is set to partial. Once this link is refined to a point where both ends are polygons, a proper occlusion test can be performed.

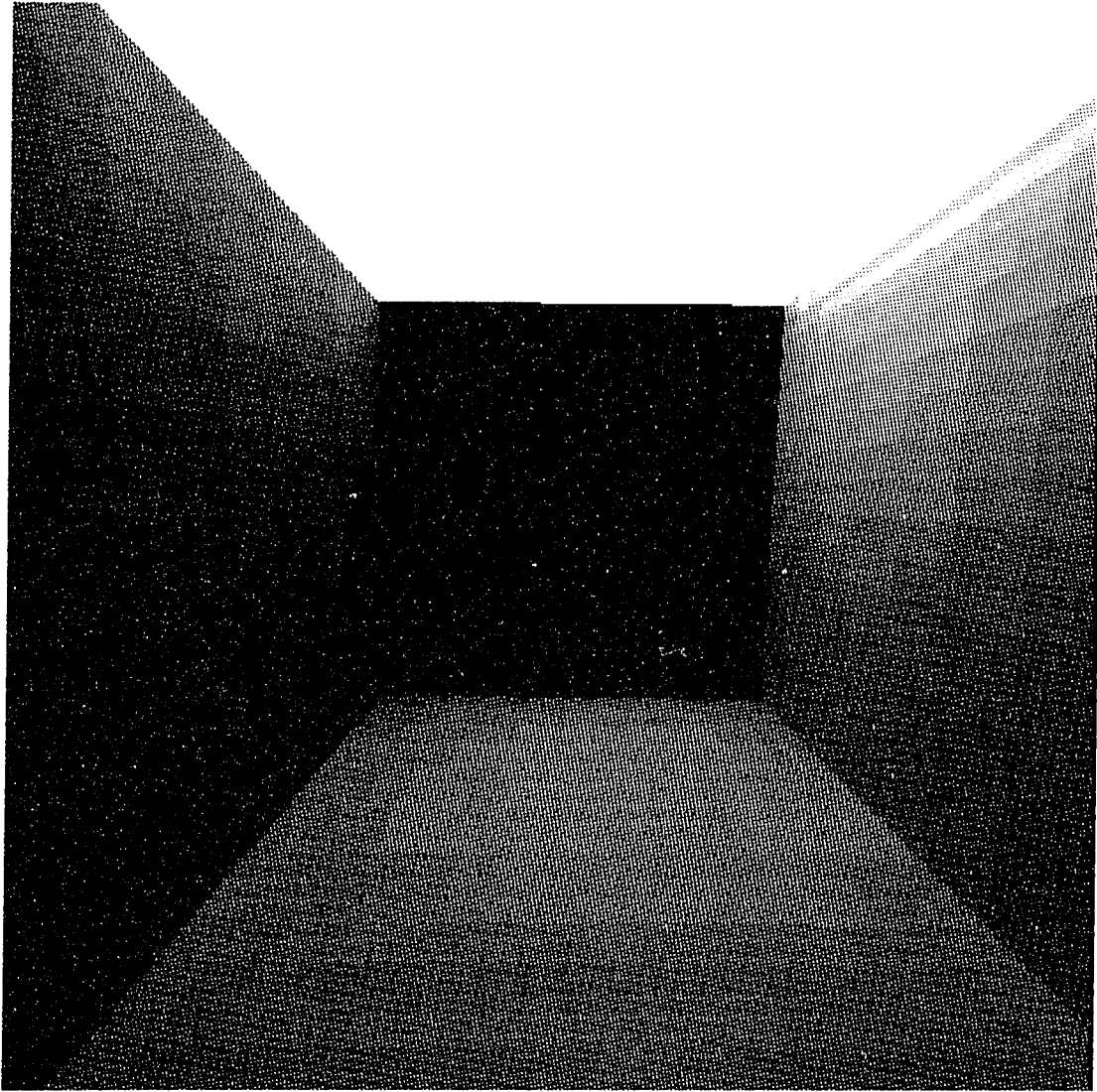


Figure 12: Rowsum corrected scene

An added benefit of clustering is that hierarchical bounding boxes may be built above the initial polygons. This helps with occlusion testing since bounding box checks are much faster than the “airtight” occlusion test to be discussed next.

As mentioned in the introduction, the algorithm may be started with a single link from the root hierarchy node to itself. Normally, a link from a node to itself would be meaningless, because a polygon cannot “see itself.” The root hierarchy node, however, is not a polygon, but a composite of several polygons or other composites. The question arises of how to refine a link from a composite node to itself. Normally, given a link between two unique nodes p and q (they may be composites, patches, or a mixture), the refinement process will split either p or q , and establish links between the daughters of the split patch and the other original patch. When a self-link ($c \leftrightarrow c$) from a composite node c is split, it is replaced with *three* links. Assume c can be split into daughters $c1$ and $c2$. The following links are created: $c1 \leftrightarrow c1$, $c1 \leftrightarrow c2$, $c2 \leftrightarrow c2$. Since either $c1$ or $c2$ may be composites, self-links must be created for them. If $c1$ or $c2$ are polygons, the self-link will be discarded. With this simple scheme, global interchange of light among all patches is accounted for.

4.3.4 Airtight occlusion testing

If couplings are to be approximated within a given error tolerance, and the solution need only be computed to within that same error tolerance, why then should an occlusion test fire a constant number of rays between two patches to approximate visibility? This violates the principle of keeping all sources of error in the calculation at roughly the same level. If this kind of scheme were asked to keep up with the error tolerance in coupling estimates and the solution, a prohibitive number of rays would have to be fired between the patches. A catastrophic error is possible if all sample rays hit or miss a partial occluder. In other words, the maximum error for ray casting will *always* be 100%. *No amount of further refinement will change this.* A better way of determining visibility is needed.

With this in mind, an occlusion test has been created which returns one of three answers: *visible*, *occluded*, or *partial*. *Visible* or *occluded* are returned with certainty. It will return *partial* obscurement if it cannot determine anything else. Visible links never need any further visibility testing performed on any links derived from them. Occluded links are simply thrown out since they propagate no light. Partially obscured links must be tested again at lower levels of subdivision to determine if splitting a patch has caused the visibility to change.

The “airtight” occlusion test takes as input three convex polygonal patches: o , p , and q . Note that each patch has a front and a back determined by the orientation of its normal vector. A patch may only emit or receive light on its front side.

After it is determined that p and q face one another and that they do not split the other with their support planes, the test forms a convex hull between p and q . This hull, together with p and q , forms a closed, convex polyhedron. The occluding polygon, o , is tested against this polyhedron to see if it lies outside, inside, or straddles this polyhedron. In the following description, the poly-

gons p and q will be referred to as the *endcaps*, and the rest of the hull will be referred to as the *waist*.

Note that the waist hull can only be constructed for p and q when they face one another, and the support plane of one polygon does not split the other.

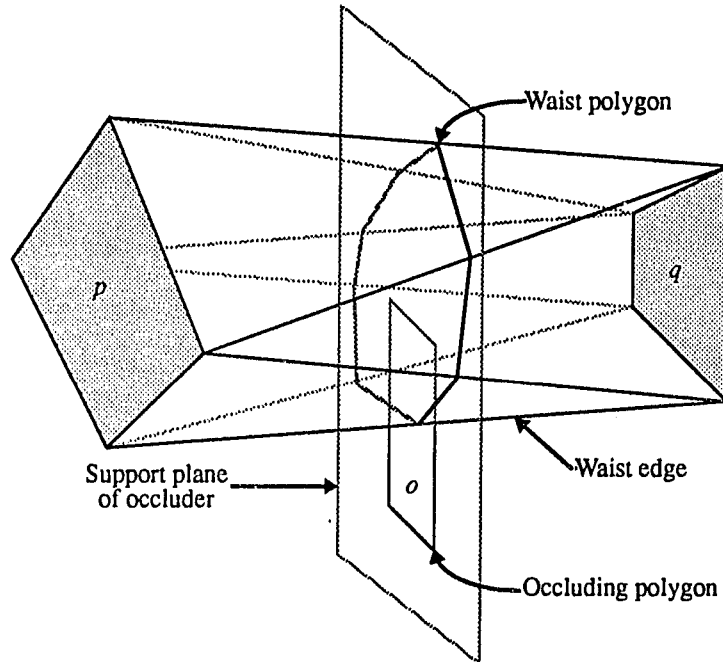


Figure 13: Construction of waist hull

The waist hull (Figure 13) is constructed from the intersection of a number of half-spaces. Each face of the waist hull is planar, and is represented as an oriented plane called a *waist plane*. In the above diagram, each waist plane touches two vertices of one polygon (p or q), and one vertex of the other, thus forming a triangle.

An important computational issue to be addressed is what happens when a point is tested for being on one side or another of a plane, and other similar comparisons against zero. Numerical error can cause a point lying on a plane to appear to lie on either side, or both sides! Instead of simply using the sign of a dot product to test, four ideas are employed: a point may be, with respect to an oriented plane: strictly in front of; on or in front of; strictly behind; and on or behind. This is equivalent to testing the result of the dot product r being $r > \epsilon$, $r > -\epsilon$, $r < -\epsilon$, or $r < \epsilon$.

In the following two algorithms, the $<$ symbol is used to mean “strictly behind,” the \leq sign to mean “on or behind,” and similarly for the $>$ and \geq signs. The algorithm is expressed as Algorithm 9.

1. Visibility Test:
 If $p < q$ or $q < p$
 return occluded
2. Support Plane Splitting Test:
 If support plane of p splits q
 or support plane of q splits p
 return partial
3. Endcap Test:
 If $o < p$ or $o < q$
 return visible
 If $p \geq o$ and $q \geq o$
 return visible
 If $p \leq o$ and $q \leq o$
 return visible
4. Waist Plane Tests:
 Construct waist planes
 If $o \leq 1$ or more waist planes
 return visible
 If the intersections of all
 waist edges with support plane
 of o lie on or inside of o
 return occluded
5. Failing all else
 return partial

Algorithm 9: Airtight occlusion test

The details of constructing the waist planes are contained in Algorithm 10. Algorithm 10 details how to construct the waist planes which rest against an edge of p and a vertex of q . A similar procedure must be done to construct the waist planes that rest against the edges of q and the vertices of p . Care must also be taken to ensure that the waist edges are kept in the proper order such that their intersections with the occluder support plane naturally sweep out the waist polygon.

The current implementation of Algorithm 9 for quadrilaterals takes, in the worst case, about 2200 floating-point operations per call. On average it takes about 320 floating-point operations per call because of early return exits. Note that the occlusion routine will typically be called with the same p and q many times, but with different occluding polygons. Since most of the calculations are specific to p and q , a drastic reduction in work can be achieved if one reuses the waist hull from a previous call with the same p and q .

The airtight occlusion test has one drawback for a small number of links. Since it can only test against one occluding polygon at a time, it will sometimes classify links as partial when, in fact, they are completely occluded. Consider some p and q with two large abutting polygons $o1$ and $o2$ between them such that p and q are completely obscured by the combination of $o1$ and $o2$. The airtight occlusion test will say that p and q are partially visible with respect to either $o1$ or $o2$. Since it has no precise geometric information about the union of $o1$ and $o2$, it cannot detect that p and q

```

For all vertices  $i$  in  $p$ :
   $e = \text{Pvertex}[i+1] - \text{Pvertex}[i]$ 
  For all vertices  $j$  in  $q$ :
     $f = \text{Qvertex}[j] - \text{Pvertex}[i]$ 
     $n = f \times e$ 
    If  $q \geq \text{plane}(f, e)$ 
      Accept plane  $(f, e)$ 
      Break  $j$  loop
    End if
  End for
End for

```

Algorithm 10: Waist plane construction

are actually completely occluded. In effect, p sees q “through the crack” between $o1$ and $o2$. The upside to this problem is that further refinement will eventually attenuate the amount of light which “leaks through the crack” to an arbitrarily small amount—an advantage that the ray tracing approach to occlusion testing does not share.

4.3.5 Binary vs. quadtree subdivision

Hanrahan *et al.* promote the idea of subdividing a quadrilateral into four quadrants when it needs to be split. This has a major drawback: if an eccentric quadrilateral is split using this philosophy, the four pieces will also be eccentric. Eccentric polygons provide very poor coupling estimates. Therefore, with quadtree subdivision, the coupling estimates will improve very slowly.

Binary subdivision, on the other hand, does not share this problem. The subdivision algorithm now has a choice as to the way it splits the quadrilateral. It can choose to split the longest side and the side opposite that side. If a quadrilateral of eccentricity less than $\sqrt{2}$ is split in this way, the resulting daughter patches will be more eccentric than the mother. However, when the daughters are split again, their eccentricities will decrease to less than or equal to their grandmother's.

Applied recursively, this scheme will tend to reduce the eccentricity of the subdivided patches. Since daughter patches are less eccentric than their mothers, they will have better coupling estimates to other patches in the scene.

4.3.6 Flaw in area/form factor threshold reasoning

The original algorithm proposed by Hanrahan, Aupperle, and Salzman used two error criteria to terminate subdivision: A_c and F_c . F_c was the smallest form factor a link was allowed to have. If a link ever fell below this threshold, it was never again considered for refinement. A_c was the smallest area a patch was allowed to have. If a patch became smaller than A_c , then it could not be subdivided further.

Apparently, the A_c and F_c criteria still exist in both formulations of the hierarchical radiosity method [Hanrahan 91, Smits 92]. Both disclaim that with BF refinement and/or importance-driven refinement, the A_c test is seldom necessary. Form factor error is not in itself important.

Only as an element of energy transport does it have any significance. F_e implies error is the goal of refinement; it is not. The real error criterion is reflected power: $\rho_p C_{pq} b_q$. *Only* the reflected power criterion should be used; it should not be augmented with an arbitrary A_e or F_e test.

In fact, it is simply wrong to use an arbitrary threshold of any kind in the refinement process. If the error in a link is so large that a patch needs to be split, then it should be split. If it is not, then one has effectively established a minimum error below which the algorithm can no longer accurately refine. The A_e and F_e thresholds are not simply superfluous; it is incorrect to use them and expect the algorithm to proceed accurately.

4.3.7 Link subdivision

The algorithms of both Hanrahan and Smits refine links to some chosen error criterion, which we will call $(BF)_e$, at each subdivision step. All links whose errors are greater than $(BF)_e$ are refined until the error in all subsequent links falls below $(BF)_e$. In neither case does the author indicate how $(BF)_e$ is chosen initially, or how it is lowered.

We propose keeping the links in a priority queue (link heap) organized by their estimated link error. This way, the link with the largest error is immediately available. Pushing links onto and popping links off of the heap are both $O(N)$ operations. When a system solution step is performed, all link error estimates are changed, and thus the heap must be reheapified. This is an $O(N)$ operation, and does not change the overall time complexity of the algorithm. Keeping the links in a heap structure has the advantage of always attacking the greatest source of error among the links. Thus, the overall link error is lowered as quickly as possible, and the greatest economy is achieved in terms of the number of links used to obtain a certain error.

4.3.8 Unidirectional vs. bidirectional links

Even with the space for coupling factors reduced from $O(N^2)$ in the number of patches in a scene to $O(N)$, the hierarchical radiosity method is still memory-limited on most computers. Several factors have contributed to this:

- Coupling estimates between patches can be generated quickly.
- Since the coupling matrix has only $O(N)$ blocks, solution time is also $O(N)$.
- With alternation of error types, excess time is not wasted solving to too high an accuracy.
- Occlusion testing between patches is relatively inexpensive when compared to ray casting.

In its original form, the hierarchical radiosity method used unidirectional links. That is, a coupling between patches p and q had a link at p pointing to patch q , and *vice versa*. Thus, the storage used for couplings was twice what it should be. This exacerbates the memory-limited nature of the hierarchical algorithm. Far worse in a computational sense is that the storage for these links is scattered throughout physical memory in a computer. Since each node in the hierarchy must have room for a variable number of links to other nodes, several serious performance-limiting issues exist:

- *Static storage.* One may allocate a constant amount of space per node for storing links. If the list is too small, a node may overflow its link table. Even if chosen correctly, having a constant-size list of links at each node will waste a tremendous amount of memory.
- *Dynamic storage.* One may dynamically manage link tables at each node. The overhead of calling a memory allocator every time a link table needs to be expanded will become crippling. Furthermore, memory will become extremely fragmented after the algorithm has run for a while. And again, the possibility of egregious memory waste exists.
- *Priority queueing of links.* In order for link refinement to proceed, the link with the greatest error in propagated energy must be available for refinement quickly. This means either keeping a copy of all links in the hierarchy or searching the hierarchy.

The only practical reason for storing the links owned by a particular patch in the patch data structure itself is to facilitate shooting of light. A more efficient strategy is to remove the links from the hierarchy, and keep them in a dedicated priority queue, or “heap.” This way, one only stores one copy of the link, makes the best use of memory possible, and keeps links in a form where the link with the largest error is available quickly.

4.3.9 Coupling estimates

Hanrahan *et al.* use a single point-to-disk coupling estimate. We have observed that the coupling factor estimate takes a small fraction of the total time in a hierarchical radiosity calculation. Thus, investing more work to more accurately estimate the coupling and its error is warranted.

We use the four corners, plus the average of the four corners, as five sample points on each patch. The coupling estimate between 8 pairs of these sample points is then computed. The coupling estimate between the “centers” is calculated, and counted twice. The minimum and maximum of these 10 coupling factor estimates are tracked to estimate the error in the final coupling factor estimate. Singular couplings produced by patches that touch are replaced with zero, since the singularity disappears in integration.

At this stage, coupling factor samples are allowed to be negative. Negative couplings will happen when one of the sample radii passes behind either patch, such as when the support plane of one patch splits the other patch. Since the minimum of all coupling estimates will be negative in these cases, the difference between the maximum and minimum will be larger than if negative couplings are simply discarded, or treated as zero. This has the beneficial effect that links between patches where a plane splitting occurs are split sooner than links between other patches. To form the overall coupling estimate, all non-negative coupling estimates are averaged. Also, partial couplings are scaled by 0.5 in the absence of information regarding *how* partial the visibility is.

4.3.10 Estimation of error in coupling estimates

The error estimate in a coupling factor estimate serves a dual purpose. First, it seeks to quantify the error which is intrinsic in the coupling estimate between two patches. Second, it seeks to quantify the variation in coupling factor across a patch. This variation in coupling factor is a mea-

sure of discretization error. Thus, this error estimate is used to control both the error in links and the error in discretization of patches.

4.4 Results

The "harpsichord practice room" (Figure 14, Figure 15) was solved on a DECstation 5000/240 with 32 MBytes of physical memory in 50 minutes. 500,000 links were created, and the resulting hierarchy had 23,423 leaf-level patches. 63.5% of all links are completely unoccluded, the remainder are partially occluded. 19,291 links were thrown out due to total obscurement. Only 36.5% of the total time was spent refining links. 62.9% of the time was spent solving for patch radiosities. The remaining 0.6% was spent reading the scene description, and writing the patch positions and radiosities. The algorithm required approximately 19 MBytes of physical memory to run. Below in Table 4 is a report from our program. Note that we have purposely left Figure 14 and Figure 15 unsmoothed so that any solution artifacts that remain can be clearly identified.

Table 4: Program performance report

500,000 links:				
Task	Seconds	Operations	MFLOPS	% of Time
Reader	0.16	19022	0.122	0.0%
Refine	1092.25	5709913505	5.228	36.5%
Solver	1881.17	1379848860	0.734	62.9%
Storer	19.21	4931430	0.257	0.6%
TOTALS	2992.79	7094712817	2.371	100.0%

4.5 Summary

In any physical problem, there are multiple sources of error. By understanding these sources and exploiting the fact that no part of the problem need be solved to an accuracy greater than that of any other part, we may arrive at an acceptable solution with minimal work.

Previous work has treated solution error in a cavalier manner. Shooting was used to solve for patch radiosities without regard for the error present in patch-to-patch coupling estimates. We have set forth a method of objectively alternating between link refinement and radiosity solution that keeps both types of error in balance. A measure of discretization error is incorporated into our link error estimate as well.

A novel self-consistency check called "rowsum correction" is based upon well-known properties of form factors, and is effective in dealing with image artifacts created by systematic inaccuracies in coupling factor estimates.

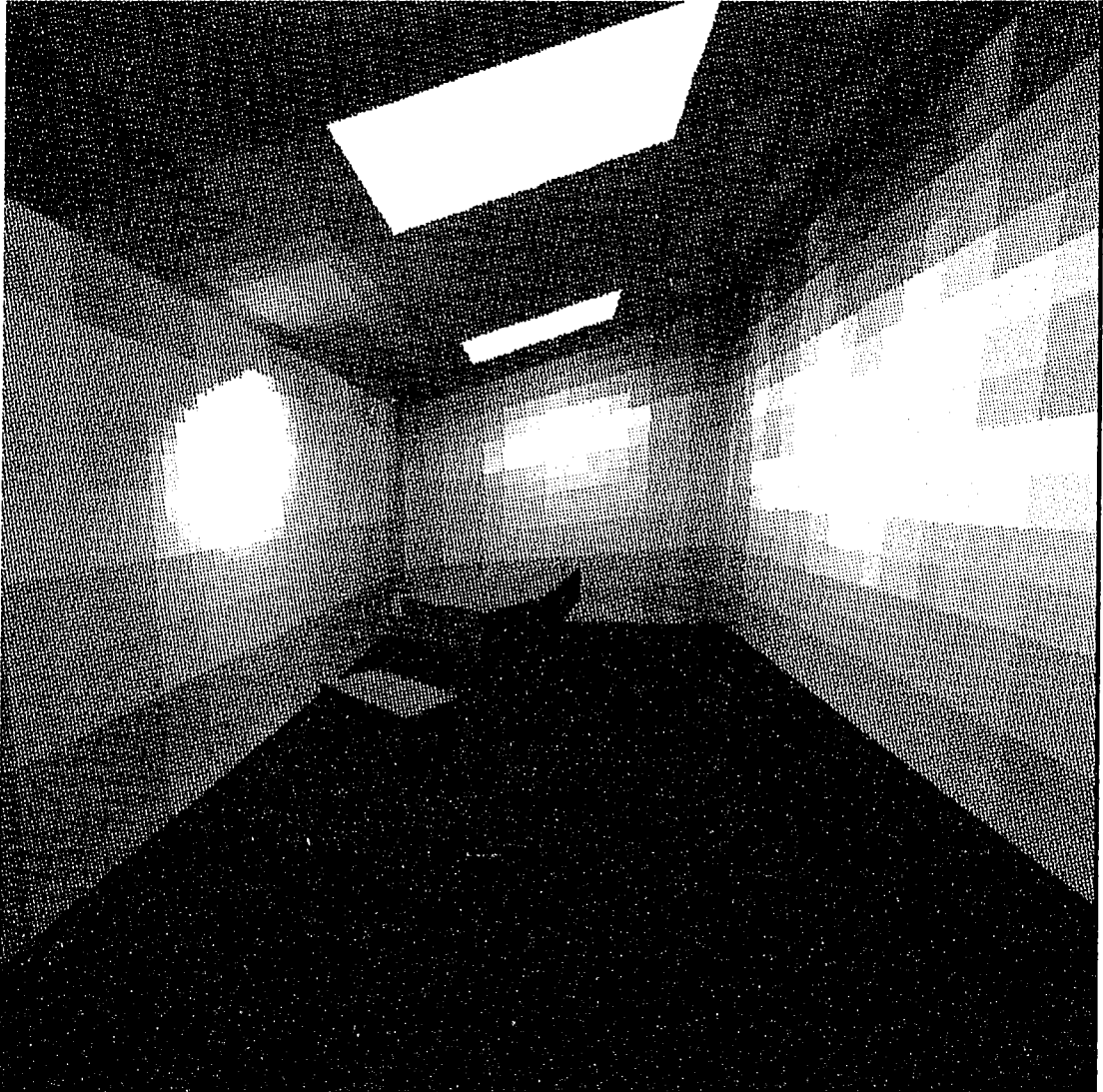


Figure 14: Harpsichord practice room without rowsum correction

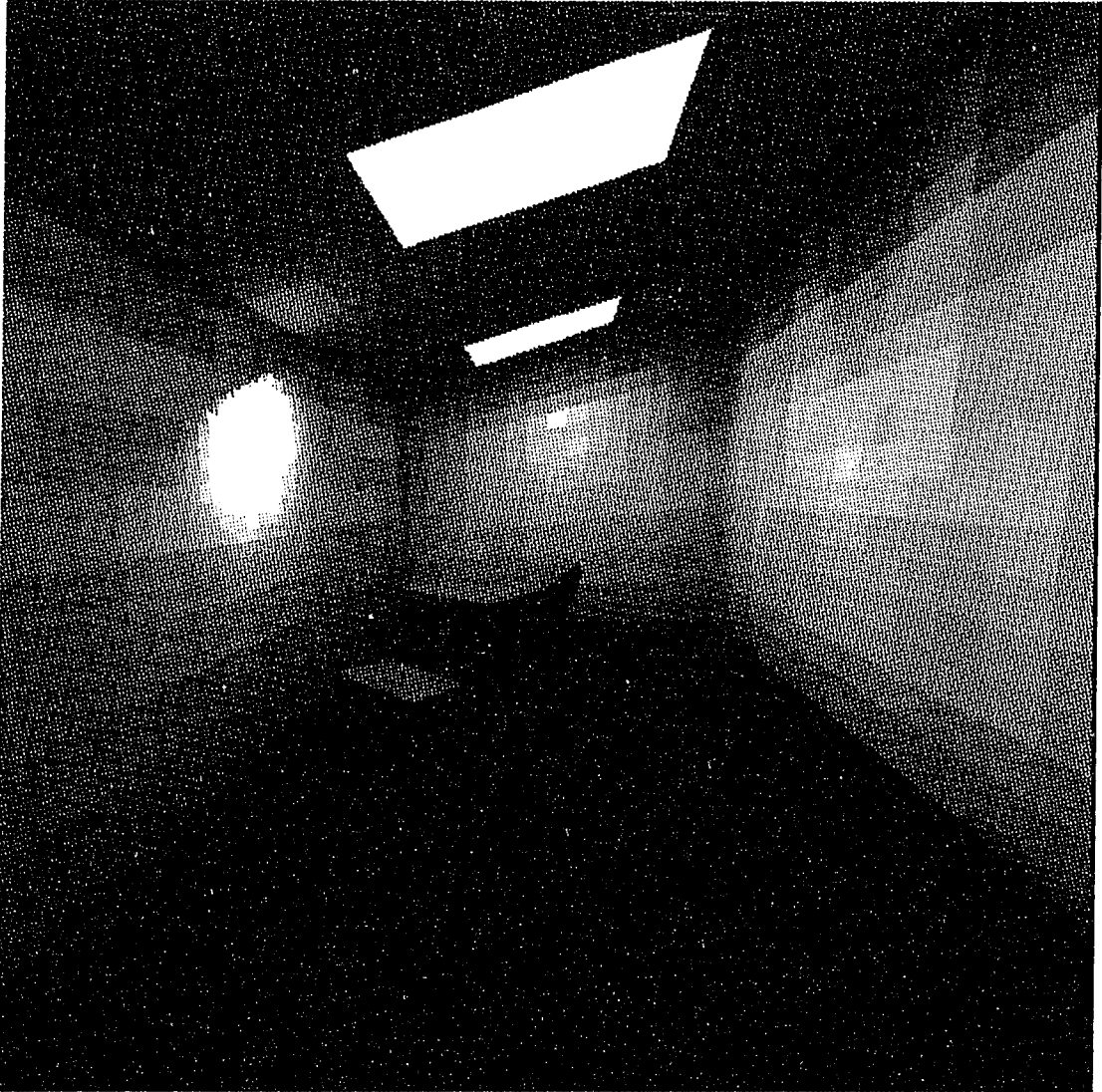


Figure 15: Harpsichord practice room with rowsum correction

CHAPTER V

MAKING THE HIERARCHICAL METHOD PARALLEL

5.1 Elements of a Good Parallel Program

Before launching into the details of how to implement the hierarchical radiosity algorithm on a parallel machine, let us pause to review what the attributes of a successful parallel program are. Some of these attributes are common-sense matters, and some have had their importance emphasized only through extensive experience with multiple parallel architectures.

First, let us consider efficiency. In this context, we define efficiency as a subjective measure of "how well" a particular algorithm utilizes the hardware it is running on relative to another algorithm running on the same hardware. A parallel implementation should not suffer a large penalty just because it is being run in parallel. Many factors contribute toward the final efficiency of an algorithm-hardware combination such as algorithm choice, hardware platform choice, data decomposition and mapping, control structure decomposition and mapping, effective load balancing, and effective use of language features. A most effective illustration of just how much effort has been expended on this topic is in the area of linear equation solving. A vast number of matrix decomposition strategies have been studied to see how well-suited they are to a particular parallel architecture. Matrix decomposition strategies tried include: row-wise wrap mapping, column-wise wrap mapping, row-wise serpentine mapping, column-wise serpentine mapping, horizontal strip wrap mapping, vertical strip wrap mapping, 2D block decomposition, 2D scattered decomposition, etc. Hardware topologies studied include hypercube, torus, mesh, ring, and various bus-based systems. The point is that a problem may be approached from many different angles; one must be extremely careful when laying out the software architecture for an application from the very beginning.

Next, we focus on algorithm choice. On a uniprocessor machine, algorithm choice is not nearly as critical as on a parallel machine. Notwithstanding data dependencies, the manner in which data is accessed and the order in which computations are performed matter little, aside from pipeline and cache effects. On a parallel machine, however, things are different. First, there must be sufficient work to perform at all times to keep all processors busy. If there is insufficient parallelism in the algorithm, efficiency will suffer due to unutilized processors. There is also the issue of data locality. All present large-scale parallel computers have fast memory local to each processor, and a slower method of retrieving data from other processors. In some cases, this data retrieval is performed via explicit message-passing; in others, it is handled by a shared virtual address space. In all cases, remote memory access is significantly slower than local memory access. If an algorithm constantly requires data from other processors to operate, its efficiency will suffer. Every effort must be made by the programmer to insure that as much data as possible is local so as not

to incur communication penalties. Sometimes, this means not choosing the best serial algorithm available, but rather backing up to an algorithm which has a sufficient amount of parallelism to exploit.

With the recent availability of large scale parallel computers, scalability issues must be taken into account. Put simply, scalability is a measure of how well an algorithm is able to effectively utilize an increasing number of processors. An algorithm should not be prejudiced toward a particular size of parallel computer. Where possible and reasonable, it is desirable to make an algorithm independent of exact machine topology, or at least modularize the communication primitives so that they can be easily modified if the algorithm is moved to a new architecture. This leads us to our next topic: portability.

From the beginning, an algorithm should be designed as architecturally-independent as possible. The extremely short product lifespan of current parallel computers makes forward-thinking software design critical if an application is to survive more than a few years without major redesign.

5.2 Statement of Algorithm

Until now, only a general outline of the hierarchical radiosity algorithm has been given. In order to conduct a meaningful discussion on how to make it parallel, a more detailed description of the hierarchical algorithm is in order. Provided below is a detailed pseudocode description of the exact algorithm to be discussed in this chapter.

We will now review the serial implementation details of several key steps in Algorithm 11. First, let us define some terms, and visualize the major data structures of this algorithm. The scene is defined by the user in terms of a set of *polygons* which form a closed environment. Another user-specified quantity is the number of links they wish the algorithm to use to propagate light about the scene. It is the algorithm's responsibility to use these links in the most effective manner possible. Step 3 states that a hierarchy should be built atop the given polygons. Figure 16 shows how the hierarchy of composite nodes is constructed. Note that the composite nodes are numbered, starting from 1, in the order that they are created. The root composite is numbered zero. The input polygons are placed into a queue to start with. Pairs of nodes are removed from the head of the queue, and a composite node constructed as their parent. The new composite node is then pushed onto the tail of the queue, and the process repeated until there are no nodes left. The last composite formed is called the root node, here numbered zero.

A word is in order about what kind of data is and is *not* stored in the nodes of the hierarchy. Every node in the hierarchy contains the following data: the *area* of what the node represents, be that a polygon or a composite; a *center* which will be used as the 3D coordinates of a patch; locations for storing *hierarchical vector elements* (to be discussed below); and *left*, *right*, and *parent* pointers to implement the tree structure of the hierarchy. Each polygon node contains the following additional data: its four *vertices* (which must be coplanar), a *normal* vector to the polygon, and a *bounding box* to be used to speed up occlusion tests involving the node. When polygons are sub-

1. Read the list of polygons comprising the scene and their properties
2. Read in the number of links to form, *reqlinks*
3. Build hierarchy above these polygons forming composites recursively
4. Initialize all composite nodes
5. Initialize the link heap
6. Initialize brightness solution vector to patch emissivities
7. Form a single link from the root node to itself
8. Push this link onto the link heap
9. Set *numlink* = 0
10. Set *solv_error* = 0
11. Set *link_error* = error in the root self-link
12. While *numlink* < *reqlinks*
13. While *link_error* > *solv_error*
14. Remove link from top of heap
15. Subdivide the end of the link with the greatest error
16. Form two new links from the unsplit end of the original link
 and the two new patches
17. Evaluate the new links' *visibility*
18. Approximate couplings for the new links
19. Compute an error estimate for each link
20. Push the links onto the heap if they are not totally occluded
21. Increment *numlink* accordingly
22. Set *link_error* = error in link at the top of the link heap
23. End while
24. Set *solve_flag* = 0
25. While *solv_error* > *link_error*
26. Conduct one step of Jacobi or Conjugate Gradient iteration
27. Compute a new *solv_error*
28. Set *solve_flag* = 1
29. End while
30. If *solve_flag* = 1
31. Update all link error estimates due to new solution vector
32. Reheapify the link heap
33. End if
34. End while
35. Write out the leaf-level patches and their brightnesses

Algorithm 11: Improved hierarchical radiosity

divided during the solution process, daughters are added below them which are identical in structure and content to the original polygon nodes.

In previous radiosity renderers, the solution to the radiosity equation is a vector of patch intensities (see equation (8) on page 15). In the hierarchical radiosity algorithm, the solution can still be regarded as a vector, but the definition of a vector must be modified slightly. For purposes of hierarchical radiosity, we shall call the left-to-right ordering of the patch brightnesses in the leaf level of the hierarchy a *hierarchical vector*. All quantities in (8) which were conventional vectors become hierarchical vectors in the context of hierarchical radiosity. The form factor matrix, F_{ij} becomes a *hierarchical matrix*, whose structure and elements are induced by the links created in the refinement process. Shown in Figure 17 is a hypothetical patch hierarchy with a set of six links connecting the nodes of the hierarchy. Note that nodes in the hierarchy are not identified as

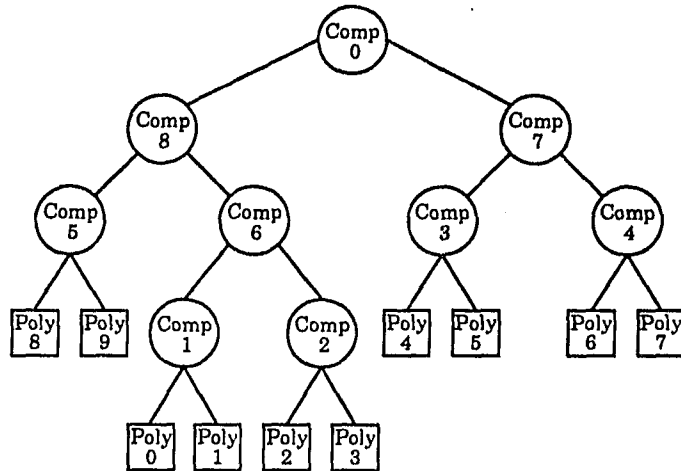


Figure 16: Construction of composite hierarchy

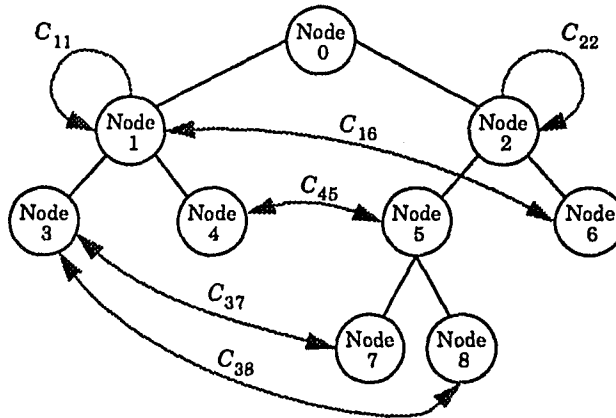


Figure 17: Couplings in a patch hierarchy

polygons or composites. It does not matter whether the end of a link points to a polygon or a composite; the nature of the link remains the same. Each link is labeled with its coupling factor value. There are five leaf nodes in the sample hierarchy. Therefore, we should expect to solve a system of five equations in five unknowns. The linear system coefficient matrix that is induced by the links in Figure 17 is shown in Figure 18. This is matrix C in equation (15) on page 17. Note that $C_{pq} = C_{qp}$ due to the definition of coupling factors (equation (14) on page 16). Now that we have the concepts of hierarchical vectors and hierarchical matrices, it makes sense to talk about operations on them. Element-wise operations on hierarchical vectors are trivial. One simply applies the operation (addition, subtraction, scaling, negation, etc.) to the hierarchical vector element or elements in each leaf node in the hierarchy. Hierarchical vector dot product, and norms are handled

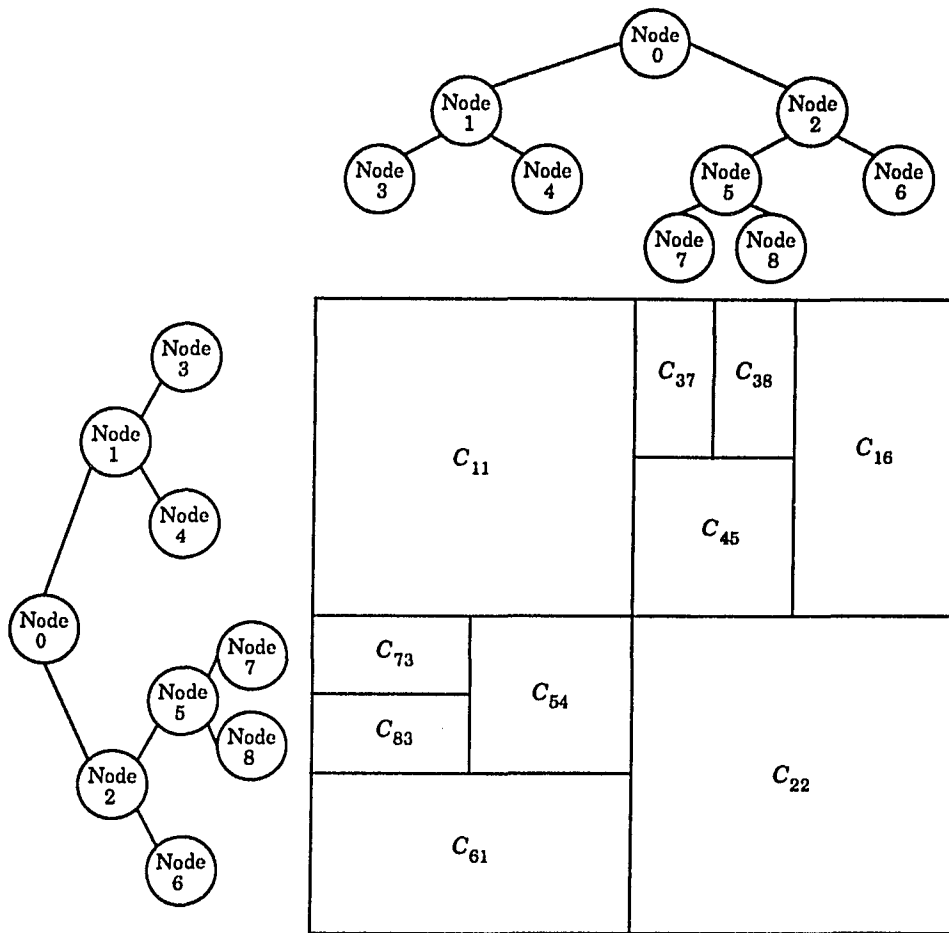


Figure 18: Structure of coupling matrix

similarly. Hierarchical matrix-vector multiply is somewhat more complex, and will be discussed later in this chapter.

Step 4 calls for the initialization of all composite nodes in the hierarchy. As discussed in the section titled "Clustering of polygons" on page 50, a composite node serves to summarize salient geometrical features of all patches in its subtree. The area of a composite is the sum of the areas of its daughters. Each hierarchical vector element is the area-weighted average of its daughter's corresponding elements.

Step 5 calls for the initialization of the link heap. Later on in the algorithm, it will be necessary to have fast access to the link with the largest error. A heap provides a log-cost method for popping the link with the largest error, and adding new links.

Steps 7 and 8 form a single link from the root hierarchy node to itself, and push it onto the link heap. The intuitive significance of this step is important. The root hierarchy node represents all patches in the scene. A link between two nodes in the hierarchy says that those two nodes are exchanging energy. A link from a node to itself represents an exchange of energy among its constituents. Thus, a link from the root (composite) node to itself represents a scalar summary of all light interaction in the scene. Note that a link from a node to itself only makes sense for a composite node. A patch cannot emit light that will directly fall on itself because it is flat. Another important point is that the system starts out *fully connected*; every patch sees every other patch through this single link. Whenever a link subdivision is performed, this full connectivity is maintained.

In the next several steps, the variables `link_error` and `solv_error` are initialized. The variable `link_error` is the largest error estimate of any link in the system. This error is lowered by subdividing links. The variable `solv_error` is the residual from the approximate iterative solution to the radiosity equation. It is lowered by running the iterative solver for additional iterations. The key idea with these two variables is that there is no reason to solve the system to an accuracy greater than that of the link subdivision, and no need to subdivide links any more accurately than the patch radiosity solutions. Thus, the algorithm alternately subdivides links, and solves in a leap-frog fashion until the desired number of links have been constructed.

Step 15 performs two distinct actions; it decides which end of the link to split, and then actually subdivides that node. The decision regarding which end to split is relatively simple: If the nodes at both ends of the link are composites, the larger composite is split. If only one of the nodes that the link connects is a composite, then the composite is split. Otherwise, both ends of the link are polygons, and the larger one is split. Splitting a composite node is a null operation since composite nodes already have daughters. Splitting a polygon is a null operation if it has already been split by a previous link refinement. Note that it is possible to split the same hierarchy node many times during the course of link refinement. If the polygon has not been split, it is split into two smaller polygons along a line connecting the midpoints of its longest side and the side opposite.

Steps 16 through 19 form and initialize two new links to take the place of the one link popped off the top of the heap in step 14. A link contains the following quantities: a *coupling estimate* between two nodes in the hierarchy, *references* to each of these nodes, a *visibility flag*, and an *error estimate*. Steps 20 through 22 push these new links onto the link heap, update the link count, and retrieve the new maximum coupling error estimate.

Steps 24 through 28 conduct just enough iterative solution steps to bring the solution residual under the maximum estimated link error. An explanation is in order about the choice of hierarchical vector norms for use in these steps. One-, two-, and infinity-norms all work for hierarchical vectors, but they have different physical interpretations in the context of the radiosity problem. If we use the one-norm, we imply that the error in the solution is proportional to the sum of the errors in all patch radiosities. The two-norm implies that errors in patch radiosities are independent random variables. Both of these norms tend to smear out the effect of a single large error.

The eye is not so forgiving! In a scene where a single patch's radiosity is significantly in error, the human eye will pick it out right away, even though the one- or two-norm will show a small overall error. The infinity-norm, or max-norm, gives us the maximum error in any patch radiosity, thus more closely mimicking what the eye does. Similarly, the `link_error` measure is the result of a max-norm type operation. It is only because these two error quantities are arrived at via similar means that it makes sense to directly trade them off against one another.

Steps 30 through 33 are performed only if the system solution has been changed by the preceding steps. Their purpose is to update the link error for all links. Note that the link error estimate changes when patch radiosities change. This was explained in the section titled "Flaw in area/form factor threshold reasoning" on page 55. When the error for all links in the heap changes, the ordering imposed by the heap structure is no longer valid. Therefore after each solution refinement phase, the link heap must be "reheapified." A critical observation here is that rebuilding a heap from an unordered array is $O(N)$, not $O(N \log N)$ as is popularly believed (see [Cormen 90], page 145). Therefore, the complexity of the reheapify operation is asymptotically similar to that of the link refinement and system solution steps.

The While loop from step 12 to step 34 continues to run, alternately refining links and solving until the requested number of links have been formed. Then, the final leaf-level patch geometry is written to an answer file, and the algorithm terminates.

5.3 Observations

After reviewing Algorithm 11, we are in a position to make some observations which will be useful in making it parallel. The first two observations concern the link heap, one of the two major data structures in the algorithm. Algorithm 11 refines *one* link from the link heap at a time. With a large number of links in the link heap, many of the top links in the heap will be subdivided before the next solution step. Why not take several links off the heap at once and subdivide them all as a batch? This would save $L - 1$ reheapify steps, where L is the number of links in a batch. The possibility exists, however, that some links in the batch would subdivide into links which themselves would otherwise have been immediately split. These links, which would normally have been further refined in an imminent *refinement step* will have to wait for the next *batch* to be split. Thus, some links which would have been split had we refined them one at a time will not be refined, and some links which would not have been split will be. Experimental evidence, to be presented later, shows that this effect is not a serious problem provided the batch size is kept reasonably small.

The serial Algorithm 11 maintains a single monolithic link heap so that the link with the largest error can be removed at each step. Once we have decided to split links in batches, there is no reason for the heap to remain a monolithic structure. Indeed, it may be broken into a number of smaller heaps, and distributed across the processors in a parallel machine. Link subdivision may then proceed locally on each processor from its local heap. The exact nature of this decomposition will be discussed below in Section 5.4.

Principle 1 (Link heap): The link heap may be broken into several smaller link heaps, and these “subheaps” distributed across the processor array. Links from each subheap may be refined separately to increase parallelism.

The other major data structure in the hierarchical radiosity algorithm is the hierarchy of patches itself. All hierarchical vector operations involve calculations at the leaf level. Hierarchical matrix-vector multiply, as we will show later, requires calculations involving every link in the link heap, and every node in the hierarchy. Furthermore, these calculations must be carried out in a partial ordering that is most conveniently satisfied by preorder and postorder tree traversals. For a tree traversal to make sense, a connected path must exist from root to leaf. Thus, any decomposition strategy we develop for the patch hierarchy must be conformal with tree traversals.

Principle 2 (Hierarchy decomposition): Any patch hierarchy decomposition strategy must leave a traversable tree structure intact in every processor. Furthermore, the decomposition strategy must not incur any unnecessary interprocessor communication to conduct a top-down or bottom-up traversal.

5.4 Identifying Sources of Parallelism

The first step in making any algorithm parallel is to identify potential sources of parallelism to be exploited. The second step is deciding upon decomposition strategies for the data structures and control structures which have been targeted in the first step. There are two general classes of parallelism: *data parallelism* and *operational parallelism*. Data parallelism is exploited by decomposing and distributing data structures across processors. Operational parallelism is exploited by decomposing and distributing operations across processors.

5.4.1 Data parallelism

As mentioned earlier, there are two major data structures in the hierarchical radiosity algorithm: the link heap, and the patch hierarchy. Both of these data structures are candidates for distribution across the processing elements (henceforth referred to as PEs) of a parallel computer. We must decide which of these data structures, if not both, should be distributed. We base our decision on the ratio of links to patches, as per the arguments in the section titled “Analysis of time complexity” on page 28, the section titled “Analysis of time complexity” on page 36, and [Hanrahan 91]. All of these arguments state that the ratio of the number of links to the number of patches in a hierarchical rendering is $O(1)$. Thus, neither data structure will ever dominate the other in terms of the total amount of memory consumed. Therefore, *both* the link heap, and the patch hierarchy must be distributed. A single link and a single hierarchy node are respectively *atomic units* of each data structure, and will not be decomposed further.

5.4.2 Operational parallelism

Analysis of a serial version of the hierarchical algorithm has shown that there are four main tasks or task classes which must be made parallel. They are: link subdivide, link heap reheapify, operate on hierarchical vectors, and hierarchical matrix-vector multiply. All of these operations

are $O(N)$ with respect to the number of links and are therefore the asymptotically-limiting components of the algorithm.

5.5 Data Decomposition Strategy

5.5.1 Node hierarchy

A link will require data from two different nodes in the patch hierarchy when used in the matrix-vector multiply operation. In order to minimize interprocessor communication, we could require that a PE owning a certain link also own the hierarchy nodes at *both* ends of the link. However, we know that the patch hierarchy is fully connected via links at all times, therefore a PE will own hierarchy nodes which are also owned by other PEs. This is impractical due to the ambiguity introduced in determining which PE or PEs update the duplicated hierarchy nodes. Also, extra work would be required to keep all duplicated versions of a hierarchy node up to date. Thus, requiring a PE to own the hierarchy nodes at both ends of all links its owns is impractical. A compromise is requiring a PE to own all the hierarchy nodes on only *one end* of its links.

Principle 3 (Relationship between owned links and hierarchy nodes): Regardless of how the link heap is decomposed, we require a PE to own all the hierarchy nodes on the (arbitrarily-defined) "left" end of all owned links.

To derive more desirable properties of a patch hierarchy subdivision scheme, we consider the act of subdividing a link. Suppose a given PE owns some portion of the link heap and some portion of the patch hierarchy, according to some unspecified decomposition strategy. We also assume, in keeping with Principle 3 and without loss of generality, that a PE owns the hierarchy nodes corresponding to the "left" ends of all owned links. When a link is pulled off the local link heap for subdivision, several things could happen depending on the specifics of the hierarchy decomposition. Refer to Table 5 and Figure 19 for illustrations of the four cases of parallel link subdivision.

Table 5: Situations in parallel link subdivision

		Node ownership	
		<i>Owns</i> daughters of left node	<i>Does not own</i> daughters of left node
Subdivision action	Subdivide <i>left</i> end of link	<p>Case 1</p> <ul style="list-style-type: none"> Local link subdivision Links remain local 	<p>Case 2</p> <ul style="list-style-type: none"> Subdivide patch and send to owning PE Links must migrate to owners of left daughters
	Subdivide <i>right</i> end of link	<p>Case 3</p> <ul style="list-style-type: none"> Tell owner of right end to subdivide and send to owning PE Links remain local 	<p>Case 4</p> <ul style="list-style-type: none"> Tell owner of right end to subdivide and send to owning PE Links remain local

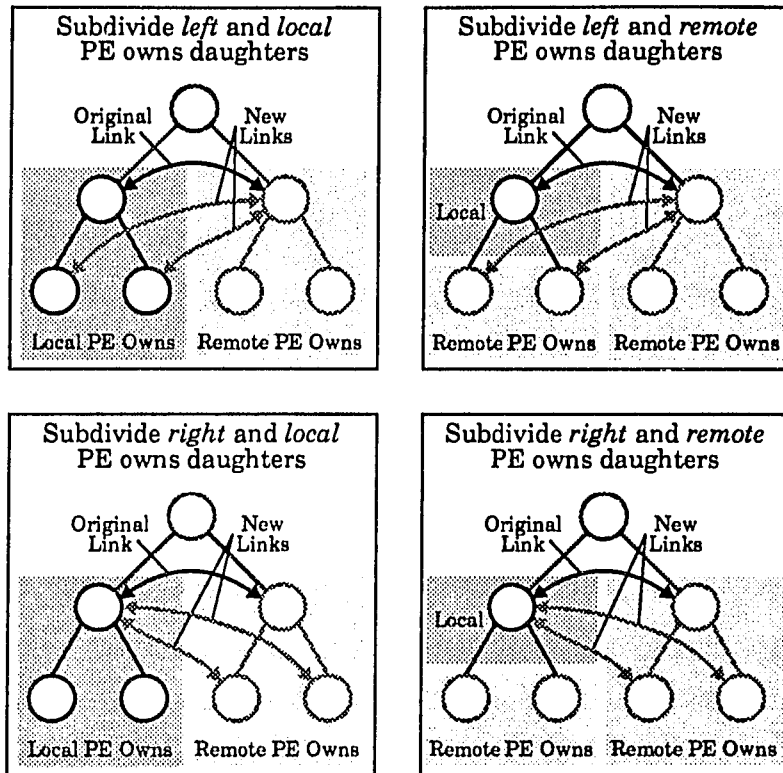


Figure 19: Cases of parallel link subdivision

Case 1 requires no internode communication at all; all operations are local to a processor. Case 2 incurs potentially four messages to two other PEs: two to send new patch geometry data, and two to migrate the new links to their owners. Cases 3 and 4 each potentially require three messages to be sent: one from the owner of the link to the owner of the patch to be split, and two from that PE to the owners of its daughters. One has no control over whether the left or right end of a link is split. One does have control over whether the owner of a particular hierarchy node also owns the children of that node as well. This divides Table 5 into two groups: Cases 1 & 3, and Cases 2 & 4. Assuming that it is equally likely that the left and right end of a link is split, the average number of messages for Cases 1 & 3 is 1.5 messages while the average number of messages for Cases 2 & 4 is 4 messages. The choice seems clear.

Principle 4 (Hierarchy locality): The PE which owns a given hierarchy node should also own one or both daughters of that node. This is equivalent (in the case of both) to saying that the patch hierarchy should be distributed by subtrees. For the case of a PE owning both daughters, we shall refer to this subtree as an *owned subtree*.

Regardless of what specific data decomposition is chosen for the hierarchy, it must provide a good load balance both in terms of data volume and computation volume. If it does not, the algo-

rithm will poorly utilize the processors of the parallel machine. Principle 3 effectively ties together the decompositions of the link heap and the patch hierarchy, so if one is load-balanced, the other will be. Principle 4 places preferences on how hierarchy nodes should be clustered on a PE, so the patch hierarchy decomposition will drive the link heap decomposition.

Suppose we choose a level (counting from the root, starting at 0) in the patch hierarchy, and call it *dlevel*, for *distribution level*. We will distribute the subtrees rooted at level *dlevel* across the processors in our parallel computer. This type of distribution satisfies Principle 4. But what of the hierarchy nodes between the root and *dlevel*?

5.5.1.1 Hierarchy decomposition method 1

A “first-blush” attempt at resolving this question might be to duplicate the patch hierarchy above *dlevel* to all PEs in the system. This duplication will also satisfy Principle 2. All processors would own and update all hierarchical vector elements and links associated with all duplicated hierarchy nodes. Such a scheme has the advantage of simplicity, and a low overhead for communication. Its disadvantages, however, are crushing. First, such duplication of hierarchy nodes and links imposes an inherent scalability problem. We know that

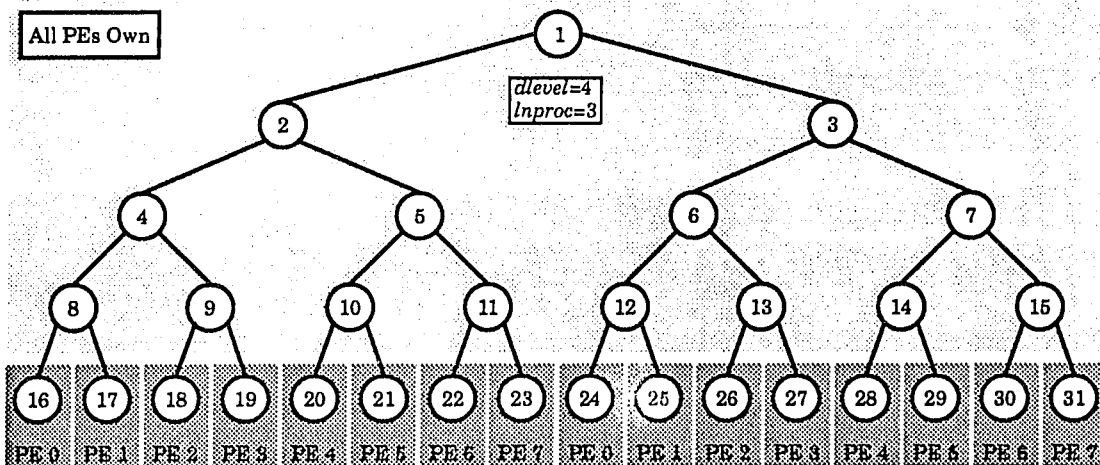


Figure 20: Hierarchy decomposition method 1

$$dlevel \geq \lnproc \quad (56)$$

where $\lnproc = \lceil \log_2(nproc) \rceil$, and $nproc$ is the number of PEs in the system.

If the inequality in (56) were not satisfied, then there would not be enough subtrees to distribute to all PEs. The total amount of memory consumed by the duplicated hierarchy nodes is:

$$\begin{aligned}
M_{hnode} \times nproc \times (2^{dlevel+1} - 1) &\geq M_{hnode} \times nproc \times (2^{lnproc+1} - 1) \\
&= M_{hnode} \times nproc \times (2 \times nproc - 1) \\
&= O(nproc^2)
\end{aligned} \tag{57}$$

where M_{hnode} is the amount of memory consumed by one hierarchy node.

Under this decomposition scheme, the total amount of memory consumed by the algorithm will be on the order of $nproc^2$, which clearly makes the algorithm unscalable. This alone is enough to disqualify such a decomposition from consideration.

Note that the hierarchy nodes in Figure 20 have been renumbered. This new node numbering scheme has the following properties: the left daughter of a node has an ID number twice that of its parent, the right daughter has an ID number twice that of its parent plus one, and the nodes at level L in the hierarchy are numbered from 2^L to $2^{L+1} - 1$ inclusive. Also, if the ID number is viewed as a binary number, it has a specific form: a 1 in bit L , followed by L unique position-determining bits. That is, the ID number for a hierarchy node completely encodes its absolute position in the hierarchy. One simply works from left to right in the L bits to the right of the leading 1 bit, and treats a 0 bit as "left" and a 1 bit as "right." By following the path specified in this interpretation down from the root node, one arrives at the hierarchy node. For example, let us consider node 25 in Figure 20. The decimal number 25 is 11001 in binary. The four bits to the right of the leading 1 bit say to follow a path of right-left-left-right down the hierarchy. If we follow this path from the root, we arrive at node 25. Although not particularly important now, this property will be of critical importance later.

This hierarchy decomposition method has an additional drawback. Consider what happens at the beginning of Algorithm 11. A single link is pushed onto the link heap and refined. Under the current hierarchy (and therefore link heap) decomposition strategy, all PEs in the system would refine exactly the same set of links until one of them refined a link-end below $dlevel$ in the hierarchy. Thus, there is *no parallelism whatsoever* until links are split below $dlevel$. In a system with only a few processors, this might not be a problem since $dlevel$ will be small. On a larger system, however, the algorithm might never run long enough to split a single link below the distribution threshold! This observation shows further unscalable behavior in the hierarchy decomposition method under consideration.

5.5.1.2 Hierarchy decomposition method 2

In an effort to reduce the amount of hierarchy node and link duplication, we may designate a PE to own a hierarchy node above $dlevel$ if and only if some portion of the subtree rooted at the node is owned by that PE. An illustration of this modified hierarchy decomposition method is shown in Figure 21. This new decomposition method has a number of advantages over the previous method. Redundant storage for hierarchy nodes is decreased significantly (this will be quantified shortly). Parallelism above $dlevel$ is increased because not as many links are duplicated on multiple processors. Parallelism increases gradually as one proceeds down the hierarchy; this is

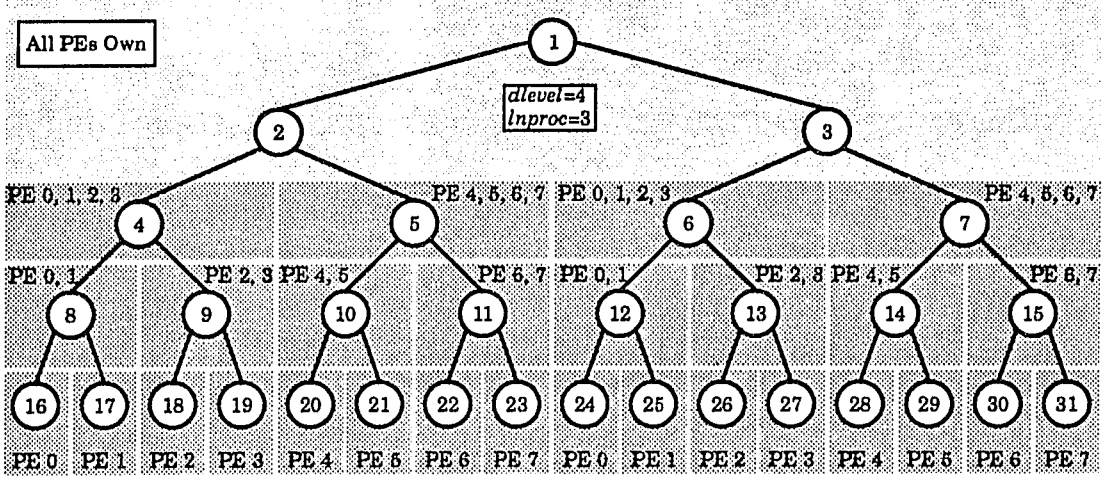


Figure 21: Hierarchy decomposition method 2

in stark contrast to method 1 where there is no parallelism at all until a link is subdivided below $dlevel$.

Before we analyze the memory consumption of method 2, we first notice that there are 2^L hierarchy nodes at level L . We may also notice that the number of PEs on which a hierarchy node is stored is $2^{dlevel-L}$. Now, the total amount of memory consumed by the hierarchy above $dlevel$ is

$$\begin{aligned}
 M_{hnode} & \left[\sum_{L=0}^{dlevel-lnproc} nproc \times 2^L + \sum_{L=dlevel-lnproc+1}^{dlevel} 2^{dlevel-L} \times 2^L \right] \\
 & = M_{hnode} \left[nproc \times (2^{dlevel-lnproc+1} - 1) + \sum_{L=dlevel-lnproc+1}^{dlevel} 2^{dlevel} \right] \\
 & = M_{hnode} [nproc \times (2^{dlevel-lnproc+1} - 1) + (lnproc - 1) 2^{dlevel}] \\
 & \geq M_{hnode} \left[2 \times nproc \times \frac{2^{lnproc}}{2^{lnproc}} - nproc + lnproc \times nproc - nproc \right] \\
 & = M_{hnode} \times lnproc \times nproc \\
 & = O(nproc \times \log(nproc)) \tag{58}
 \end{aligned}$$

The total amount of memory consumed by the duplicated nodes is much less than in method 1, especially for a large number of processors. However, there is still the duplication of hierarchy

nodes, and the corresponding links associated with these nodes. There are other drawbacks, which will become clear in subsequent sections when load-balance issues are studied.

5.5.1.3 Hierarchy decomposition method 3

The primary failing of methods 1 and 2 is their inability to deal with parallelism in the patch hierarchy above *dlevel*. Both of these are strongly affected by the duplication of hierarchy nodes. Solving the duplication problem would seem to herald a major step forward. If we start at the level above *dlevel*, and work our way up, assigning ownership of each node to the PE owning the left daughter, we have the decomposition shown in Figure 22.

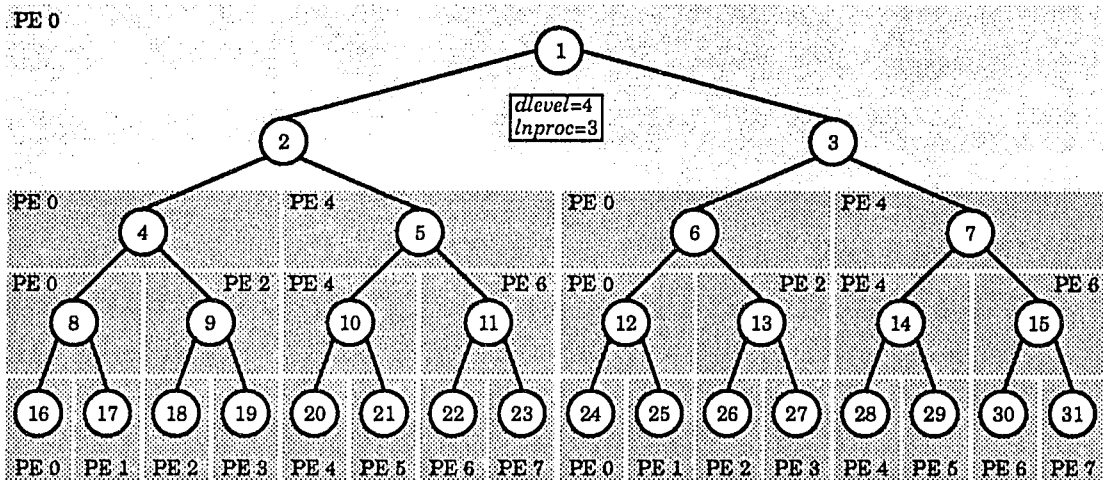


Figure 22: Hierarchy decomposition method 3

Figure 22 shows a hierarchy decomposed in such a way that there is no duplication of hierarchy nodes across PEs. There is just one problem with this decomposition; it violates Principle 2. We can solve this problem too by asserting that a PE owns all nodes in the hierarchy above *dlevel* which have descendants owned by that PE. Such a duplication is identical to that shown in Figure 21 with one major difference: every hierarchy node has a unique *owner*. A particular hierarchy node may exist on multiple PEs, but it is only owned by one. The instances of the node on non-owner PEs are merely placeholders in the hierarchy structure; they contain no data, and they have no operations performed upon them. Note that since duplication (in ownership) of hierarchy nodes has been eliminated, so has duplication of links.

Eliminating duplicate node ownership exposes the major flaw in this decomposition. Notice in Figure 22 that PE 0 owns 9 hierarchy nodes on or above *dlevel*. Also note that PE 7 owns only 2. This load imbalance only gets worse as *dlevel* is increased for a constant *lnproc* because all nodes in the first *dlevel*-*lnproc* levels are owned by PE 0. Since ownership of hierarchy nodes dictates ownership of links (by Principle 3), there will be a significant imbalance in the distribution of links, and hence in the amount of work for each PE. Even though all links will eventually be

refined below $dlevel$, performance will suffer early on. This is precisely the failing of methods 1 and 2 which we had hoped to fix.

5.5.1.4 Hierarchy decomposition method 4

Instead of numbering the leaf nodes as shown in Figure 22, we can renumber them to cause a more even distribution of nodes further up the hierarchy. In the arrangement shown in Figure 23, we can see that PE 0 now owns only 6 hierarchy nodes, while PE 7 still owns 2. The burden on PE 0 has been significantly reduced, even for this small number of processors. With a larger number of processors, the advantage becomes greater. This numbering is not arbitrary, but rather cycles through the bits of the owning PE number and flips the bit for the right daughter of a node, and does not flip it for the left daughter of a node. Note that if a PE owns node p , it also owns the left daughter of p . Total memory consumption for decomposition method 4 is still $O(nproc \times \log(nproc))$ due to the necessity of placeholder nodes.

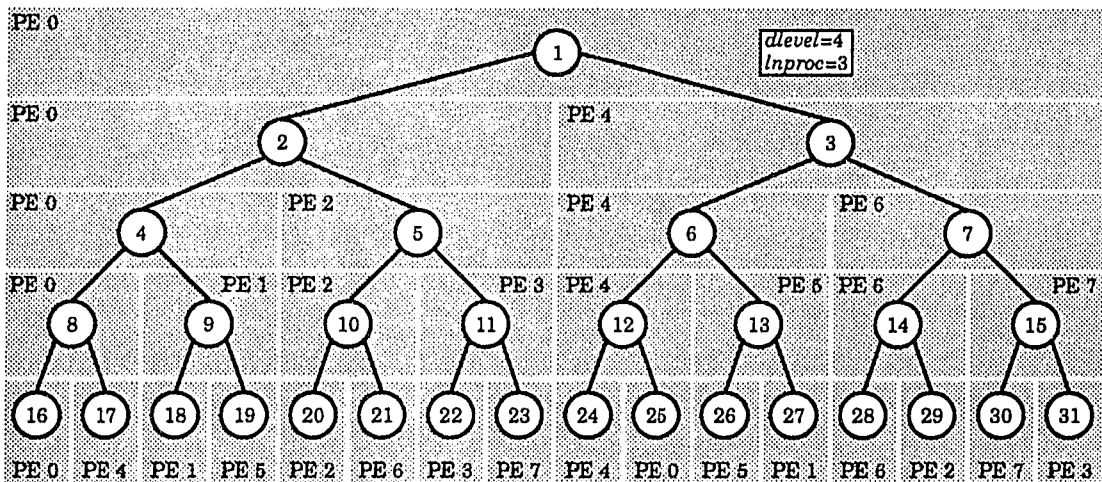


Figure 23: Hierarchy decomposition method 4

We may best derive the ownership of a hierarchy node using operations on binary numbers. The owning PE number is constructed in the following way using bits from its ID number. Recall that the ID number is a 1 bit followed by $dlevel$ position-determining bits, which we will call G , followed by some number of irrelevant bits, X . The G bits encode a path from the root to a node at $dlevel$ in the hierarchy. The X bits encode a path below $dlevel$, and are therefore not needed. For some nodes, all of the G bits may not be present; nodes above $dlevel$ have fewer than $dlevel$ available G bits. Take all available G bits and pad with 0 bits on the right until there is a multiple of $lnproc$ bits. Finally, take all groups of $lnproc$ bits and exclusive-OR them together to obtain the owning processor number.

Example: Consider a system where $dlevel=7$, $lnproc=3$ and $ID=101100$ (binary). First, we excise the leading 1 bit, and take up to 7 ($dlevel$) bits to the right of it. There are only 5 bits to be

had: 01100. We pad these bits with 0's on the right until we have a multiple of 3 (*lnproc*) bits: 011000. This 6 bit number is broken into two groups of 3 (*lnproc*) bits and XOR-ed together to form the owning PE number: $011 \oplus 000 \rightarrow 011$. Thus, PE 3 owns the hierarchy node.

5.5.2 Link heap

Principle 1 and Principle 3 effectively define exactly how the link heap must be decomposed. Let us examine this decomposition further and evaluate its suitability in terms of ease of implementation, efficiency, and load balance.

The links contained in the link heap are referenced by two processes: link refinement, and hierarchical matrix-vector multiply. The link refinement process, as stated in the section titled "Node hierarchy" on page 69, acts only on links which are local to a PE, and therefore imposes no constraints upon the link heap decomposition scheme. The same section establishes that it is impossible for a PE to uniquely own the hierarchy nodes at both ends of all links it owns. Some amount of communication with other PEs is necessary. On the positive side, the ownership of hierarchy nodes, and therefore the hierarchy decomposition, does not change as the algorithm proceeds. Link ownership and the link heap decomposition also do not change as a consequence.

There is some correlation between the load balance associated with the patch hierarchy and the load balance associated with the link heap. Ideally, we want them *both* to be well load balanced. If each patch in the hierarchy with a constant number of other patches, then load balancing the hierarchy would automatically load balance the link heap. This is not realistically the case since geometrical effects and bright light sources cause a great number of links to concentrate on a few patches in some circumstances. This will be illustrated in greater detail in the section titled "Results and Analysis" on page 91.

5.6 Critical Operations

The previous section defines specific architectural mappings for the key data and control structures. With this in place it is appropriate to expound on the exact structure of several critical operations that the algorithm performs.

Until now, no mention has been made of any specific parallel architecture. It is now necessary to do so because the machine architecture will have a large influence on the structure of the kernel operations to be discussed below. Differing machine grain sizes or memory models would warrant alternate design choices for kernel algorithm structure.

The machine architecture chosen for the first part of the parallel hierarchical radiosity algorithm is the nCUBE 2. It uses a proprietary CISC microprocessor in a multiple-instruction multiple-data (MIMD) hypercube interconnect which is scalable from 4 to 8192 processing elements (PEs). Each PE has a 64-bit internal architecture including registers, on-chip floating point hardware, and on-chip communications channels. Memory may range from 1 MB to 32 MB per PE. Each PE is capable of a theoretical maximum floating-point performance of about 3 single-precision MFLOPS (measured in terms of multiply-add operations where multiply and add each count

as one FLOP). The peak communication bandwidth between PEs is 2 MB per second in each direction and on each communication channel.

5.6.1 Random all-to-all communication

The parallel link refinement, parallel hierarchical matrix-vector multiply, and reheapify operations will all require sending and receiving randomly distributed short messages on all PEs. These messages could be sent individually, and the hardware left to deal with routing them where they need to go. Such a policy is disastrous for two reasons. First, short messages are notoriously inefficient on the nCUBE and similar machines due to message startup latency. Long messages are greatly preferable to short ones so that latency can be amortized over a longer actual transmission time. The second reason is contention. If thousands of short messages suddenly flooded the communication channels of the hypercube, there would be many messages competing for the same physical communication channels. Such contention is almost always disastrous, especially when the communication pattern is random.

A better method is to alternately exchange packets between pairs of hypercube neighbors for each hypercube dimension. This way, there is no contention whatsoever for communication channels, and messages of the longest possible length are used.

Algorithm 12 takes as input two buffers, and the length of these buffers. The first buffer, called `dests`, contains a destination PE number for the corresponding element in the `keys` buffer. The algorithm assumes that it is called on all PEs at once, and that there are a power of 2 PEs, but does not assume that the length of each buffer is the same on all PEs. As before, `iproc` is the current PE number, and `nproc` is the number of PEs in the system.

Since all communication in Algorithm 12 is in the form of send-receive pairs, PEs come into synchrony during its execution. Furthermore, if one PE is the source or destination of more messages than its neighbor during a given phase, the other PE must wait for it to complete its transmission before it can continue. Thus load balance in terms of message volume is crucial to the performance of Algorithm 12. In terms of the hierarchical radiosity algorithm, we must balance the size of individual link heaps and the distribution of PEs to which the link owner connects. The first condition will be satisfied by a well-distributed patch hierarchy. The second condition is difficult to control, although alternatives exist and will be discussed later in this chapter.

5.6.2 Link refinement

Cases 1 and 3 in Table 5 introduce the basic structure of the link parallel link refinement process. As a starting point, let us consider the exact structure of the serial link refinement algorithm. Algorithm 13 is straightforward since all data necessary for link refinement are immediately available. When the link heap is decomposed across a set of PEs, however, there are cases where all data needed to split a link lies on more than one PE.

Step 1 provides the first situation where all PEs do not have all the data they need. The variable `numlinks` in the serial algorithm is simply the number of links in the one link heap. In a parallel implementation, there is a link heap on every PE and each one potentially is a different

```

( Initialize bit mask for each hypercube dimension )
mask = floor(nproc / 2)
( Loop for each hypercube dimension )
While mask ≥ 1
  ( Separate dests and keys into two groups: one that )
  ( stays on this PE, and one that should be sent off.)
  j = k = 0
  For each element i in dests
    If (dests[i] AND mask) = (iprocs AND mask)
      dests[j] = dests[i]
      keys[j] = keys[i]
      j = j + 1
    Else
      tmpdests[k] = dests[i]
      tmpkeys[k] = keys[i]
      k = k + 1
    End if
  End for
  ( Compute a hypercube neighbor. )
  neighbor = iproc XOR mask
  Send tmpdests to PE neighbor
  Send tmpkeys to PE neighbor
  Receive new dests from neighbor into end of dests vector
  Receive new keys from neighbor into end of keys vector
  Set length of dests and keys to j+number received from neighbor
  mask = floor(mask / 2)
End while

```

Algorithm 12: All-to-all communications

size. Global communication is therefore necessary to add all link heap sizes. Also, the variable `link_error` is multifarious in a parallel implementation. It contains the estimated coupling error in the link at the top of the link heap. Its value will potentially be different on each PE in a parallel implementation. Since much global communication will be necessary inside the `while` loop, it is necessary to have all PEs execute the body of the `while` loop whether or not they take part in splitting any links (see Section 5.6.1). Therefore, the global maximum of `link_error` must be computed, and used in the test in Step 1.

Steps 7-11 may always be performed without communication on the PE owning the link because all composites and initial polygons reside on every PE. Note that this only consumes $O(1)$ memory on each PE because the number of initial polygons is $O(1)$.

Steps 14-17 split into four cases in a parallel implementation. In the following, we shall represent the left and right ends of a link with p and q , respectively. Recall that a PE owns a link if and only if it owns the hierarchy node p . The four cases are described below and in Table 6

Case 1: A PE owns both daughters of p as well as node q . This happy circumstance occurs only when node p lies on or below $dlevel$, and the PE happens to own node q as well. A subdivision

```

1.  While (numlinks < reqlinks) and (link_error < solv_error)
2.      Pop a link from the link heap
3.      Set p = left end of link
4.      Set q = right end of link
5.      { If the link is a composite self-link, split 3 ways. }
6.      If (link is a self-link)
7.          Subdivide patch p
8.          Form and initialize left↔left link
9.          Form and initialize left↔right link
10.         Form and initialize right↔right link
11.         Push all non-zero links onto heap
12.     { Decide whether to subdivide p or q. }
13.     Else if
14.         ((p is composite) and (q is not composite)) or
15.         ((area of p > area of q) and (p is composite)) or
16.         ((area of p > area of q) and (q is not composite))
17.         Subdivide patch p
18.         Form and initialize left(p)↔q link
19.         Form and initialize right(p)↔q link
20.         Push non-zero links onto heap
21.     Else
22.         Subdivide patch q
23.         Form and initialize p↔left(q) link
24.         Form and initialize p↔right(q) link
25.         Push non-zero links onto heap
26.     End if
27.     Set link_error to error in the link at the top of heap
28.     Set numlinks to the link heap size
29. End while

```

Algorithm 13: Serial link refinement

action requires no communication. Node p is simply subdivided, two new links formed, and then pushed onto the local link heap.

Case 2: A PE owns both daughters of p but does not own q . In order to split the link, the PE must acquire geometry data about node q from its owner. This involves a global communication step, as many PEs may have links with similar requirements. Then the PE may split node p , form two new links using the newly-acquired geometry data about node q , and push them onto the local link heap.

Case 3: A PE owns only one daughter of node p and owns node q . This case may only happen when node p lies strictly above $dlevel$. In this case, the link between the owned daughter of p and node q may be formed as in Case 1. The case of the unowned daughter is more complicated. Let us denote the unowned daughter of node p as node r . This is the first case where splitting a link produces a link which is not owned by the PE owning node p . In this case, the new link from r to q will be owned by the PE which owns node r . Since the owner of node r has no information about the original link from p to q , it must be sent a message informing it that it is the new owner of a link from r to q ; thus, the link from r to q is *migrated* from the owner of node p to the owner of

Table 6: Situations in splitting left link end in parallel

		Ownership of daughters of left end	
		Owns <i>both</i> daughters of <i>p</i>	<i>Does not</i> own both daughters of <i>p</i>
Ownership of right end	Owns <i>q</i>	Case 1 <ul style="list-style-type: none"> Local link subdivision Links remain local 	Case 3 <ul style="list-style-type: none"> One link remains local One link must migrate to owner of unowned daughter of <i>p</i>
	Does not own <i>q</i>	Case 2 <ul style="list-style-type: none"> Get geometry information from owner of <i>q</i> Links remain local 	Case 4 <ul style="list-style-type: none"> Get geometry information from owner of <i>q</i> One link remains local One link must migrate to owner of unowned daughter of <i>p</i>

node *r*. Since the geometry of node *q* is available immediately for the link from *r* to *q*, we may send it along with the migration message so the owner of *r* will not have to request geometry data on *q* later.

Case 4: A PE owns only one daughter of node *p* and does not own node *q*. As with Case 3, this case may only happen when node *p* lies strictly above *dlevel*. Even more communication is required here than in Case 3. Since node *q* is not available to the owner of node *p*, a separate communication step is necessary to obtain geometry information for node *q*. The link from the owned daughter of *p* may then be formed and pushed onto *p*'s local link heap. The *r* to *q* link migration may then be conducted exactly as in Case 3 to complete the link subdivision.

Steps 19-22 split into two cases. Since the left end of both refined links will still be *p*, they will both reside on the PE which owns node *p*.

Case 1: The owner of node *p* also owns node *q*. In this case, no communication is necessary. The PE simply subdivides node *q*, forms new links, and pushes them onto its local link heap.

Case 2: The owner of node *p* does not own node *q*. Here, the owner of *p* must send off a splitting request to the owner of *q* and receive the geometrical information about the new daughter patches. The new links may then be created and pushed onto the local link heap. One may ask what happens when the daughters of node *q* are not owned by the owner of *q*. In this case, the owner of *q* may still obtain valid *geometrical* information about both daughters, even though it does not own them. Only geometrical information is needed to form new links; ownership-dependent information is only needed to update the link error. Link error estimates may be updated as a body once a whole batch of links has been split. Updating the link error estimates is the topic of the section titled "Reheapifying" on page 83.

With the preceding issues discussed, we may now present the complete parallel link refinement algorithm. This algorithm takes place in nine phases. Most phases involve a global communication step to route a list of packets between PEs using the algorithm described in the section titled "Random all-to-all communication" on page 77. The variable *batchsize* is used in step 7 to control how many links a PE splits at once. This quantity was discussed in the section titled "Observations" on page 67.

```

1.  Set numlinks to the global sum of all link heap sizes
2.  Set link_error to global maximum link error
3.  While (numlinks < reqlinks) and (link_error < solv_error)
4.    { Phase 1: Separate a batch of links }
5.    { into LOCAL, GEOM, and REMOTE lists. }
6.    Initialize LOCAL, GEOM, and REMOTE lists to empty
7.    For i = 1 to batchsize
8.      Pop a link from the local link heap
9.      If link is a composite self-link
10.     Put the link on the LOCAL list
11.     Else if both ends of link are owned by this PE
12.       Put the link on the LOCAL list
13.     Else if the left end of the link should be split
14.       Put the link on the GEOM list
15.     Else
16.       Put the link on the REMOTE list
17.     End if
18.   End for
19.   { Phase 2: Work on the REMOTE list by sending splitting }
20.   { requests to the PEs owning the "right" ends.           }
21.   For each link in REMOTE list
22.     Synthesize a splitting request packet to owner of link
23.   End for
24.   Route all splitting packets to their owning PEs
25.   { Phase 3: Service splitting requests and send }
26.   { geometry information back to sending PE.     }
27.   For each splitting request packet just received
28.     Subdivide the requested node
29.     Synthesize a splitting reply with the new geometry data
30.   End for
31.   Route all splitting reply packets back to requesting PEs
32.   { Phase 4: Split the REMOTE links using }
33.   { the data received in phase 3.         }
34.   For each splitting reply packet just received
35.     Form a new link from p to left(q)
36.     Push onto local link heap if coupling is nonzero
37.     Form a new link from p to right(q)
38.     Push onto local link heap if coupling is nonzero
39.   End for
40.   { Phase 5: Work on the GEOM list by sending geometry }
41.   { requests to the PEs owning the right ends.         }
42.   For each link in the GEOM list
43.     Synthesize a geometry request packet to owner of right end
44.   End for
45.   Route all geometry request packets to their owning PEs
46.   { Phase 6: Service requests for geometry data }
47.   { and send back to the requesting PE.         }

```

```

48.   For each geometry request packet just received
49.       Pack up the geometry of the requested node
50.       Synthesize a geometry reply packet
51.   End for
52.   Route all geometry reply packets back to the requesting PE
53.   { Phase 7: Split the GEOM links using }
54.   { data received in phase 6.           }
55.   For each geometry reply packet just received
56.       Subdivide the left end, p, of the associated link
57.       Form a link from left(p) to q
58.       If left(p) is owned by this PE
59.           Push link onto local link heap
60.       Else
61.           Synthesize a link migration packet
62.       End if
63.       Form a link from right(p) to q
64.
65.       If right(p) is owned by this PE
66.           Push link onto local link heap
67.       Else
68.           Synthesize a link migration packet
69.       End if
70.   End for
71.   { Phase 8: Split links in the LOCAL list. }
72.   For each link in the LOCAL list
73.       If link is a self-link
74.           Subdivide node at left end of link
75.           Form left(p) to left(p) link
76.           If left(p) is owned by this PE
77.               Push link onto local link heap
78.           Else
79.               Synthesize a link migration packet
80.           End if
81.           Form left(p) to right(p) link
82.           If left(p) is owned by this PE
83.               Push link onto local link heap
84.           Else
85.               Synthesize a link migration packet
86.           End if
87.           Form right(p) to right(p) link
88.           If right(p) is owned by this PE
89.               Push link onto local link heap
90.           Else
91.               Synthesize a link migration packet
92.           End if
93.       Else if left end of link should be subdivided
94.           Subdivide node p at left end of link
95.           Form a link from left(p) to q
96.           If left(p) is owned by this PE
97.               Push link onto local link heap
98.           Else
99.               Synthesize a link migration packet
100.          End if
101.          Form a link from right(p) to q
102.          if (right(p) is owned by this PE

```

```

103.         Push link onto local link heap
104.     Else
105.         Synthesize a link migration packet
106.     End if
107. Else
108.     Subdivide node  $q$  at right end of link
109.     Form a new link from  $p$  to left( $q$ )
110.     Push onto local link heap if coupling is nonzero
111.     Form a new link from  $p$  to right( $q$ )
112.     Push onto local link heap if coupling is nonzero
113. End if
114. End for
115. { Phase 9: Migrate links from previous phases. }
116. Route link migration packets to owning PEs.
117. For each link migration packet just received
118.     Form a link from data in link migration packet
119.     Push link onto local link heap
120. End for
121. Reheapify all local link heaps
122. Set link_error to global maximum link error
123. Set numlinks to the global sum of all link heap sizes
124. End while

```

Algorithm 14: Parallel link refinement

Batch splitting of links is worthy of special mention in Algorithm 14. Not only does it obviate the need for a single unified link heap, it has the side effect of making it unnecessary to compute link error estimates for newly-created links. In Algorithm 13, we assume that each time a link is pushed onto the link heap, a heap insertion is performed, thus preserving its heap structure. Algorithm 14 does not need to know link error values until it is completely finished refining a batch of links. Thus, the update of link error estimates is also batched as a consequence of splitting links in batches.

5.6.3 Reheapifying

As mentioned above, the parallel link refinement algorithm refines links in batches, and also adds links to the local link heaps in batches. There is insufficient local data on a PE to form the link error estimate when links are formed. Rather than perform a communication step when the link is formed in order to update the link error estimate, we may delay all such communications until the end of the batch and do them all at once. Recall from the section titled “Flaw in area/form factor threshold reasoning” on page 55 that link error depends on the brightness and reflectivity of the patches at both ends of a link, as well as the link’s coupling value.

As with Algorithm 14, Algorithm 15 assumes that all PEs execute it at the same time due to the global communication. Link heap load balance and link connectivity influence the efficiency of Algorithm 15 for the same reason they influence the efficiency of Algorithm 14.

5.6.4 Hierarchical vector operations

All operations on one hierarchical vector or between two hierarchical vectors are handled largely the same as if the vectors were not hierarchical in nature. Only the nodes at the leaf level

```

{ Update the link error estimate for all links. }
For each link L in local link heap
  Set p to node at left end of link L
  Set q to node at right end of link L
  { Phase 1: Handle local links and synthesize }
  { brightness/reflectivity request to owner of q. }
  If q is a local node
    Update the link error estimate using local data.
  Else
    Synthesize a brightness request packet to owner of q
  End if
  Route brightness request packets to their owners
  { Phase 2: Service remote brightness requests. }
  For each brightness request packet just received
    Locate the requested hierarchy node
    Synthesize a brightness reply packet to requestor
  End for
  Route brightness reply packets to their requestors
  { Phase 3: Update remainder of local links using }
  { remote brightness information just received. }
  For each brightness reply packet just received
    Update corresponding link's error estimate using remote data
  End for
End for
{ Reheapify on the now-valid link error estimates }
Perform a standard reheapify operation on the local link heap

```

Algorithm 15: Parallel reheapify

of the hierarchy are operated upon. They represent the smallest level of subdivision of any polygon, and are treated as independent variables. As with operations on a traditional vector of numbers, order is not important in a hierarchical vector operation. Any method of traversing the hierarchy may be used so long as it visits each of the leaf nodes exactly once. Values at interior hierarchy nodes are not needed by any routine other than the hierarchical matrix-vector multiply. That routine updates interior hierarchy nodes for its one vector operand as needed.

The following operations on hierarchical vectors are necessary to carry out the hierarchical radiosity algorithm: copy, initialize to a constant, add, subtract, multiply, invert (element-wise), inner product, scale by a constant, and norm.

5.6.5 Hierarchical matrix-vector multiply

Figure 18 on page 65 introduces the idea of viewing the links as a dense coupling matrix. Since both Jacobi and Conjugate Gradient iteration can be formulated in terms of matrix-vector multiply operations, it makes sense to take advantage of the $O(N)$ nature of such an operation (N is the number of links). In order to derive an algorithm for performing a hierarchical matrix times a hierarchical vector operation, let us manually work through an example using the matrix in Figure 18. The matrix in Figure 18 is a hierarchical matrix of *coupling factors*. We will require a matrix-vector multiplication by a matrix of form factors in the solution process.

In the following, we shall perform the operation $Fx \rightarrow b$ and denote by A_p the area of patch p , by C_{pq} the coupling between nodes p and q , by $F_{pq} = C_{pq}/A_p$ the form factor from p to q , and by x_p the value of the hierarchical vector x at node p . Refer to the section titled "Patch couplings and link splitting" on page 38 for a discussion of the mathematics behind link splitting. We begin with the scalar equation

$$b_0 = F_{00}x_0. \quad (59)$$

Since F_{00} is not one of the final links in Figure 18, we split it on the left and right to yield the following:

$$\begin{aligned} b_1 &= F_{11}x_1 + F_{12}x_2 \\ b_2 &= F_{21}x_1 + F_{22}x_2. \end{aligned} \quad (60)$$

We now note that F_{12} is not a final link, so we split F_{12} on the *right* and F_{21} on the *left* to yield:

$$\begin{aligned} b_1 &= F_{11}x_1 + F_{15}x_5 + F_{16}x_6 \\ b_5 &= F_{51}x_1 + F_{22}x_2 \\ b_6 &= F_{61}x_1 + F_{22}x_2. \end{aligned} \quad (61)$$

Note that when we split a link on the *right*, we expand one term into two terms in a single equation. When we split a link on the *left*, we split one equation into two equations, each with the same number of terms as the parent equation. There are now two terminal links in (61), F_{16} in the b_1 equation, and F_{61} in the b_6 equation. Note that $C_{16} = C_{61}$. These links will not be split any further. We now split link F_{15} in the b_1 equation on the left, and F_{51} in the b_5 equation on the right to obtain

$$\begin{aligned} b_3 &= F_{11}x_1 + F_{35}x_5 + F_{16}x_6 \\ b_4 &= F_{11}x_1 + F_{45}x_5 + F_{16}x_6 \\ b_5 &= F_{53}x_3 + F_{54}x_4 + F_{22}x_2 \\ b_6 &= F_{61}x_1 + F_{22}x_2. \end{aligned} \quad (62)$$

Now, all that remains to be done is to split F_{35} in the b_3 equation on the right, and F_{53} in the b_5 equation on the left. This yields the final set of equations for leaf nodes:

$$\begin{aligned} b_3 &= F_{11}x_1 + F_{37}x_7 + F_{38}x_8 + F_{16}x_6 \\ b_4 &= F_{11}x_1 + F_{45}x_5 + F_{16}x_6 \\ b_6 &= F_{61}x_1 + F_{22}x_2 \\ b_7 &= F_{73}x_3 + F_{54}x_4 + F_{22}x_2 \\ b_8 &= F_{83}x_3 + F_{54}x_4 + F_{22}x_2. \end{aligned} \quad (63)$$

Suppose that PE 0 owns links C_{11} , C_{37} , C_{38} , that PE 1 owns links C_{22} , C_{16} , C_{45} , and the hierarchy from Figure 18 is distributed across two processors as shown in Figure 24.

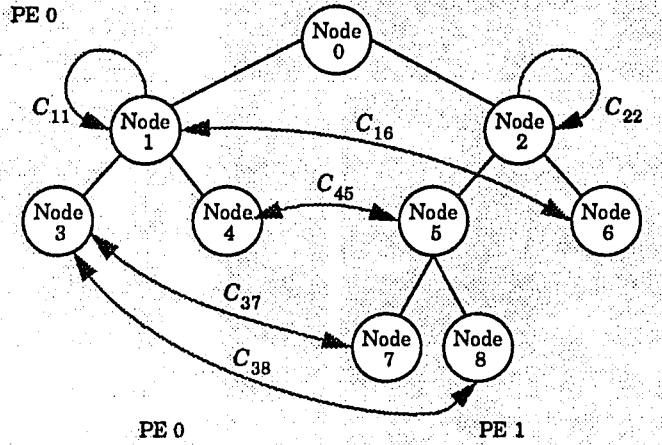


Figure 24: Decomposition of example hierarchy

Now, let us examine each of the terms in (63). Note that the terms $F_{11}x_1$ and $F_{16}x_6$ appear in the equations for both b_3 and b_4 . Also note that node 1 is an ancestor of nodes 3 and 4. The term $F_{22}x_2$ appears in the equations for all leaf nodes which are descendants of node 2. The term $F_{54}x_4$ appears in the equations for all leaf nodes which are descendants of node 5. In fact, all terms of the form $F_{pq}x_q$ are reused in all leaf descendants of node p . This observation follows directly from (44) through (48).

Let us formulate the irradiance incident upon patch p as follows:

$$b_p = \chi_p + \alpha_p + \beta_p \quad (64)$$

where:

$$\chi_p = \sum_{q \in \{L_p\}} F_{pq}x_q, \text{ or all link contributions at node } p,$$

$\{L_p\}$ is the set of all nodes to which node p is linked,

α_p is the sum of all link contributions of all *ancestors* of node p in the hierarchy, and

β_p is the sum of all link contributions of all *descendants* of node p in the hierarchy.

We may expand the α_p and β_p terms in the following way:

$$\alpha_p = \chi_{parent(p)} + \alpha_{parent(p)}, \quad (65)$$

$$\beta_p = \chi_{left(p)} + \chi_{right(p)} + \beta_{left(p)} + \beta_{right(p)}. \quad (66)$$

These recursions suggest the serial algorithm for hierarchical matrix-vector multiply shown in Algorithm 16. A few words are in order about the notation used in this algorithm. We assume that

each hierarchy node p stores several quantities: references to its mother and left and right daughter nodes, denoted by $p.parent$, $p.left$, and $p.right$, respectively; a temporary hierarchical vector element denoted by $p.t$; the multiplicand hierarchical vector element denoted by $p.x$; and the resultant product hierarchical vector element denoted by $p.v$. In vector notation, the operation performed by Algorithm 16 is $v \leftarrow Fx$. The `Hprep` function in Algorithm 16 is used to set the

```

{ Multiply hierarchical matrix contained in 'heap' by
{ hierarchical vector 'x', and place the result in 'v'. }
MatVecMult (root, heap)
{
  { Prepare x vector for subsequent use. }
  { Initialize t temporary vector to 0. }
  Hprep(root)
  { Accumulate link contributions into t. }
  For each link L in heap
    p.t += Fpq * q.x
    if (p != q)
      q.t += Fqp * p.x
  End for
  { Propagate t values up and down }
  { to form matrix-vector product. }
  Prop(root)
}
Hprep(p)
{
  p.t = 0.0
  If p has daughters
    Hprep(p.left);
    Hprep(p.right);
    p.x = (p.left.x * p.left.area +
          p.right.x * p.right.area) / p.area;
  End if
}
Prop(p)
{
  p.v = p.t
  If p has a parent
    p.v += p.parent.v
  End If
  If p has daughters
    Prop(p.left)
    Prop(p.right)
    p.t += p.left.t + p.right.t
    p.v += p.t;
  End if
}

```

Algorithm 16: Serial hierarchical matrix-vector multiply

hierarchical vector values of interior nodes to the area-weighted average of their daughters' values. This area-based summarization is appropriate for the irradiance vector because it is a vector of power *densities* rather than a vector of *powers*. Though we are only concerned with hierarchical

vector values in the leaf nodes in the final solution, the matrix-vector multiply routine requires valid data in all nodes of the multiplicand vector because the links do not just couple leaf nodes.

Making Algorithm 16 parallel is fairly straightforward. The three phases of Algorithm 16 are multiplicand vector preparation, link contribution accumulation, and partial-product propagation. All three phases require interprocessor communication, but they are independent. Let us examine them each in turn.

First, we will examine the preparation phase. The basic operation in this phase is a postorder tree traversal, with a node update involving data from each daughter. Below the distribution level, $dlevel$, such an operation is possible with no interprocessor communication because entire subtrees exist on one PE. Above $dlevel$, a PE owning node p only owns one of p 's daughters. The PE must therefore obtain data from the PE which owns the other daughter. Figure 25 shows with

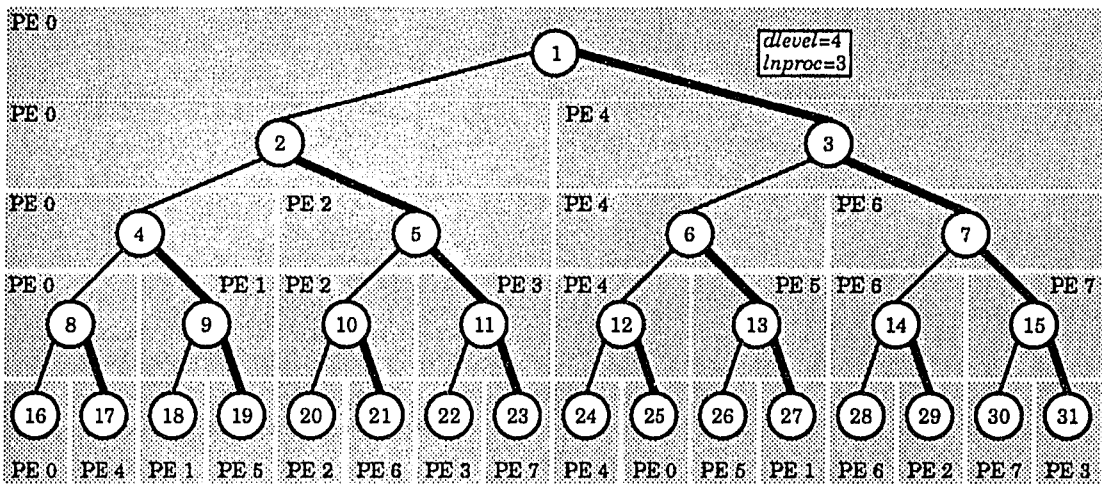


Figure 25: Loci of communication in Hprep

thick lines the relationships between hierarchy nodes where communication must be performed. A parallel version of the H_{prep} function is given in Algorithm 17. One might be curious why Algorithm 17 is in two parts. Most of the operations and most of the parallelism in the H_{prep} function lies below $dlevel$. By first collapsing just up to $dlevel$, we are able to make all PEs perform useful work in parallel in their owned subtrees. Once all PEs have done this, they may come back and collapse data from the roots of the subtrees the rest of the way up the hierarchy.

Next, we shall examine the link contribution phase of Algorithm 16. Recall that a PE owns only those links whose "left" end is also owned by that same PE. Also, observe from equation (63) that for every form factor of the form F_{pq} used, there is also an F_{qp} used. These terms come from the link contributions at the nodes connected by the link C_{pq} . Thus, for every link C_{pq} owned by a PE, there are *two*, not one, link contributions made by that link. There are three cases in the inner loop of the link contribution phase:


```

{ Collapse area-weighted sums of hierarch- }
{ ical vector 'x' up the hierarchy.      }
Hprep(root)
{
    Hprep_subtrees(root)
    Hprep_body(root)
}
{ Perform the collapse on and below dlevel. }
Hprep_subtrees(p)
{
    p.t = 0.0
    If p has daughters
        Hprep_subtrees(p.left);
        Hprep_subtrees(p.right);
        If (node p is on or below dlevel) and (this PE owns node p)
            p.x = (p.left.x * p.left.area +
                   p.right.x * p.right.area) / p.area
        End if
    End if
}
{ Perform the collapse strictly above dlevel. }
Hprep_body(p)
{
    If (node p has daughters) and (node p is strictly above dlevel)
        Hprep_body(p.right)
        Hprep_body(p.left)
    End If
    If node p has daughters
        If this PE owns the left daughter
            Receive irradiance and area of right daughter from owner
            p.x = (p.left.x * p.left.area +
                   p.right.x * p.right.area) / p.area
        Else if this PE owns the right daughter
            Send irradiance and area of right daughter to owner of node p
        End if
    End if
}

```

Algorithm 17: Parallel hierarchical vector preparation

Case 1: A link is a composite self-link in which case all data necessary to calculate its contribution to the matrix-vector product are local to the PE. A self-link contributes to the partial product on the PE owning the link only.

Case 2: A link is not a self-link, but both ends of the link are local to the PE owning the link. Here, too, all data to calculate the links contributions are local, but the link makes two partial product contributions: one to the node at the left end of the link, and one to the node at the right end of the link. In this case, exactly the same operations as in the kernel of the serial link contribution loop are executed.

Case 3: A link is not a self-link and the node at the right end of the link is owned by another PE. In this case, it is instructive to examine the two link contributions in the kernel of the serial loop to see where the data to perform each calculation resides. The blocks labeled p in Figure 26 contain data which is owned by the owner of hierarchy node p . Similarly, the blocks labeled q contain data which is owned by the owner of hierarchy node q . Note that both F_{pq} and F_{qp} are derived from the quantity C_{pq} , which is owned by the owner of node p . This layout of data suggests a three-phase update operation. First, the owners of all nodes p pack up C_{pq} and x_p values, and send them to the owners of the corresponding q 's. These PEs perform the second accumulation in Figure 26, pack up x_q values, and send them back to the owners of p . Finally, the owners of p perform the first accumulation in Figure 26.

$$\begin{array}{c}
 t_p \leftarrow t_p + F_{pq} \times x_q \\
 \mathbf{p} \qquad \qquad \mathbf{q} \\
 t_q \leftarrow t_q + F_{qp} \times x_p \\
 \mathbf{q} \qquad \mathbf{p} \qquad \mathbf{p}
 \end{array}$$

Figure 26: Locus of link contribution data

Combining all three cases leads us to the following algorithm for parallel link contribution accumulation (Algorithm 18):

The final step in hierarchical matrix-vector multiply is the propagation of link contributions, or partial products, up and down the hierarchy to form the final products. This process is very similar to that already given in Algorithm 17 for hierarchical vector preparation, except that it can be performed efficiently in one subroutine rather than two. The places where communication must be performed are the same as those in Algorithm 17. The difference being that partial products must be propagated both upward *and* downward rather than just downward.

5.6.6 Writing the answer file

As with most scientific applications, the time spent in I/O is no small portion of overall application time. Most applications, however, do not incur I/O of the same order of complexity as their computational kernel. With hierarchical radisity, link subdivision, system solving, *and* I/O are all $O(N)$. There exists the possibility that overall application time might be dominated by the typically slower I/O operations.

The answer from a radisity rendering (at least in this case) is a list of geometrical patches together with their red, green, and blue brightness values. There is no prescribed order in which these patches must be arranged in the answer file, so we may feel free to write them in whatever order is most convenient.

```

{ Accumulate partial products for hierarchical matrix-vector }
{ multiply into hierarchical vector 't' at each node.          }
Link_contrib(root, heap)
{
  { Phase 1: Local link resolution and remote request generation. }
  For each link L in local link heap
    Set p to left end of link
    Set q to right end of link
    If p == q
      p.t += Cpq * p.x / p.area
    Else if q is owned by this PE
      p.t += Cpq * q.x / p.area
      q.t += Cpq * p.x / q.area
    Else
      Synthesize contribution request packet to q with Cpq and p.x
    End if
  End for
  Route all contribution request packets to their destinations
  { Phase 2: Remote right-link-end accumulation and reply. }
  For each contribution request packet just received
    q.t += Cpq * p.x / q.area
    Synthesize contribution reply packet back to p with q.x
  End for
  Route all contribution reply packets to their originators
  { Phase 3: Remote left-link-end accumulation }
  For each contribution reply packet just received
    p.t += Cpq * q.x / p.area
  End for
}

```

Algorithm 18: Parallel link contribution accumulation

Only the leaf-level patches need be written because interior nodes in the hierarchy are simply the union of two or more leaf-level patches. We also know that all leaf-level patches are uniquely owned by a single PE due to the hierarchy decomposition scheme. It is, therefore, a simple matter for each PE to traverse its hierarchy, and format output records from the leaf-level patches that it owns. In the presence of a parallel I/O subsystem, all PEs would be able to write their completed output records at once. In the absence of a parallel I/O subsystem, all output records may be concatenated, and written to a single sequential filesystem. In both cases, output records must be written in large blocks to achieve reasonable I/O throughput. The latter case is implemented here, and even so presents no major bottleneck (See Figure 28, task "Storer").

5.7 Results and Analysis

The parallel hierarchical radiosity algorithm is implemented in approximately 7,000 lines of C++ code on an nCUBE 2 parallel supercomputer. An object-oriented approach was used to compartmentalize methods for dealing with key data structures such as: polygons, patches, composites, the node hierarchy, the link heap, hierarchical vectors, and the hierarchical coupling matrix.

```

{ Propagate partial products up and down the hierarchy }
{ to form the final hierarchical matrix-vector product. }
Prop(p)
{
  { Preorder propagation of t values down into v. }
  If node p is local
    p.v = p.t
    If (p has right daughter) and (right daughter is NOT local)
      Send p.v value to owner of p.right
    End if
    If p has a parent
      If p.parent is NOT local
        Receive parent p.parent.v value from owner of p.parent
        p.v += p.parent.v
      End if
    End if
  { Recursion }
  If p has daughters
    Prop(p.left)
    Prop(p.right)
  End if
  { Postorder propagation of t values up into t, and accum. into v. }
  If node p is local
    If (p has parent) and (p.parent is NOT local)
      Send p.t to owner of p.parent
    End if
    If (p has right daughter) and (right daughter is NOT local)
      Receive value of p.right.t from owner of p.right
    End if
    If p has daughters
      p.t += p.left.t + p.right.t
    End if
    p.v += p.t
  End if
}

```

Algorithm 19: Parallel partial product propagation

5.7.1 A visit from reality

Section 5.5 took great care to efficiently balance hierarchy nodes across a group of PEs, and distributed the links in such a way as to minimize communication and spread them as evenly as possible. This was done, however, in the absence of the knowledge of any specifics about the character of how links will be split. As we shall see, the character of how links will be subdivided, and the distribution of connectivities in the hierarchy, is highly data dependent. Figure 27 shows a sample breakdown of the time spent by each PE in each of the nine phases of Algorithm 14. The most striking feature in the graph is the disastrous imbalance in the amount of time spent in phases 4 and 7, the remote and geometry-only link splitting phases. Similar imbalance exists in the communications load across the PEs for other phases. Worse yet, no choice of *dlevel* or amount of further link refinement evens out this imbalance.

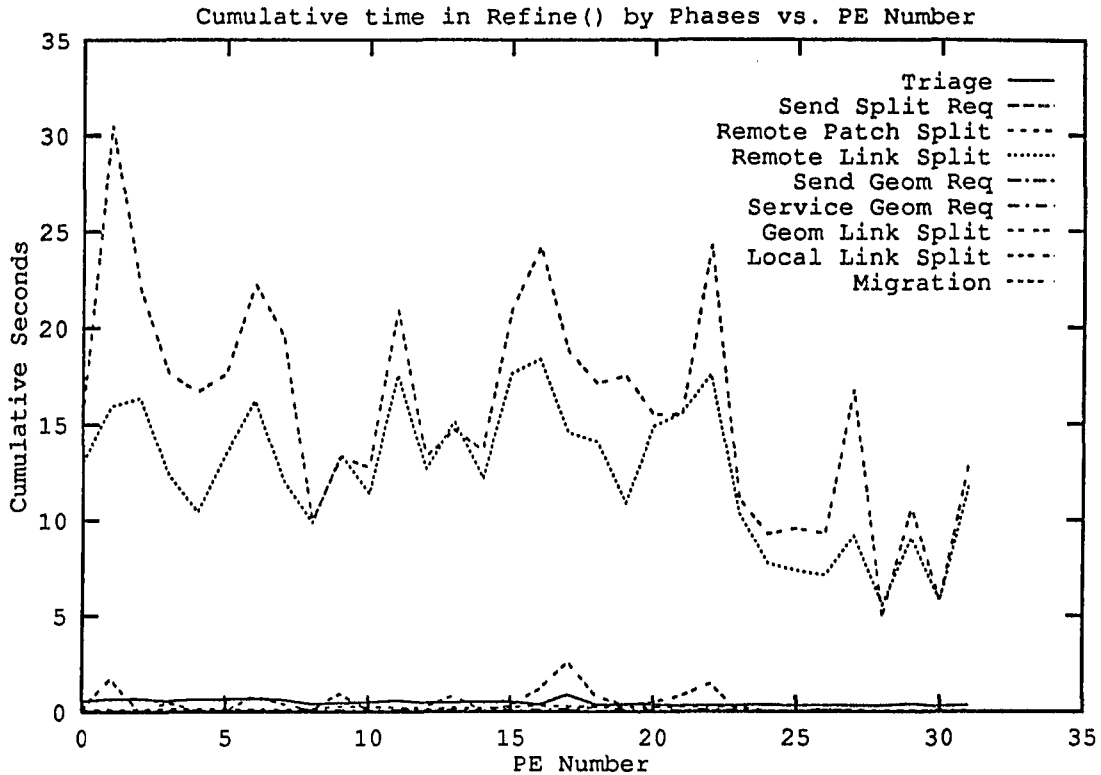


Figure 27: Link contribution phases vs. processor for original algorithm

In order to fix the problem, we must understand its causes. In this case, it is caused by the highly nonuniform nature by which links connect hierarchy nodes on different PEs. We have endeavoured to balance the number of links that each PE owns through judicious choice of a hierarchy decomposition method coupled with link ownership based on the left ends of the links. Even so, there is over a three to one ratio between the size of the largest and smallest node link heaps. This creates load imbalance in both the solver and link refinement tasks.

Since it appears that Principle 3 and Principle 4 are costing more performance due to load imbalance than they gain by locality of reference, let us consider alternatives that do not follow the axiom. One way to even out the link heaps is to abolish Principle 4 and *dlevel* and continue alternating ownership of hierarchy nodes down the hierarchy indefinitely. This would eventually even out the local heap sizes, and the left-link-end distribution across the PEs without making the communication any worse in the link refinement or link contribution algorithms. It would, however, have a disastrous impact on the communications in the vector preparation and partial product propagation algorithms. These algorithms rely on short messages at every hierarchy node, and would thereby suffer greatly in terms of performance.

A less disastrous option is abolish Principle 3, and spread the links out among the PEs evenly regardless of hierarchy node ownership. This scheme has the benefit of being able to equalize all local heap sizes and link refinement loads regardless of data-dependent effects. It also does not affect the hierarchy decomposition or the communication load of either hierarchical vector preparation or partial product propagation. It has the disadvantage of doubling the communication load of link refinement and link contribution. But it is *only* a doubling. This contrasts with a scheme which would much more than double the worst kind of communication traffic in vector preparation and partial product propagation.

The remainder of this chapter will assume revised forms of the algorithms for parallel link refinement, parallel link contribution accumulation, and parallel reheapification. These algorithms are generally simpler in form, but require more communication.

Algorithm 20 is the revised parallel link refinement algorithm. It has fewer phases than Algorithm 14, and a less confusing structure. At the same time, we may also pack up node brightness and reflectivity data with all reply packets to be used in the link error estimate. This small addition obviates the need for a reheapify step after every batch of link splittings.

Accumulating link contributions in parallel is a bit more tricky. The added complication is that now, potentially three different PEs own the data necessary to calculate the link contributions. One PE owns the coupling between nodes p and q , one PE owns node p , and one PE owns node q . The solution shown in Algorithm 21 involves two steps. First, the PE owning a particular link C_{pq} , will send the coupling and the values of p and q to the two PEs owning p and q . Then, the PEs owning p and q will exchange their respective values of x_p and x_q . The link contribution may then be completed locally on the PEs owning p and q using data from the two communication steps.

```

{ Accumulate partial products in each hierarchy node. }
{ Phase 1: Synthesize exchange packets to p and q. }
For each link L in local link heap
    Synthesize an exchange packet to owner of each end of link
End for
Route exchange request packets to their owners
{ Phase 2: Service exchange requests. }
For each exchange packet just received
    Locate the requested hierarchy node
    Synthesize a brightness packet to owner of other end of link
End for
Route brightness packets to their destinations
{ Phase 3: Calculate link contributions on p and q. }
For each brightness packet just received
    Accumulate link contribution into local hierarchy node
End for

```

Algorithm 21: Revised parallel link contributions

```

Set numlinks to the global sum of all link heap sizes
Set link_error to global maximum link error
While (numlinks < reqlinks) and (link_error < solv_error)
  { Phase 1: Remove a batch of links from the local heap. }
  For i = 1 to batchsize
    Pop a link from the local link heap
    If the link's error is less than solv_error
      Push link back onto local heap
      Exit for loop
    End if
    Place the link into the list of links to be split
  End for
  { Phase 2: Synthesize splitting and geometry requests to the }
  { owners of the nodes at both ends of each link to be split. }
  For each link L to be split
    If L is a self link
      Synthesize a single splitting request to the owner
    Else if the link should be split on the left
      Synthesize a splitting request to owner of left end
      Synthesize a geometry request to owner of right end
    Else
      Synthesize a splitting request to owner of right end
      Synthesize a geometry request to owner of left end
    End if
  End for
  End for
  Route all request packets to their destination PEs
  { Phase 3: Service requests for splittings and geometry data }
  { and pack up reply messages to the sending processor. }
  For each request packet just received
    If the request is a splitting request
      Subdivide the requested hierarchy node
      Pack up a splitting reply packet to the requestor
    Else if the request is a geometry request
      Pack up a geometry reply packet to the requestor
    End if
  End for
  End for
  Route all reply packets back to the requesting PEs
  { Phase 4: Split links using information from remote owners. }
  For each reply packet just received
    Locate the link to which this packet belongs and record it
  End for
  For each link in the list of links to be split
    If the link is a self-link
      Form three new link using data received from other PEs
    Else if the link should be split on the left
      Form two new links using data received from other PEs
    Else
      Form two new links using data received from other PEs
    End if
    Push the new links onto the local link heap
  End for
End while

```

Algorithm 20: Revised parallel link refinement

```

{ Update the link error estimate for all links. }
{ Phase 1: Synthesize exchange packets to p and q. }
For each link L in local link heap
    Synthesize a brightness request packet to owner of left end
    Synthesize a brightness request packet to owner of right end
End for
Route brightness request packets to their owners
{ Phase 2: Service brightness requests. }
For each brightness packet just received
    Locate the requested hierarchy node
    Synthesize a brightness reply packet to the requestor
End for
Route brightness reply packets to the requestors
{ Phase 3: Update link error estimates. }
For each link L in local link heap
    Update error estimate for L using brightness data just received
End for
{ Reheapify on the now-valid link error estimates }
Perform a standard reheapify operation on the local link heap

```

Algorithm 22: Revised parallel reheapify

Reheapifying is fairly straightforward, but again requires more communication than before. Brightness request packets must be sent off to the PEs owning the hierarchy nodes at both ends of all links in the local link heap. These owners must then send the brightness data back to the requesting processor. Then, the error estimate may be updated for each link using the brightness data received from other PEs. Algorithm 22 is the pseudocode for this revised version of parallel reheapify.

With the link heap delocalized, any convenient method may be used to distribute the links across the PEs. There are two important factors to consider when distributing the links: local link heap size and link refinement load. The link heap sizes can be equalized very easily by calculating the average local heap size, and redistributing excess links from PEs having a greater than average number. Link refinement load is a bit more problematic. We can, however, note that links which are closer to the top of a heap will probably be split sooner than link further down in the heap due to their larger estimated link error. Thus, if the top several elements of all link heaps are periodically shuffled randomly around the PEs, then the link subdivision load should be equalized as well. The next section presents empirical evidence to support this claim.

5.7.2 Revised algorithm

Shown below in Figure 28 is the output from a typical run on 64 nCUBE processors using the revised link heap decomposition strategy and Algorithm 20, Algorithm 21, and Algorithm 22. Note the logarithmic decrease in the maximum link error, and the exponential increase in the number of links as the algorithm progresses. The theoretical maximum performance for 64 nCUBE processors, in terms of MFLOPS (millions of floating-point operations per second) is about 200 MFLOPS. We see the majority of the time spent solving the problem lay in the tasks

Reader: Read 75 polygons from 'geom' file.
 Distribution starts at level 12. Chunk size is 0.

Red / Grn / Blu Residual	Link Resid	Links	Iterations
1.00e+10 / 1.00e+10 / 1.00e+10	3.8820e+03	1002	RGB
1.90e+02 / 2.30e+02 / 2.50e+02	3.8820e+03	1002	
1.90e+02 / 2.30e+02 / 2.50e+02	2.4560e+02	1139	B
1.90e+02 / 2.30e+02 / 2.00e+02	1.5560e+03	1139	
1.90e+02 / 2.30e+02 / 2.00e+02	2.2360e+02	1198	GG
1.90e+02 / 7.20e+01 / 2.00e+02	9.1350e+02	1198	
1.90e+02 / 7.20e+01 / 2.00e+02	1.9730e+02	1291	B
1.90e+02 / 7.20e+01 / 8.70e+01	3.1990e+02	1291	
1.90e+02 / 7.20e+01 / 8.70e+01	1.9110e+02	1348	RR
7.30e+01 / 7.20e+01 / 8.70e+01	2.9150e+02	1348	
7.30e+01 / 7.20e+01 / 8.70e+01	8.6420e+01	1828	BB
7.30e+01 / 7.20e+01 / 2.50e+01	2.1810e+02	1828	
7.30e+01 / 7.20e+01 / 2.50e+01	7.3040e+01	2078	RR
2.90e+01 / 7.20e+01 / 2.50e+01	1.3200e+02	2078	
2.90e+01 / 7.20e+01 / 2.50e+01	7.2410e+01	2167	GG
2.90e+01 / 2.20e+01 / 2.50e+01	7.9520e+01	2167	
...			
1.20e-01 / 1.40e-01 / 1.30e-01	1.4390e-01	551976	GGG
1.20e-01 / 1.00e-01 / 1.30e-01	2.7070e-01	551976	
1.20e-01 / 1.00e-01 / 1.30e-01	1.2510e-01	630864	BBB
1.20e-01 / 1.00e-01 / 8.80e-02	1.9330e-01	630864	
1.20e-01 / 1.00e-01 / 8.80e-02	1.2190e-01	646256	RRR
7.80e-02 / 1.00e-01 / 8.80e-02	1.2260e-01	646256	
7.80e-02 / 1.00e-01 / 8.80e-02	1.0060e-01	778404	GGG
7.80e-02 / 4.10e-02 / 8.80e-02	1.5770e-01	778404	
7.80e-02 / 4.10e-02 / 8.80e-02	8.7630e-02	890649	BBB
7.80e-02 / 4.10e-02 / 2.90e-02	1.5480e-01	890649	
7.80e-02 / 4.10e-02 / 2.90e-02	7.8170e-02	994319	RRR
4.50e-02 / 4.10e-02 / 2.90e-02	1.1360e-01	994319	
4.50e-02 / 4.10e-02 / 2.90e-02	1.0460e-01	1002190	RGGGB

Total form factor = 1.04743
 Links in hierarchy = 1002190
 Patches in hierarchy = 35715
 Average links per node = 14.0306
 Totally visible links = 788985
 Partly visible links = 213205
 Occluded links = 26986
 Approximate memory usage: 56671052

1000000 interactions:

Task	Seconds	Operations	MFLOPS	% of Time
Reader	5.41	3586	0.000663	0.5 %
SetUp	418.69	9024547348	21.554335	38.8 %
Solver	572.29	250102686	0.437019	53.0 %
Storer	83.21	7512750	0.090282	7.7 %
TOTALS	1079.60	9282166370	8.597745	100.0 %

Figure 28: Typical output from a 64 PE run

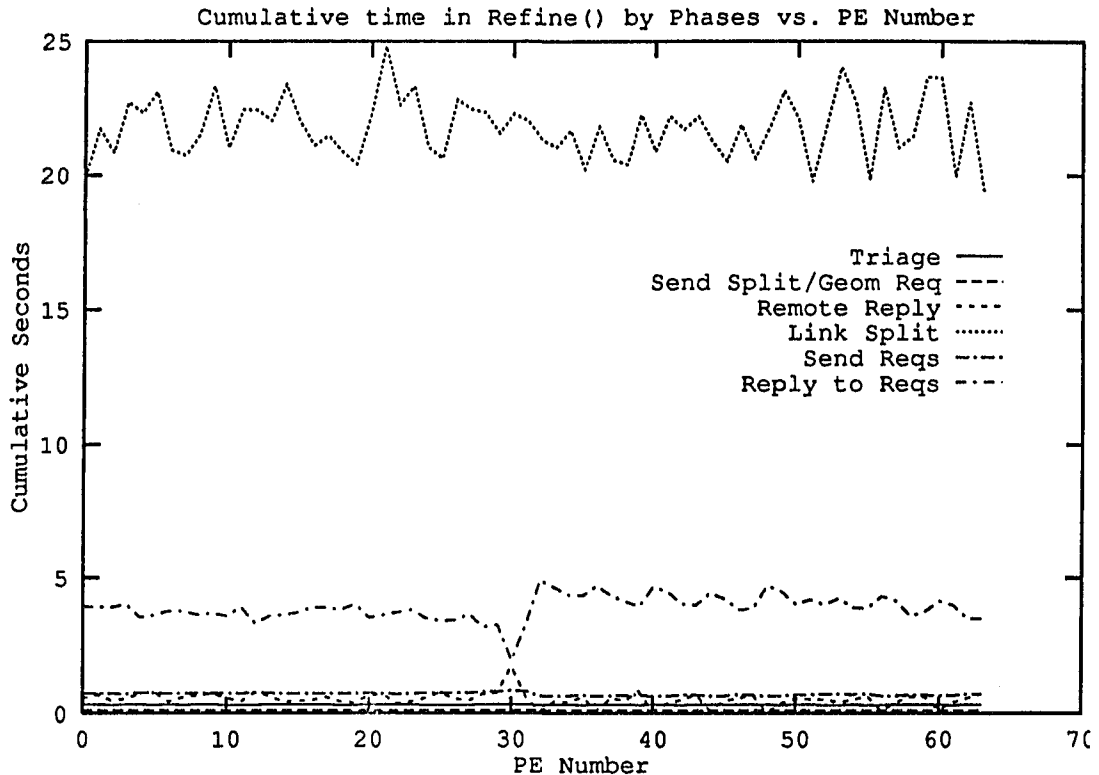


Figure 29: Time spent in link refinement phases vs. processor

called "SetUp," and "Solver." SetUp corresponds to the link refinement process, and supporting operations, and Solver corresponds to the iterative radiosity solution. Further accounting of the Solver task shows that the vast majority of time spent in Solver is consumed by the link contribution part of parallel hierarchical matrix-vector multiply shown in Algorithm 18. The apparently poor showing in Figure 28 in terms of MFLOPS serves to emphasize the communications-intensive nature of the algorithm and the need for further investigation into the algorithm's behavior in a practical setting.

The parallel code has been highly instrumented to collect link distribution statistics, node hierarchy statistics, and timings for various sections of the code. Such data has been extremely useful in locating sources of inefficiency, and in developing the aforementioned hierarchy decomposition strategies. In the following pages, graphs are presented which have been constructed from this performance information.

The first performance graph, shown in Figure 29, shows the breakdown of time spent in the parallel link refinement of Algorithm 14. By far the dominant phase is the one labeled "Link Split." This phase is analogous to phases 4 and 7 in Algorithm 14. The next most dominant is the

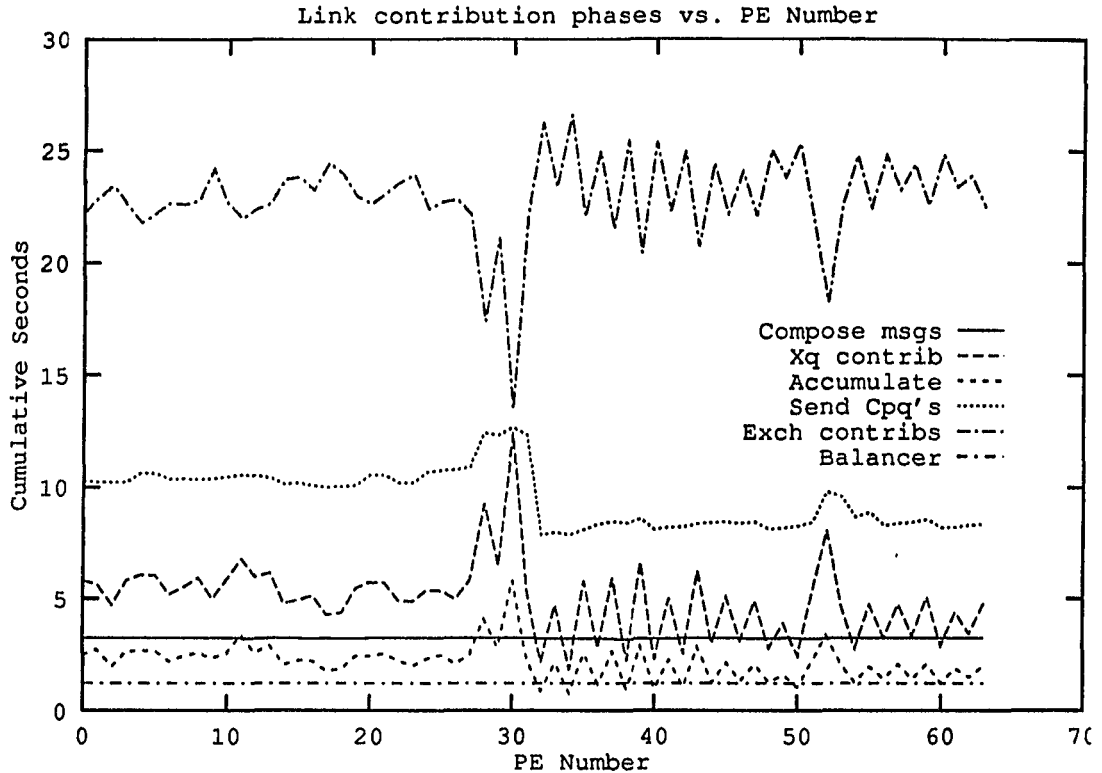


Figure 30: Time spent in link contribution phases vs. processor

phase labeled "Reply to Requests." This phase is analogous to steps 31 and 53 in Algorithm 14. All other phases of link refinement consume less than two seconds each. In all cases, the load balance is excellent.

Figure 30 shows a graph of the time taken by the various link contribution phases, as well as time spent in communication, and time spent in the link load balancer. The first three symbols in the legend correspond to the three phases of Algorithm 21. The next two symbols correspond to the two message routing steps in the same algorithm. The final symbol corresponds to the total amount of time spent in the link heap balancing algorithm. One may notice that the vast majority of time consumed by the link contribution algorithm lay in the two communication phases. Load balance appears to be good with a few minor exceptions.

The next graph, shown in Figure 31, is a profile of link connectivity for both the left and right ends of all links. The left-link-end ownership is shown as "Links from," and right-link-end ownership is shown as "Links to." The graph was constructed by histogramming the owning PE number of the left end of all links on all PEs. In other words, a point on the graph shows the number of links for which the hierarchy node at the left (or right) end is owned by a certain PE. This graph

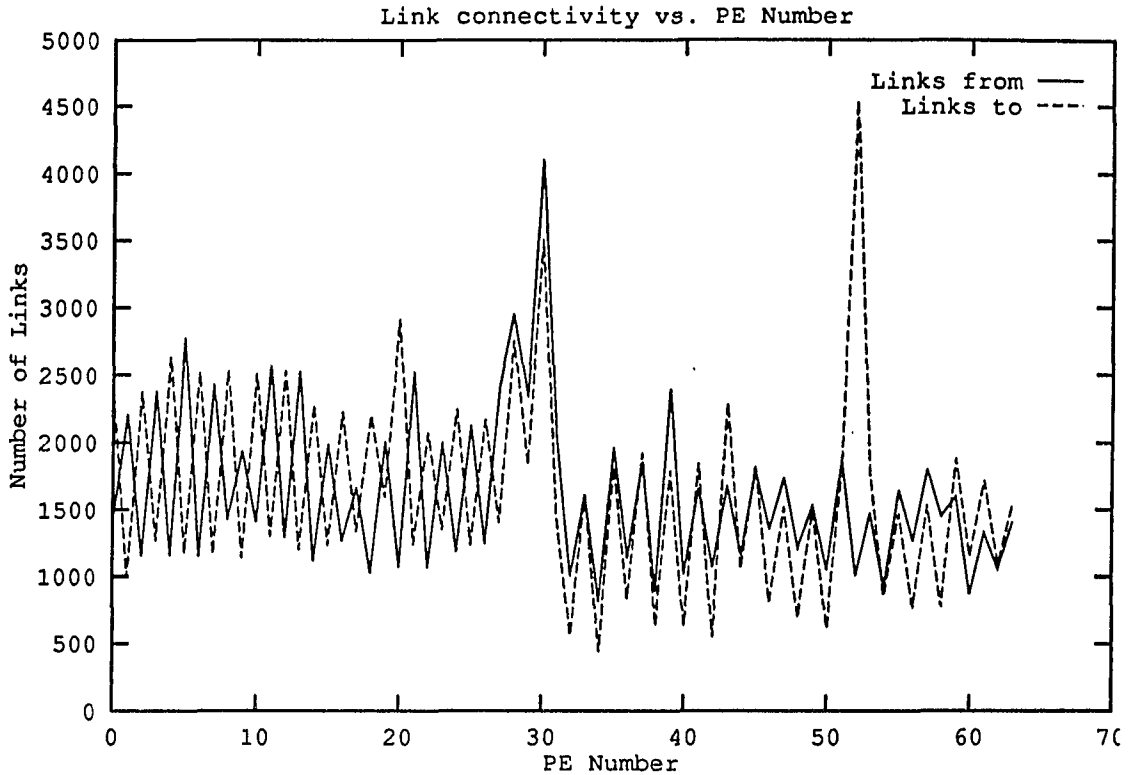


Figure 31: Histogram of link connectivity vs. processor number

does not give any information on how well the local link heaps are load balanced. It indicates how well the hierarchy distribution scheme has spread out node ownership among the PEs. Note that the total number of links connecting to a PE is roughly constant except for PE 30. PE 30 owns a bright light source which the hierarchy subdivision strategy has not distributed across the PEs. Either a larger *dlevel* or a larger light source would more evenly distribute the links to this light source.

Communication volume for one link refinement step is shown in Figure 32. Again, the communication is well-balanced with the exception of the load on PE 30, which has an inordinate number of receives. This indicates that the link connectivity during the time of the refinement step was more highly connected to a hierarchy node or nodes owned by PE 30. Methods for reducing overall communication burden are presented in the next chapter.

The final performance graph plots the overall performance of the algorithm against the number of processors used. The absolute performance is comparable over a range of problem sizes. However, the performance scales somewhat less than linearly for the range of machine sizes shown for a fixed-size problem. There are two main reasons for this. As the number of PEs

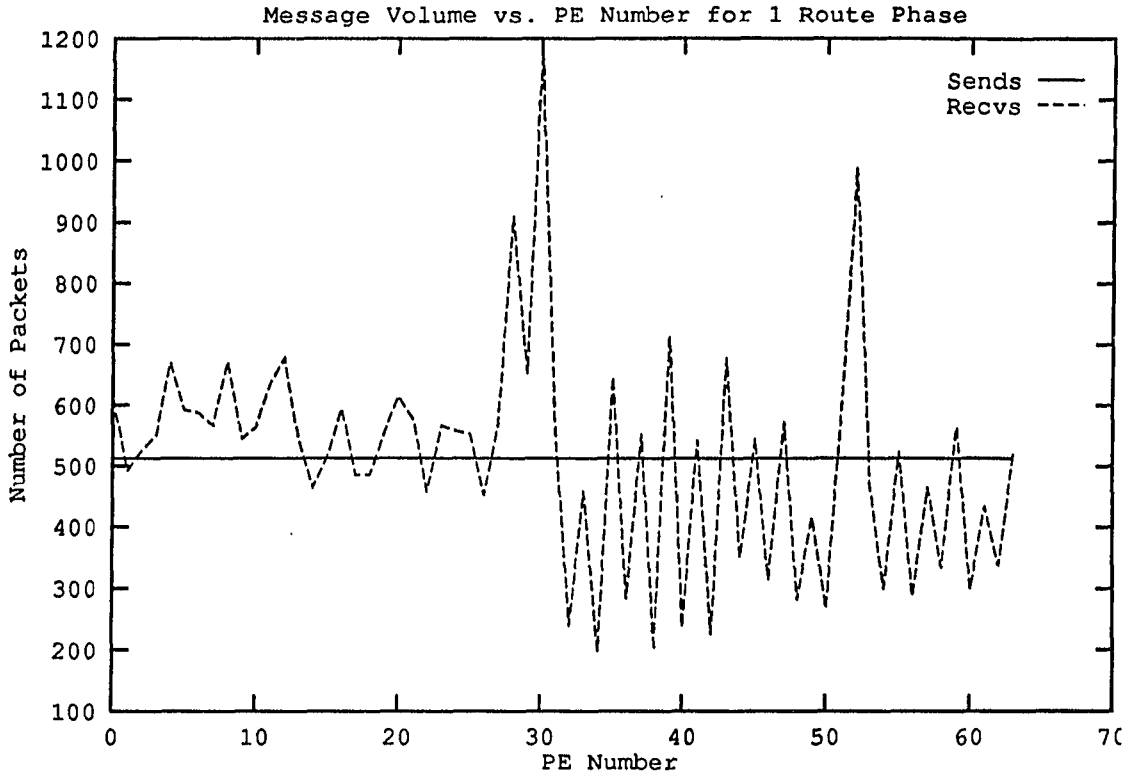


Figure 32: Time spent in a single refine step vs. processor number

increases for a fixed size problem, the total amount of data that must be transmitted between PEs increases due to an increased chance that the hierarchy nodes referenced by a PE's local links are nonlocal. The second reason has to do with the nature of the communication pattern itself. Suppose each PE wishes to send N packets to other randomly selected PEs. The routing operation will take place in *lnproc* stages with each stage moving an average of $0.5 \times N$ packets between each pair of PEs. Thus, the total time for a routing operation involving N packets will be

$$T_{route} = (\alpha + \beta N) \log(p) \quad (67)$$

where α is the constant setup time for a message,
 β is the time it takes to transmit one packet, and
 p is the number of PEs used.

With a fixed-size problem, N gets smaller as the links get spread out over more PEs. Thus, we may model the overall time take by the hierarchical radiosity algorithm as

$$T(p, N) = \gamma + \frac{\delta N}{p} + \left(\alpha + \frac{\beta N}{p}\right) \log(p) \quad (68)$$

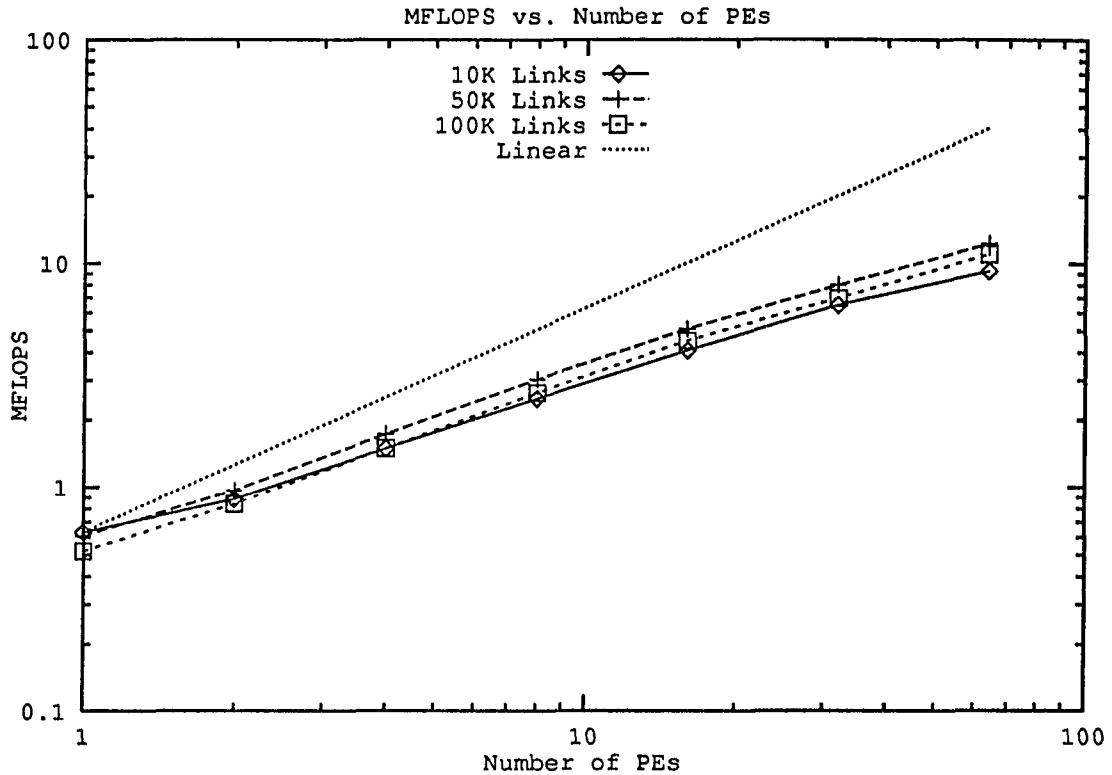


Figure 33: Performance vs. number of PEs

where γ is constant overhead time for the whole algorithm,
 δ is the time to process one link to completion, and
 N is the number of links created.

The first term in (68) is constant time taken to load the program onto the PE array, read in the problem geometry description, and create the initial hierarchy. Note that all these operations are independent of the number of processors used, and the number of links to be created. The second term accounts for all of the $O(N)$ work associated with setting up and creating link, traversing the hierarchy, and solving the system. The third and final term accounts for the time spent routing packets among the PEs during link subdivision, reheapify operations, and matrix-vector multiply.

CHAPTER VI

SUMMARY AND FURTHER RESEARCH

6.1 Summary

In Chapter I, the fields of nonrealistic and realistic image synthesis are introduced, and two key approaches to realistic image synthesis are also presented. Several parallel architectures with applications to realistic image synthesis, and one parallel ray tracing code are also described. Chapter II details the realistic image synthesis method of radiosity, and presents a way of formulating the radiosity equation as a symmetric system. Various solution methods are analyzed both theoretically and experimentally for their suitability to the radiosity application. In Chapter III, the concept of a hierarchical method is introduced. The origins of the hierarchical methods are traced through the astrophysics literature, and through their introduction into the computer graphics community. Chapter IV presents several enhancements to the two existing hierarchical radiosity methods and explains their significance and benefits. Finally, Chapter V details the construction of a parallel code to implement the enhanced hierarchical radiosity method on an nCUBE 2 parallel supercomputer. Performance is analyzed, shortcomings discovered, and methods to deal with them either proposed or implemented.

The renderer presented in Chapter V is the first and only parallel implementation of the hierarchical radiosity method to date to the knowledge of this author. As with many initial ventures into making a new class of algorithm parallel, the absolute performance realized is not impressive. However, it deals with the key issues involved in making the algorithm parallel, and paves the way for future analysis and improvements.

6.2 Further research

6.2.1 Optimizations to existing code

Clearly, there is still much room for improvement in the performance of the parallel hierarchical radiosity implementation given in Chapter V. As it is an extremely communication-intensive algorithm, the most gains will be had by optimizing the communication patterns, especially in the linear equation solver. One can observe that there is significant reuse of x_p values during the link refinement, link contribution, and reheapify operations. The potential for a drastic reduction in communication volume exists by exploiting this data reuse. At present, a message is being generated for every reference to a particular x_p when, in fact, fewer messages would suffice.

6.2.2 Other areas of investigation

Many other avenues of research lay open to further scrutiny with the introduction of hierarchical methods. Hierarchical methods have been applied to a rather narrow class of physical problems to date (gravitational N -body, and diffuse radiosity transport). Many other physical problems

exist for which hierarchical solution methods might be beneficial such as finite element methods, weather modeling, molecular modeling, and radar cross-section estimation. All of these problems have feature sets which can be approximated to varying degrees of accuracy on multiple resolution levels.

Furthermore, the specific interpretation of the hierarchical method as applied to the radiosity problem is still not completely defined. Several specific issues are discussed below.

6.2.2.1 Exact coupling factors

All existing hierarchical radiosity methods approximate the coupling between patches to some level or another. The quadruple integral for computing coupling factors, (14), is difficult to solve in closed form for general surfaces. However, if the surfaces are sufficiently restricted in their generality, a tractable integration problem might be found, perhaps with the aid of Stokes' theorem [Sparrow 78]. If an expression for the exact coupling were found, then there would be no coupling factor estimate error, and some other quantity would have to be found to drive the link refinement process.

6.2.2.2 Discretization error

Discretization error is mentioned *en passant* in the section titled "Alternation of error types" on page 46. This form of error has not been rigorously quantified by any radiosity methods to date, and thus, is not well accounted for while balancing link error against solution error. If a tractable expression for the exact coupling between two arbitrary patches is ever found, an understanding of discretization error will become mandatory. No longer will coupling estimate error be able to drive link refinement.

Discretization error is a measure of how well a continuously varying function (the continuous radiosity solution) is approximated by a piecewise constant function (patch brightnesses). Thus, it is related to patch geometry, the brightness gradient across a patch, and the error (if any) present in couplings to the patch. The brightness gradient across a patch is, in turn, related to the coupling gradient across a patch. Even in the presence of exact coupling factors, this gradient will not be known. Regardless of what factors influence discretization error, it is desirable to formulate it in terms of power so that it may be compared against solution error for purposes of driving the link refinement process.

One immediate application for discretization error measure is in dealing with the tartan artifact. Although rowsum correction minimizes the tartan artifact, a more elegant solution is desirable. The tartan artifact is not caused by errors in coupling factor estimates. Even if exact patch couplings are known, experiment has shown that the tartan artifact remains. The artifact is much more heavily influenced by the choice of patch subdivision near corners than it is by coupling estimate errors.

An immediate consequence of knowing more about discretization error will come in the form of more intelligent choices for patch subdivision. Hierarchical radiosity renderers now either subdivide a patch equally into two or four subpatches. If more is known about how subdivision will

affect the brightness solution, subdivision can be modified such that *discretization* error is reduced with patch subdivision, not just *coupling factor* error. This may mean splitting a patch into unequal areas, or along a different subdivision line, or both. Another important consequence affects rendering curved surfaces. Presently, curved surfaces must be tessellated *a priori*, and dealt with as a fixed set of flat polygons. A more efficient approach would be to represent a curved surface as a single object, and tessellate it adaptively *during the solution process* based on the current viewpoint and lighting conditions. Such an approach will create far fewer tessellation polygons for a given level of discretization error and solution error than a flat *a priori* tessellation of the same curved surface under the same ambient conditions.

6.2.2.3 Specularity

Perhaps the most interesting avenue of future research with the hierarchical radiosity method lies in modeling specular effects. In order to obtain the data necessary to evaluate a specular shading model on a surface, interactions would have to be between *three* patches, not just two. This corresponds well with the notion of the three-point transport geometry shown in Figure 1 on page 4. A hierarchical scheme using these three-ended links, or *bonks*, would concentrate its effort in areas of high specularity and high brightness while expending much less effort in areas of low specularity or low brightness. This is similar to the existing diffuse method which concentrates its effort in areas of high brightness while expending much less effort in areas of low brightness.

BIBLIOGRAPHY

- [Appel 85] Appel, Andrew W., "An Efficient Program for Many-Body Simulation," *SIAM Journal of Scientific and Statistical Computing*, Vol. 6, No. 1, January 1985, pp. 85-103.
- [Arnoldi 87] Arnaldi, Bruno, Thierry Priol, and Kadi Bouatouch, "A New Space Subdivision Method for Ray Tracing CSG Modelled Scenes," *The Visual Computer*, Vol. 3, No. 2, August 1987, pp. 98-108.
- [Arvo 87] Arvo, James, and David Kirk, "Fast Ray Tracing by Ray Classification," *Computer Graphics*, Vol. 21, No. 4, 1987, pp. 55-64.
- [Badouel 90] Badouel, Didier, Kadi Bouatouch, and Thierry Priol, "Ray Tracing on Distributed Memory Parallel Computers: Strategies for Distributing Computations and Data," *Technical Report 508, Institut de Recherche en Informatique et Systemes Aleatoires (IRISA)*, January 1990.
- [Barnes 86] Barnes, Josh, and Piet Hut, "A Hierarchical $O(N \log N)$ Force-Calculation Algorithm," *Nature*, Vol. 324, December 1986, pp. 446-449.
- [Barr 84] Barr, Alan, "Global and Local Deformations of Solid Primitives," *Computer Graphics*, Vol. 18, No. 3, 1984, pp. 21-30.
- [Barr 86] Barr, Alan, "Ray Tracing Deformed Surfaces," *Computer Graphics*, Vol. 20, No. 4, 1986, pp. 287-296.
- [Bjørstad 91a] Bjørstad, Petter E., and Erik Boman, "SLALOM: A Better Algorithm," *Supercomputing Review*, November 1991, pp. 57-62.
- [Bjørstad 91b] Bjørstad, Petter E., and Erik Boman, "A New Algorithm for the SLALOM Benchmark," *Technical Report No. 55, Department of Informatics, University of Bergen, Norway*, May 1991.
- [Blinn 82] Blinn, Jim, "A Generalization of Algebraic Surface Drawing," *ACM Transactions on Graphics*, Vol. 1, No. 3, July 1982, pp. 235-256.
- [Brønsvoort 85] Brønsvoort, Willem F., and Föpke Klok, "Ray Tracing Generalized Cylinders," *ACM Transactions on Graphics*, Vol. 4, No. 4, October 1985, pp. 291-303.
- [Burger 89] Burger, P., and D. Gillies, "Rapid Ray-tracing of General Surfaces of Revolution," *New Advances in Computer Graphics - Proceedings of Computer Graphics International '89*, R. A. Earnshaw and B. Wyvill ed., Springer-Verlag, New York, 1989, pp. 523-532.
- [Carter 89] Carter, Michael B., "Ray Tracing Complex Scenes on a MIMD Concurrent Computer," Master's Thesis, Oklahoma State University, Stillwater Oklahoma, 1989.

- [Carter 90] Carter, Michael B., and Keith A. Teague, "The Hypercube Ray Tracer," *Proceedings of the Fifth Annual Conference on Distributed Memory Concurrent Computers*, Spring, 1990.
- [Carter 93a] Carter, Michael B., and John L. Gustafson, "The Symmetric Radiosity Formulation," *Ames Laboratory Technical Report IS-J 4880*, Ames, Iowa.
- [Carter 93b] Carter, Michael B., and John L. Gustafson, "An Improved Hierarchical Radiosity Method," *Ames Laboratory Technical Report IS-J 4881*.
- [Chen 89] Chen, Shenchang Eric, "A Progressive Radiosity Method and its Implementation in a Distributed Processing Environment," Master's Thesis, Cornell University, January 1989.
- [Chen 90] Chen, Hong, En-Hua Wu, "An Adapted Solution of Progressive Radiosity and Ray-Tracing Methods for Non-diffuse Environments," T. S. Chua ed., Tosiyasu L. Kunii ed., *CG International '90: Computer Graphics Around the World*, Springer-Verlag, Tokyo, 1990, pp. 477-490.
- [Cohen 85] Cohen, Michael F., and Donald P. Greenberg, "The hemi-cube: A radiosity approach for complex environments.," *Computer Graphics*, Vol. 19, No. 3, July 1985, pp. 31-40.
- [Cohen 86] Cohen, Michael F., Donald P. Greenberg, David S. Immel, and Philip J. Brock, "An Efficient Radiosity Approach for Realistic Image Synthesis," *IEEE Computer Graphics and Applications*, Vol. 6, No. 2, pp. 26-30.
- [Cohen 88] Cohen, Michael F., Shenchang Eric Chen, John R. Wallace, and Donald P. Greenberg, "A Progressive Refinement Approach to Fast Radiosity Image Generation," *Computer Graphics*, Vol. 22, No. 4, Aug. 1988, pp. 75-84.
- [Cook 82] Cook, Robert L., and Kenneth E. Torrance, "A Reflectance Model for Computer Graphics," *ACM Transactions on Graphics*, Vol. 1, No. 1, January 1982, pp. 7-24.
- [Cook 84] Cook, Robert, Thomas Porter, and Loren Carpenter, "Distributed Ray Tracing," *Computer Graphics*, Vol. 18, No. 3, 1984, pp. 137-145.
- [Cook 86] Cook, Robert, "Stochastic Sampling in Computer Graphics," *ACM Transactions on Graphics*, Vol. 5, No. 1, January 1986, pp. 51-72.
- [Coquillart 85] Coquillart, Sabine, "An Improvement of the Ray-Tracing Algorithm," *Proceedings of Eurographics '85*, C. E. Vandoni, ed., Elsevier / North-Holland, Amsterdam, 1985, pp. 77-88.
- [Cordonnier 85] Cordonnier, E., C. Bouville, I. Marchal, and J. L. Dubois, "Creating CSG Modelled Pictures for Ray-Casting Display," *Proceedings of Eurographics '85*, C. E. Vandoni, ed., Elsevier / North-Holland, Amsterdam, 1985, pp. 171-184.
- [Corman 90] Corman, Thomas H., Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 1990.
- [Cottingham 89] Cottingham, M. S., "Efficiently Ray Tracing CSG Trees," *Proceedings of Ausgraph '89*, Australasian Computer Graphics Association, 1989, pp. 269-274.

- [Deguchi 86] Deguchi, Hiroshi, *et al.*, "A Tree-Structured Parallel Processing System for Image Generation by Ray Tracing," *Systems and Computers in Japan*, Vol. 17, No. 12, 1986.
- [Devillers 89] Devillers, Olivier, "The Macro-Regions: an Efficient Space Subdivision Structure for Ray Tracing," *Proceedings of Eurographics '89*, W. Hansmann, F. R. A. Hopgood and W. Strasser ed., Elsevier / North-Holland, Amsterdam, 1989, pp. 27-38.
- [Drucker 92] Drucker, Steven M., and Peter Schroeder, "Fast Radiosity Using a Data Parallel Architecture," *Third Eurographics Workshop on Rendering*, Bristol, UK, May 1992, pp. 247-258.
- [Edwards 82] Edwards, Bruce, "Implementation of a Ray-Tracing Algorithm for Rendering Superquadric Solids," Master's Thesis, Rensselaer Polytechnic Institute, Troy, New York, December 1982.
- [Filip 89] Filip, Daniel J., "Blending Parametric Surfaces," *ACM Transactions on Graphics*, Vol. 8, No. 3, July 1989, pp. 164-173.
- [Fontes 84] Fontes, Steve, "Ray Tracing Surfaces of Revolution," Master's Thesis, Worcester Polytechnic Institute, 1984.
- [Fuchs 77] Fuchs, H., *et al.*, "Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes," *ACM Computer Graphics*, Vol. 19, No. 3, July 1985, pp. 111-120.
- [Fuchs 89] Fuchs, Henry, *et al.*, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *Computer Graphics*, Vol. 23, No. 3, July 1989.
- [Fujimoto 86] Fujimoto, Akira, Tkayuki Tanaka, and Kansei Iwata, "ARTS: Accelerated Ray-Tracing System," *IEEE Computer Graphics and Applications*, Vol. 6, No. 4, April 1986, pp. 16-26.
- [Fussell 88] Fussell, Donald, and K. R. Subramanian, "Fast Ray Tracing Using K-D Trees," *Technical Report No. TR-88-07*, Department of Computer Sciences, University of Texas at Austin, March 1988.
- [Gervautz 86] Gervautz, Michael, "Three Improvements of the Ray Tracing Algorithm for CSG Trees," *Computers and Graphics*, Vol. 10, No. 4, 1986, pp. 333-339.
- [Getto 89] Getto, P., "Fast Ray Tracing of Unevaluated Constructive Solid Geometry Models," *New Advances in Computer Graphics - Proceedings of Computer Graphics International '89*, R. A. Earnshaw and B. Wyvill, ed., Springer-Verlag, New York, 1989, pp. 563-578.
- [Giger 89] Giger, Christine, "Ray Tracing Polynomial Tensor Product Surfaces," *Proceedings of Eurographics '89*, W. Hansmann, F. R. A. Hopgood and W. Strasser, eds., Elsevier / North-Holland, Amsterdam, 1989, pp. 125-136.
- [Glassner 84] Glassner, Andrew, "Space Subdivision for Fast Ray Tracing," *IEEE Computer Graphics and Applications*, Vol. 4, No. 10, October 1984, pp. 15-22.

- [Goldsmith 87] Goldsmith, Jeffrey, and John Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing," *IEEE Computer Graphics and Applications*, Vol. 7, No. 5, May 1987, pp. 14-20.
- [Golub 89] Golub, Gene H., and Charles F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1989.
- [Greengard 87] Greengard, L., and V. Rokhlin, "A Fast Algorithm for Particle Simulations," *Journal of Computational Physics*, Vol. 73, 1987, pp. 325-349.
- [Greengard 88] Greengard, *The Rapid Evaluation of Potential Fields in Particle Systems*, MIT Press, Cambridge, MA, 1988.
- [Guitton 91] Guitton, P., J. Roman, and Christophe Schlick, "Two Parallel Approaches for a Progressive Radiosity," *Second Eurographics Workshop on Rendering*, Barcelona, Spain, May 1991.
- [Gustafson 91] Gustafson, John L., Diane Rover, Stephen Elbert, and Michael Carter, "The Design of a Scalable, Fixed-Time Computer Benchmark," *Journal of Parallel and Distributed Computing*, Vol. 12, 1991, pp. 388-401.
- [Hanrahan 83] Hanrahan, Pat, "Ray Tracing Algebraic Surfaces," *Computer Graphics*, Vol. 17, No. 3, 1983, pp. 32-90.
- [Hanrahan 91] Hanrahan, Pat, David Salzman, and Larry Aupperle, "A Rapid Hierarchical Radiosity Algorithm," *Computer Graphics*, Vol. 25, No. 4, July 1991, pp. 197-206.
- [Hart 89] Hart, John, Daniel Sandin, and Louis Kauffman, "Ray Tracing Deterministic 3-D Fractals," *Computer Graphics*, Vol. 23, No. 3, 1989, pp. 298-296.
- [He 91] He, Xiao D., Kenneth E. Torrance, François X. Sillion, and Donald P. Greenberg, "A Comprehensive Physical Model for Light Reflection," *Computer Graphics*, Vol. 25, No. 4, July 1991, pp. 174-186.
- [Hermitage 90] Hermitage, Shirley A., Terrance L. Huntsberger, and Beverly A. Huntsberger, "Hypercube Algorithm for Radiosity in a Ray Tracing Environment," *Proceedings of the 5th Distributed Memory Computing Conference*, David W. Walker and Quentin F. Stout, eds., IEEE Computer Society Press, Washington, April 1990, pp. 206-211.
- [Immel 86] Immel, David S., Michael F. Cohen, and Donald P. Greenberg, "A Radiosity Method for Non-Diffuse Environments," *Computer Graphics*, Vol. 20, No. 4, August 1986, pp. 133-142.
- [Jansen 86] Jansen, F. W., "Data Structures for Ray Tracing," *Data Structures for Raster Graphics*, L. R. A. Kessener, F. J. Peters, and M. L. P. van Lierop, eds., Springer-Verlag, New York, 1986, pp. 57-73.
- [Jessel 91] Jessel, J. P., M. and Paulin, R. Caubet, "An Extended Radiosity Using Parallel Ray-Traced Specular Transfers," *Second Eurographics Workshop on Rendering*, Barcelona, Spain, May 1991.

- [Joy 86] Joy, Kenneth, and Murthy Bhetanabhotla, "Ray Tracing Parametric Surface Patches Utilizing Numerical Techniques and Ray Coherence," *Computer Graphics*, Vol. 20, No. 4, 1986, pp. 279-285.
- [Kajiya 82] Kajiya, James, "Ray Tracing Parametric Patches," *Computer Graphics*, Vol. 16, No. 3, 1982, pp. 245-254.
- [Kajiya 83a] Kajiya, James, "New Techniques for Ray Tracing Procedurally Defined Objects," *Computer Graphics*, Vol. 17, No. 3, 1983, pp. 91-102.
- [Kajiya 83b] Kajiya, James, "New Techniques for Ray Tracing Procedurally Defined Objects," *ACM Transactions on Graphics*, Vol. 2, No. 3, 1983, pp. 161-181.
- [Kajiya 84] Kajiya, James, and Brian von Herzen, "Ray Tracing Volume Densities," *Computer Graphics*, Vol 18, No. 3, 1984, pp. 165-174.
- [Kajiya 85] Kajiya, James, "Anisotropic Reflection Models," *Computer Graphics*, Vol. 19, No. 3, 1985, pp. 15-21.
- [Kalra 89] Kalra, Devendra, and Alan Barr, "Guaranteed Ray Intersections with Implicit Surfaces," *Computer Graphics*, Vol. 23, No. 3, 1989, pp. 297-306.
- [Kay 86] Kay, Timothy, and James Kajiya, "Ray Tracing Complex Scenes," *Computer Graphics*, Vol. 20, No. 4, 1986, pp. 269-278.
- [Kunii 85] Kunii, Toshiyasu, and Geoff Wyvill, "CSG and Ray Tracing Using Functional Primitives," *Computer-Generated Images: The State of the Art*, N. Magnenat-Thalmann and D. Thalmann, ed., Springer-Verlag, New York, 1985, pp. 137-152.
- [Lee 85] Lee, Mark, Richard Redner, and Samuel Uselton, "Statistically Optimized Sampling for Distributed Ray Tracing," *Computer Graphics*, Vol. 19, No. 3, 1985, pp. 61-67.
- [Lischinski 90] Lischinski, Daniel, and Jakob Gonczarowski, "Improved Techniques for Ray Tracing Parametric Surfaces," *The Visual Computer*, Vol. 6, No. 3, June 1990, pp. 134-152.
- [MacDonald 88] MacDonald, David, "Space Subdivision Algorithms for Ray Tracing," Masters Thesis, Department of Computer Science, University of Waterloo, Spring 1988.
- [Montani 90] Montani, C., and R. Scopigno, "Ray Tracing CSG Trees Using the STICKS Representation Scheme," *Computers and Graphics*, Vol. 14, No. 3/4, 1990, pp. 481-490.
- [du Montcel 85] du Montcel, Bruno Tezenas, and Alain Nicolas, "An Illumination Model for Ray Tracing," *Proceedings of Eurographics '85*, C. E. Vandoni ed., Elsevier / North-Holland, Amsterdam, 1985, pp. 63-75.
- [Naylor 86] Naylor, Bruce, and William Thibault, "Application of BSP Trees to Ray-Tracing and CSG Evaluation," *Technical Report GIT-ICS 86/03*, School of Information and Computer Science, Georgia Institute of Technology, Feb. 1986.

- [Nishita 85] Nishita, Tomoyuki, and Eihachiro Nakamae, "Continuous Tone Representation of Three-Dimensional Objects Taking Account of Shadows and Interreflection," *Computer Graphics*, Vol. 19, No. 3, July 1985, pp. 23-30.
- [Nishita 90] Nishita, Tomoyuki, Thomas W. Sederberg, and Masanori Kakimoto, "Ray Tracing Trimmed Rational Surface Patches," *Computer Graphics*, Vol. 24, No. 3, 1990, pp. 337-345.
- [Parke 80] Parke, F. I., "Simulation and Expected Performance Analysis of Multiple Processor Z-Buffer Systems," *Computer Graphics*, Vol. 21, No. 3, July 1980, pp. 48-56.
- [Potmesil 89] Potmesil, Michael, and Eric M. Hoffert, "The Pixel Machine: A Parallel Image Computer," *Computer Graphics*, Vol. 23, No. 3, July 1989.
- [Priol 88] Priol, Thierry, and Kadi Bouatouch, "Experimenting With a Parallel Ray Tracing Algorithm on a Hypercube Machine," *Proceedings of Eurographics '88*, Elsevier / North-Holland, Amsterdam, 1988, pp. 248-259.
- [Priol 89] Priol, Thierry, and Kadi Bouatouch, "Static Load Balancing for a Parallel Ray Tracing on a MIMD Hypercube," *The Visual Computer*, Vol. 5, No. 1/2, March 1989, pp. 109-119.
- [Purgathofer 91] Purgathofer, Werner, and Michael Zeiller, "Fast Radiosity by Parallelization," K. Bouatouch ed., C. Bouville ed., *Photorealism in Computer Graphics (Proceeding Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics, 1990)*, 1991, pp. 173-183.
- [Rubin 80] Rubin, Steven, and Turner Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," *Computer Graphics*, Vol. 14, No. 3, 1980, pp. 110-116.
- [Sederberg 84] Sederberg, Thomas, and David Anderson, "Ray Tracing of Steiner Patches," *Computer Graphics*, Vol. 18, No. 3, 1984, pp. 159-164.
- [Sederberg 86] Sederberg, Thomas W., and Scott R. Parry, "Free-Form Deformation of Solid Geometric Models," *Computer Graphics*, Vol. 20, No. 4, August 1986, pp. 151-160.
- [SGI 92] *IRIS Crimson Technical Report*, Silicon Graphics Computer Systems, Mountain View, California, July, 1992.
- [Shirley 90] Shirley, Peter, "A Ray Tracing Method for Illumination Calculation in Diffuse-Specular Scenes," *Proceedings of Graphics Interface '90*, Canadian Information Processing Society, Toronto, Ontario, May 1990, pp. 205-212.
- [Shirley 91] Shirley, Peter, Kelvin Sung, and William Brown, "A Ray Tracing Framework for Global Illumination Systems," *Proceedings of Graphics Interface '91*, Canadian Information Processing Society, Calgary, Alberta, June 1991, pp. 117-128.
- [Siegel 81] Siegel, Robert and John R. Howell, *Thermal Radiation Heat Transfer*, Hemisphere Publishing Corporation, Washington, 1981.
- [Sillion 89] Sillion, François, and Claude Puech, "A General Two-Pass Method Integrating Specular and Diffuse Reflection," *Computer Graphics*, Vol. 23, No. 3, July 1989, pp. 335-344.

- [Sillion 91] Sillion, François X., James R. Arvo, Stephen H. Westin, and Donald P. Greenberg, "A Global Illumination Solution for General Reflectance Distributions," *Computer Graphics*, Vol. 25, No. 4, July 1991, pp. 187-196.
- [Slalom 91] *The SLALOM Benchmark Report*, Scalable Computing Laboratory, Ames Laboratory, Ames, Iowa, USA, November 1991.
- [Smits 92] Smits, Brian E., James R. Arvo, and David H. Salesin, "An Importance-Driven Radiosity Algorithm," *Computer Graphics*, Vol. 26, No. 2, July 1992, pp. 273-282.
- [Sparrow 78] Sparrow, E. W., and R. D. Hess, *Radiation Heat Transfer*, Hemisphere Publishing Corporation, Washington, D. C., 1978.
- [Sweeney 86] Sweeney, Michael, and Richard Bartels, "Ray Tracing Free-Form B-Spline Surfaces," *IEEE Computer Graphics and Applications*, Vol. 6, No. 2, February 1986, pp. 41-49.
- [Thirion 90] Thirion, Jean-Philippe, "Tries: Data Structures Based on Boolean Representation for Ray Tracing," *Proceedings of Eurographics '90*, Elsevier / North-Holland, Amsterdam, 1990.
- [Toth 85] Toth, Daniel, "On Ray Tracing Parametric Surfaces," *Computer Graphics*, Vol. 19, No. 3, 1985, pp. 171-179.
- [van Wijk 84a] van Wijk, Jarke, "Ray Tracing Objects Defined by Sweeping Planar Cubic Splines," *ACM Transactions on Graphics*, Vol. 3, No. 3, July 1984, pp. 223-237.
- [van Wijk 84b] van Wijk, Jarke, "Ray Tracing Objects Defined by Sweeping a Sphere," *Computers and Graphics*, Vol. 9, No. 3, July 1985, pp. 283-290.
- [Wallace 87] Wallace, John R., Michael F. Cohen, and Donald P. Greenberg, "A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods," *Computer Graphics*, Vol. 21, No. 4, July 1987, pp. 311-320.
- [Ward 88] Ward, Gregory J., Francis M. Rubinstein, and Robert D. Clear, "A Ray Tracing Solution for Diffuse Interreflection," *Computer Graphics*, Vol. 22, No. 4, August 1988, pp. 85-92.
- [Ward 92] Ward, Gregory J., "Measuring and Modeling Anisotropic Reflection," *Computer Graphics*, Vol. 26, No. 2, July 1992, pp. 265-272.
- [Westin 92] Westin, Stephen H., James R. Arvo, and Kenneth E. Torrance, "Predicting Reflectance Functions from Complex Surfaces," *Computer Graphics*, Vol. 26, No. 2, July 1992, pp. 255-264.
- [Whitted 80] Whitted, Turner, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, Vol. 23, No. 6, June 1980, pp. 343-349.
- [Wyvill 85] Wyvill, Geoff, and Toshiyasu Kunii, "A Functional Model for Constructive Solid Geometry," *The Visual Computer*, Vol. 1, No. 2, 1985, pp. 3-14.
- [Wyvill 86] Wyvill, Geoff, Toshiyasu Kunii, and Yasuto Shirai, "Space Division for Ray Tracing in CSG," *IEEE Computer Graphics and Applications*, Vol. 6, No. 4, November 1990, pp. 13-32.

[Youssef.86] Youssef, Saul, "A New Algorithm for Object Oriented Ray Tracing," *Computer Vision, Graphics, and Image Processing*, Vol. 34, No. 2, May 1986, pp. 125-137.

APPENDIX

The source code to an enhanced serial radiosity renderer is published in this appendix. It is intended to give the reader a basic understanding of the implementation details involved in a hierarchical radiosity code. Source to the parallel hierarchical radiosity renderer is too lengthy and machine specific to be useful here. The following source is written in ANSI standard C in several source modules. Each source module is delineated in the text by a section heading.

A.1 Header file slal.h

```

/*****/
/* File: slal.h
/* Version: %G% %W%
/* Type definitions for all data types used in the SLALOM93 benchmark
/*****/

#ifndef SLAL_H
#define SLAL_H

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <sys/time.h>

#ifndef TRUE
#define FALSE 0
#define TRUE (!FALSE)
#endif

/**/
/* Forward type declarations
/**/
typedef struct Node Node;
typedef struct Link Link;
typedef struct HierVec HierVec;
typedef struct LinkQueue LinkQueue;
typedef struct LinkHeap LinkHeap;

/**/
/* Typedefs
/**/
typedef float Vector[3];
typedef enum {
    none = 0, links = 1,
    refine = 2, vectors = 4,
    iterate = 8, matrix = 16,
    hierarchy = 32, isect = 64
} debugflag;
typedef enum {
    visible, partial, blocked
} Visible;
typedef enum {
    composite, patch
} NodeType;

/**/
/* Constants
/**/
#define defchunksize 1024
#define deflinkchunksize 16
#define maxhiervec 24
#define maxpoly 1024
#define maxvert 1024
#define eps ((float)1e-5)
#define big ((float)1e+10)
#define PI ((float)3.1415926)

/**/
/* Externals
/**/
extern debugflag debug;
extern int solelemR,
solelemG,

```

```

        solelemB;
extern int  rhoelemR,
           rhoelemG,
           rhoelemB;
extern LinkHeap *heap;

/**/
/* Queue of Links
/**/
struct LinkQueue {
    Link      *p;           /* Pointer to Link storage.      */
    int       head,       /* Index of head element.      */
           tail,        /* Index of tail element.     */
           alloclen;     /* Allocated list length.     */
};

/**/
/* Priority queue of Links
/**/
struct LinkHeap {
    Link      *p;           /* Pointer to Link storage.      */
    int       tail,       /* Index of tail element.      */
           alloclen;     /* Allocated list length.     */
};

/**/
/* Definition of the parent node type.
/**/
struct Node {
    /* Data structure linkage */
    Node      *left, *right; /* Pointer to daughter patches. */
    Node      *parent;      /* Pointer to mother patch.     */

    /* Vector storage */
    float     t;           /* Temporary storage.          */
    float     e[maxhiervec]; /* Storage for HierVecs.       */

    /* Node geometry */
    Vector    normal;     /* Normal of the patch in 3-space. */
    Vector    vertex[4];  /* Vertices of the polygon.     */
    Vector    center;     /* Center of the patch in 3-space. */
    float     area;       /* Magnitude of normal vector.  */

    /* Miscellaneous */
    int       id;         /* Unique node ID number.      */
};

/**/
/* HierVec type. This object is, in fact, merely a front-end to the
/* procedural hierarchical vector operations defined as part of the Node type.
/**/
struct HierVec {
    int       index;      /* Index of vec. elem. in patch hier. */
    Node      *hier;     /* Hierarchy in which this vector is */
                       /* stored.                          */
};

/**/
/* Type of elements which will be used in queue and heap types below
/**/
struct Link {
    Node      *p, *q;
    float     cpq, epq, err;
    Visible   vis;
};

#endif          _SLAL_H_

```

A.2 Header file proto.h

```

/**/
/* File: proto.h
/* Function prototypes for SLALOM93 benchmark
/* Version: %G% %W%
/**/

#ifndef _PROTO_H_
#define _PROTO_H_

/**/
/* Prototypes for major SLALOM functions
/**/
int    main      (int    argc,          /* Argument count          */
               char    *argv[]);      /* Argument strings       */

void   Usage     (char    *argv[]);    /* Argument strings       */

void   Meter     (int     nlink,        /* Number of links to create */
                 double  ops[],        /* FLOPs for each phase    */
                 double  sec[]);       /* Time for each phase     */

void   Reader    (Node    *moan,        /* Pointer to root hier node */
                 int     rho[],        /* Reflectivity HierVecs   */
                 int     emiss[],      /* Emissivity HierVecs    */
                 Node    **polygons,   /* Dyn. alloc. list of poly */
                 int     *npoly,       /* Number of polygons read  */
                 double  *work);       /* #FLOPs to do the job   */

void   Refsol    (Node    *moan,        /* Pointer to root hier node */
                 int     rho[],        /* Reflectivity HierVecs   */
                 int     emiss[],      /* Emissivity HierVecs    */
                 int     x[],          /* Solution radiosities    */
                 int     reqlinks,     /* Requested number of links */
                 int     area,         /* Hiervec of patch areas  */
                 int     p,           /* Hiervec temporary      */
                 int     rowsums,      /* Coupling matrix row sums */
                 double  sec[],        /* Seconds for each phase  */
                 double  work[],       /* FLOPs for each phase    */
                 int     stats[]);     /* Link statistics        */

float  Solver    (Node    *moan,        /* Pointer to root hier node */
                 int     rho,          /* Reflectivity HierVec    */
                 int     emiss,        /* Emissivity HierVec     */
                 int     x,           /* Solution radiosities    */
                 float  epsilon,      /* Required solution accuracy */
                 int     area,         /* Hiervec of patch areas  */
                 int     p,           /* Hiervec temporary      */
                 int     rowsums,      /* Coupling matrix row sums */
                 double  *work);       /* #FLOPs to do the job   */

void   Storer    (Node    *moan,        /* Pointer to root hier node */
                 int     x[],          /* Solution radiosities    */
                 Node    **polys,     /* List of polygon pointers */
                 int     npoly,       /* Number of initial polys */
                 double  *work);       /* #FLOPs to do the job   */

void   What      (int     nlink,        /* No. of links in solution */
                 double  ops[],        /* FLOPs for each phase    */
                 double  sec[]);       /* Seconds for each phase  */

double When      (void);               /* Timer call              */

void   Writegeom (Node    *p,          /* Pointer to hierarchy node */
                 FILE    *fp,         /* Pre-opened answer file  */

```

```

        int    d0,          /* Red radiosity slot number */
        int    d1,          /* Blu radiosity slot number */
        int    d2,          /* Grn radiosity slot number */
        double *work);     /* #FLOPs to do the job */

void    Writelinks (FILE   *fp);          /* Interactions file */

float   Cfest      (Node   *moan,        /* Pointer to root hier node */
                  Node   *p,            /* Pointer to hierarchy node */
                  Node   *q,            /* Pointer to hierarchy node */
                  float  *err,          /* Est. of error in coupling */
                  Visible *vis,        /* Visibility of the link */
                  double *work,        /* #FLOPs to do the job */
                  int    stats[]);      /* Link statistics */

void    Refine     (Node   *moan,        /* Pointer to root hier node */
                  int    *reqlinks,     /* # of links after refine */
                  float  epsilon,       /* Req'd refinement accuracy */
                  double *work,        /* #FLOPs to do the job */
                  int    stats[]);      /* Link statistics */

/**/
/* LinkQueue functions
/**/
LinkQueue *Lqalloc(void);              /* Construct a new LinkQueue. */
void       Lqfree(LinkQueue*);         /* Destroy a Linkqueue. */
void       Lqenqueue(LinkQueue*,Link); /* Enqueue an element at head. */
Link       Lqdequeue(LinkQueue *);     /* Dequeue an element from tail. */
int        Lqlength(LinkQueue *);      /* Return # of Links in queue. */
void       Lqextend(LinkQueue *, int);  /* Extend queue to a new size. */
void       Lqprint (LinkQueue *);      /* Pretty-print the queue. */

/**/
/* Priority queue of Links
/**/
LinkHeap *Lhalloc(void);               /* Construct a new LinkHeap. */
void      Lhfree(LinkHeap*);           /* Destroy a LinkHeap. */
void      Lhenqueue (LinkHeap*,Link, int[]); /* Enqueue an element at head. */
Link      Lhdequeue (LinkHeap *, int[]); /* Dequeue an element from tail. */
void      Lhclear (LinkHeap *);        /* Empty out the heap. */
void      Lhheapify(LinkHeap *, int);   /* Reheapify due to new element. */
void      Lhrebuild(LinkHeap *);       /* Rebuild heap from scratch. */
void      Lhextend (LinkHeap *, int);   /* Extend the heap to new size. */
void      Lhprint (LinkHeap *);        /* Pretty-print the heap. */

/**/
/* Link functions
/**/
void      Lupdate(Link *);              /* Update epq element based on current */
                                                /* values in solelem[RGB] and cpq. */

/**/
/* Hierarchy node functions
/**/
Node      *Nodealloc (NodeType);        /* Allocate a new node structure.*/
void      Nodefree (Node *);            /* Free up a tree of Nodes. */
void      Nodecopy (Node *, Node *);    /* Copy a node. */
void      Nodeinit (Node *);            /* Initialize center,normal,etc. */
int       Getlevel (Node *);            /* Return Node's level in hier. */
int       Numelem (Node *);             /* Return the number of nodes. */
int       Numleaves (Node *);          /* Return number of leaf nodes. */
void      Subdiv (Node *);              /* Subdivide a polygon. */
void      Makepoly (Node *, Vector, Vector, Vector, Vector); /* Init. polygon from 4 verts. */
void      Makecomp (Node*, Node*, Node*); /* Init. composite from 2 polys. */
Visible   Occlusion (Node *moan, Node *p, Node *q, double *work); /* Determine if p visible from q.*/

```

```

void      Nodeprint(Node *, int, int);      /* Pretty-print a hiervec.      */
/**/
/* Hierarchical vector functions
**/
int      Halloc(void);                      /* Allocate vector and return #. */
void     Hfree(int d);                     /* Free a vector.                */
void     Hprep(Node *, int);               /* Bubble area-weighted sums up. */
void     Prop(Node *, int);                /* Trial optimization of up & dn */
void     Propup(Node *, int);              /* Third phase of matvecmult.    */
void     Propdn(Node *, int);              /* Fourth phase of matvecmult.   */
void     Matvecmult                         /* Multiply coupling matrix by   */
      (Node *nd, int b, int x);            /* vector x giving vector b.    */
void     Hadd(Node*, int, int, int);        /* Vector d += vector s.         */
void     Hsub(Node*, int, int, int);        /* Vector d += vector s.         */
void     Hmult(Node*, int, int, int);       /* Vector d *= vector s.         */
void     Hscale(Node*,int,int,float);       /* Vector d *= scalar s.         */
float    Hdot (Node *, int, int);           /* Return dot product of d and s.*/
void     Hneg(Node *, int, int);            /* Vector d = -d.                */
void     Hinvert(Node *, int, int);         /* Vector d = 1.0 / vector d.    */
void     Hcopy(Node *, int, int);           /* Vector d = vector s.          */
void     Hfill(Node *, int, float);         /* Vector d = constant.          */
float    Hnorm(Node *, int);                /* One norm of a vector.         */
float    Hinfnorm(Node *, int);             /* Infinity norm of a vector.    */
void     Hgetarea(Node *, int);             /* Get area of each node into vec*/
void     Hprint(Node *, int);               /* Pretty-print a vector.        */
void     Hmma
      (Node *, int, int, int, int);
void     Hmsmsm
      (Node *, float *, int, int, int, int, int, int);

/**/
/* 3-vector functions
**/
void     Vzzero(Vector);                    /* Assign zero to a vector.      */
void     Vprint(Vector);                    /* Pretty-print a vector.        */
float    Vmag(Vector);                       /* Magnitude of vector           */
void     Vscale(Vector, Vector, float);      /* Scale vector by a scalar.     */

/**/
/* Macros
**/
#define Fmax(a,b) ((a) > (b)) ? (a) : (b)
#define Fmin(a,b) ((a) < (b)) ? (a) : (b)
#define Vcopy(d,a) (d[0]=a[0], d[1]=a[1], d[2]=a[2])
#define Vdiff(d,a,b) (d[0]=a[0]-b[0], d[1]=a[1]-b[1], d[2]=a[2]-b[2])
#define Vsum(d,a,b) (d[0]=a[0]+b[0], d[1]=a[1]+b[1], d[2]=a[2]+b[2])
#define Vdot(a,b) (a[0]*b[0] + a[1]*b[1] + a[2]*b[2])
#define Vmagsq(a) (a[0]*a[0] + a[1]*a[1] + a[2]*a[2])
#define Vcross(d,a,b) (d[0] = a[1]*b[2] - b[1]*a[2], \
                      d[1] = b[0]*a[2] - a[0]*b[2], \
                      d[2] = a[0]*b[1] - b[0]*a[1])

#endif          _PROTO_H_

```

A.3 Source file slal.c

```

/**/
/* File: slal.c
/* High-level driver functions for the SLALOM93 benchmark.
/* Version: %G% %W%
/**/

#include          "slal.h"
#include          "proto.h"

/**/
/* Global variables:
/**/
debugflag      debug      = none;

extern char *optarg;
extern int  optind;

int
main (int argc, char *argv[])
{
    double      ops[4],          /* Operation count for each task */
              sec[4];          /* Seconds for each major task */
    int         lnkreq,         /* Requested number of links */
              c;

    /* Parse the command line flags. */
    while ((c = getopt(argc, argv, "d:")) != EOF)
        switch (c) {
            case 'd':
                if ((debug = atoi(optarg)) == 0)
                    debug = (debugflag) 0xffffffff;
                break;
            default:
            case '?':
                Usage (argv);
                break;
        }

    /* Make sure that the syntax of invocation is OK. */
    if (argc-optind != 0) {
        Usage (argv);
        exit (1);
    }

    printf("How many links? ");
    scanf("%d", &lnkreq);

    Meter(lnkreq, ops, sec);
    What (lnkreq, ops, sec);
}

void
Usage (char *argv[])
{
    printf ("Usage: %s [-d debuglevel]\n", argv[0]);
    printf (" none      = 0, links   = 1,\n");
    printf (" refine   = 2, vectors = 4,\n");
    printf (" iterate  = 8, matrix  = 16,\n");
    printf (" hierarchy= 32, isect   = 64\n");
    exit (1);
}

void
Meter (int lnkreq, double ops[], double sec[])

```



```

{
double      work;
Node        *moan;
Node        *polygons[maxpoly];
int         area,
           emiss[3],
           npoly,
           p,
           rho[3],
           rowsums,
           stats[3],
           x[3];

/* Create root hierarchy node, and all HierVecs needed by Reader(). */
stats[visible] = stats[partial] = stats[blocked] = 0;
moan = Nodealloc(composite);
heap = Lhalloc();
lhextend(heap, lnkreq);
x[0] = Halloc();
x[1] = Halloc();
x[2] = Halloc();
rho[0] = Halloc();
rho[1] = Halloc();
rho[2] = Halloc();
emiss[0] = Halloc();
emiss[1] = Halloc();
emiss[2] = Halloc();
area = Halloc();
p = Halloc();
rowsums = Halloc();

/* Read in patch geometries from file "geom". */
sec[0] = When();
Reader(moan, rho, emiss, polygons, &npoly, &work);
sec[0] = When() - sec[0];
ops[0] = work;

/* Set up the patch couplings and solve for RGB patch radiosities. */
Refsol(moan, rho, emiss, x, lnkreq, area, p, rowsums, sec, ops, stats);

/* Write out some useful statistics. */
Hfill(moan, p, 1.0);
Matvecmult(moan, rowsums, p);
Hgetarea(moan, area);
printf("Total form factor = %g\n",
       Hdot(moan, rowsums, area) / Hdot(moan, p, area));
printf("  Links in hierarchy = %d\n", heap->tail);
printf("  Elements in hierarchy = %d\n", Numelem(moan));
printf("  Patches in hierarchy = %d\n", Numleaves(moan));
printf("Average links per element = %g\n",
       ((float) heap->tail / Numelem(moan)));
printf("  Totally visible links = %d\n", stats[visible]);
printf("  Partly visible links = %d\n", stats[partial]);
printf("  Occluded links = %d\n", stats[blocked]);
printf("Approximate memory usage: %d bytes.\n",
       (int) (Numelem(moan) * sizeof(Node) + heap->tail * sizeof(Link));

/* Write radiosities and patch geometries to the 'answer' file. */
sec[3] = When();
Storer(moan, x, polygons, npoly, &work);
sec[3] = When() - sec[3];
ops[3] = work;

/* Release dynamically allocated storage. */
Lhfree(heap);
Hfree(x[0]);
Hfree(x[1]);
Hfree(x[2]);
}

```

```

    Hfree(rho[0]);
    Hfree(rho[1]);
    Hfree(rho[2]);
    Hfree(emiss[0]);
    Hfree(emiss[1]);
    Hfree(emiss[2]);
    Hfree(area);
    Hfree(p);
    Hfree(rowsums);
}

void
What(int lnkreq, double ops[], double sec[])
{
    int          i;
    float        totaltime;
    double       totalwork;
    static char *names[] = { "Reader", "SetUp", "Solver", "Storer" };
    static char *format = "%6.6s%8.2f%17.0f%14.6f%10.1f %%\n";

    /* Print out a summary of timing information for this run. */
    totaltime = sec[0] + sec[1] + sec[2] + sec[3];
    totalwork = ops[0] + ops[1] + ops[2] + ops[3];
    printf("\n%d links:\n", lnkreq);
    printf(" Task  Seconds      Operations      MFLOPS      %% of Time\n");
    for (i = 0 ; i < 4 ; i++) {
        printf(format, names[i], sec[i], ops[i],
              (ops[i] / sec[i]) * 1e-6, 100.0 * sec[i] / totaltime);
    }
    printf(format, "TOTALS", totaltime, totalwork,
          (totalwork / totaltime) * 1e-6, 100.0);
}

/**/
/* Read in the geometry description file and produce a hierarchy of
/* rectangles and triangles below the root node 'moan'.
/**/
void
Reader(Node *moan, int (rhohv)[3], int (emshv)[3],
       Node **polygons, int *npoly, double *work)
{
    FILE          *infile;
    LinkQueue    *queue;
    Link         lnk;
    Node         *p, *q, *P;
    Vector       rho, ems,
                vtmp,
                vlist[maxvert];
    unsigned     r0, r1, r2,
                e0, e1, e2,
                nvert,
                vnum,
                v0, v1, v2, v3,
                lineno = 0;
    char         buff[257], word[32];

    (*work) = 0;

    *npoly = 0;
    queue = Lqalloc();

    /* Get the slot numbers of each component of reflectivity and emissivity.*/
    r0 = rhohv[0];
    r1 = rhohv[1];
    r2 = rhohv[2];
    e0 = emshv[0];
    e1 = emshv[1];
    e2 = emshv[2];

```

```

/* Open the geometry file. */
if ((infile = fopen("geom", "r")) == NULL) {
    fprintf(stderr, "Unable to open 'geom' file.\n");
    exit(1);
}

/* Read and create the polygons. */
while (fgets(buff, 256, infile) != NULL) {
    lineno++;

    /* Parse the first word. */
    if (sscanf(buff, "%s", word) != 1) {
        fprintf(stderr, "Bogus line number %d.\n", lineno);
        exit(1);
    }

    /* Check for a comment indicator. */
    if (word[0] == '#') {
        continue;
    }

    /* Check for each type of leading word allowed. */
    if (strcmp(word, "polyhedron") == 0)
        /* Reset vertex list. */
        nvert = 0;
    else if (strcmp(word, "polygon") == 0) {
        /* Idiot check. */
        if (nvert < 3) {
            printf("Reader: Too few vertices defined. Line %d.\n", lineno);
            exit(1);
        }
        if (*npoly >= maxpoly) {
            printf("Reader: Too many polygons; Maximum = %d.\n", maxpoly);
            exit(1);
        }

        /* Parse the remainder of the line. */
        if (sscanf (buff, "%*s%d%d%d %f%f%f %f%f%f", &v0, &v1, &v2, &v3,
            &rho[0], &rho[1], &rho[2], &ems[0], &ems[1], &ems[2]) != 10) {
            printf("Reader: Bad polygon format. Line %d.\n", lineno);
            exit(1);
        }

        /* Check the vertex indices. */
        if (v0 >= nvert || v1 >= nvert || v2 >= nvert || v3 >= nvert) {
            printf("Reader: Bad vertex number. Line %d.\n", lineno);
            exit(1);
        }

        /* Range check the reflectivity. */
        if (rho[0] < .001-eps || rho[1] < .001-eps || rho[2] < .001-eps ||
            rho[0] > .999+eps || rho[1] > .999+eps || rho[2] > .999+eps) {
            printf("Reader: Reflectivity out of range. Line %d.\n", lineno);
            printf("      Must be in the range 0.001 <= rho <= 0.999\n");
            exit(1);
        }
        (*work) += 6;

        /* Install the polygon. */
        P = Nodealloc(patch);
        Makepoly(P, vlist[v0], vlist[v1], vlist[v2], vlist[v3]);
        (*work) += 180; /* Makepoly */
        if (P->area == 0.0) {
            fprintf(stderr, "Reader: Bad polygon at line %d.\n", lineno);
            exit(1);
        }
        P->e[r0] = rho[0];
    }
}

```

```

P->e[r1] = rho[1];
P->e[r2] = rho[2];
P->e[e0] = ems[0];
P->e[e1] = ems[1];
P->e[e2] = ems[2];
lnk.p = P;
Lqenqueue(queue, lnk);
polygons[*npoly] = P;
(*npoly)++;
(*work) += 4 + 24;      /* constructor + setnormal() */
}
else if (strcmp(word, "vertex") == 0) {
/* Parse the remainder of the line. */
if (sscanf(buff, "%s%d%f%f%f", &vnum,
&vtmp[0], &vtmp[1], &vtmp[2]) != 4) {
printf("Reader: Need x,y,z vertex coords. Line %d.\n", lineno);
exit(1);
}

/* Check the vertex number. */
if (vnum != nvert) {
printf("Reader: Need sequential vertex numbers. Line %d.\n",
lineno);
exit(1);
}

/* Check for vertex list full. */
if (nvert >= maxvert) {
printf("Reader: Vertex list full. Line %d.\n", lineno);
exit(1);
}

/* Add the vertex to the vertex list. */
Vcopy(vlist[nvert], vtmp);
nvert++;
}
else {
printf("Reader: Line number %d is total garbage.\n", lineno);
exit(1);
}
}
printf("Reader: Read %d polygons from 'geom' file.\n", *npoly);

/* Form the hierarchy above the polygons. */
while (Lqlength(queue) > 2) {
lnk = Lqdequeue(queue);
p = lnk.p;
lnk = Lqdequeue(queue);
q = lnk.p;
lnk.p = Nodealloc(composite);
Makecomp(lnk.p, p, q);
Lqenqueue(queue, lnk);
}
lnk = Lqdequeue(queue);
p = lnk.p;
lnk = Lqdequeue(queue);
q = lnk.p;
moan->left = p;
moan->left->parent = moan;
moan->right = q;
moan->right->parent = moan;
Nodeinit(moan);
Hprep(moan, emshv[0]);
Hprep(moan, emshv[1]);
Hprep(moan, emshv[2]);
Hprep(moan, rhohv[0]);
Hprep(moan, rhohv[1]);
Hprep(moan, rhohv[2]);

```

```

/* Debug dump of the completed hierarchy. */
if (debug & hierarchy) {
    printf("INITIAL HIERARCHY:\n");
    Nodeprint(moan, 2, 0);
}

(work) += (*npoly - 1) * 14;      /* Composite constructors. */
Lqfree(queue);
}

/**/
/* Solve for the equilibrium balance of energy transfer in the scene using
/* the diagonally preconditioned conjugate gradient method. The matrix of
/* form factors is implied by the links in the 'moan' hierarchy.
/**/
void
Refsol(Node *moan, int (rho)[3], int (emiss)[3], int (x)[3], int reqlinks,
        int area, int p, int rowsums, double sec[], double ops[], int stats[])
{
    double        timel,
                  time2,
                  work;
    Vector         soleps;
    Visible        vis;
    float          lnkeps,
                  err,
                  t;
    int            m,
                  iterates,
                  numlinks;
    Link           lnk;

    work = 0.0;
    sec[1] = sec[2] = 0.0;
    ops[1] = ops[2] = 0.0;
    timel = When();

    LhcLEAR(heap); /* Clear the heap, and prime it for the solution phase. */
    solelemR = x[0];
    solelemG = x[1];
    solelemB = x[2];
    rhoelemR = rho[0];
    rhoelemG = rho[1];
    rhoelemB = rho[2];
    vis = partial;
    lnk.p = moan;
    lnk.q = moan;
    lnk.cpq = Cfest(moan, moan, moan, &err, &vis, &work, stats);
    lnk.err = err;
    lnk.vis = vis;
    Lhenqueue(heap, lnk, stats);
    Hfill(moan, x[0], 1.0);
    Hfill(moan, x[1], 1.0);
    Hfill(moan, x[2], 1.0);
    Hprep(moan, x[0]);
    Hprep(moan, x[1]);
    Hprep(moan, x[2]);
    lnkeps = big;
    soleps[0] = big;
    soleps[1] = big;
    soleps[2] = big;

    /* Do an initial subdivision. */
    LhREBUILD(heap);
    work += 22 * heap->tail;
    lnkeps = heap->p[0].epq;
    numlinks = 1;
}

```

```

Refine(moan, &numlinks, lnkeps, &work, stats);
printf("Made %d initial links.\n", numlinks);
Hcopy(moan, x[0], emiss[0]);
Hcopy(moan, x[1], emiss[1]);
Hcopy(moan, x[2], emiss[2]);
Hprep(moan, x[0]);
Hprep(moan, x[1]);
Hprep(moan, x[2]);
Lhrebuild(heap);
work += 22 * heap->tail;
lnkeps = heap->p[0].epq;

/* Iterate until we have the right number of links. */
printf("\n/-----ERROR ESTIMATES-----\\n");
printf(" Red   | Green | Blue | Link # of Links ErrProd\n");
printf("=====|=====|=====|=====|=====\\n");
while (numlinks < reqlinks) {

    t = Fmax(soleps[0], Fmax(soleps[1],
        Fmax(soleps[2], lnkeps))) * numlinks;
    printf("%7.1e | %7.1e | %7.1e | %7.1e %9d %7g ",
        soleps[0], soleps[1], soleps[2],
        lnkeps, numlinks, t);

    iterates = 0;
    for (m = 0 ; m < 3 ; m++) {
        if (soleps[m] >= lnkeps) {
            ops[1] += work;
            work = 0;
            sec[1] += (time2 = When()) - timel;
            timel = time2;
            soleps[m] = Solver(moan, rho[m], emiss[m], x[m],
                lnkeps, area, p, rowsums, &work);
            ops[2] += work;
            work = 0;
            sec[2] += (time2 = When()) - timel;
            timel = time2;
            Hprep(moan, x[m]);
            iterates = 1;
        }
    }
    printf("\n");
    if (iterates) {
        Lhrebuild(heap);
        work += 22 * heap->tail;
    }
    else {
        numlinks = reqlinks;
        /* Force stopping criterion to numlinks in the last Refine call. */
        t = Fmax(soleps[0], soleps[1]);
        t = Fmax(t, soleps[2]);
        Refine(moan, &numlinks, t, &work, stats);
    }
    lnkeps = heap->p[0].epq;
}

/* Finishing iteration. */
t = Fmax(soleps[0], Fmax(soleps[1],
    Fmax(soleps[2], lnkeps))) * numlinks;
printf("%7.1e | %7.1e | %7.1e | %7.1e %9d %7g ",
    soleps[0], soleps[1], soleps[2],
    lnkeps, numlinks, t);
for (m = 0 ; m < 3 ; m++) {
    ops[1] += work;
    work = 0;
    sec[1] += (time2 = When()) - timel;
    timel = time2;
    soleps[m] = Solver(moan, rho[m], emiss[m], x[m],

```

```

        lnkeys, area, p, rowsums, &work);
ops[2] += work;
work = 0;
sec[2] += (time2 = When()) - timel;
timel = time2;
}
printf("\n\n");

printf("Number of patches in hierarchy = %d\n", Numleaves(moan));

ops[1] += work;
sec[1] += When() - timel;
}

/**/
/* Storer() output is designed to be order-independent; use a sort
/* utility to restore the file to an easily-readable form.
/**/
void
Storer(Node *moan, int x[], Node **polygons, int npoly, double *work)
{
    int i, k;
    static char *fmt1 = "%4d vertex %1d %9.4f %9.4f %9.4f\n";
    FILE *fp; /* Output file pointer. */

    (*work) = 0;

    /**/
    /* Write patch geometry and radiosities to 'answer' file. */
    /**/
    if ((fp = fopen("answer", "w")) == NULL) {
        fprintf(stderr, "Unable to open 'answer' file.\n");
        exit(1);
    }

    /* Write out the vertices of each polygon. */
    fprintf(fp, "%d polygons:\n", npoly);
    for (i = 0 ; i < npoly ; i++)
        for (k = 0 ; k < 4 ; k++)
            fprintf(fp, fmt1, i + 1, k, polygons[i]->vertex[k][0],
                    polygons[i]->vertex[k][1],
                    polygons[i]->vertex[k][2]);

    (*work) += 168 * npoly;

    fprintf(fp, "%d patches:\n", Numleaves(moan));
    Writegeom(moan, fp, x[0], x[1], x[2], work);
    fclose(fp);

    /**/
    /* Write patch interactions to 'links' file.
    /**/
    if ((fp = fopen("links", "w")) == NULL) {
        fprintf(stderr, "Unable to open 'links' file.\n");
        exit(1);
    }
    if (heap->tail > 10000)
        fprintf(fp, "0 links:\n");
    else {
        fprintf(fp, "%d links:\n", heap->tail);
        Writelinks(fp);
    }
    fclose(fp);
}

/**/
/* Recursive part of Storer() that traverses the patch hierarchy and
/* writes out the geometry and radiosity information associated with
/* each leaf patch.

```

```

/**/
void
Writegeom(Node *p, FILE *fp, int d0, int d1, int d2, double *work)
{
    static char *fmt2 = "%16d %4d answer rgb   %9.4f %9.4f %9.4f\n";
    static char *fmt3 = "%16d %4d vertex %1d   %9.4f %9.4f %9.4f\n";
    int i;
    static int polynum, patnum;

    /* Set patch number to zero if this routine has just been called. */
    if (p->parent == NULL)
        patnum = 1;
    if (p->id > 0 && p->parent && p->parent->id <= 0)
        polynum = p->id;

    /* Recur to the left and right if body node, else write out the patch. */
    if (p->left)
        Writegeom(p->left, fp, d0, d1, d2, work);
    else {
        fprintf(fp, fmt2, patnum, polynum, p->e[d0], p->e[d1], p->e[d2]);
        for (i = 0 ; i < 4 ; i++)
            fprintf(fp, fmt3, patnum, polynum, i,
                p->vertex[i][0], p->vertex[i][1], p->vertex[i][2]);
        patnum++;
        (*work) += 210;
    }
    if (p->right)
        Writegeom(p->right, fp, d0, d1, d2, work);
}

/**/
/* Recursive part of Storer() for writing out an exhaustive list of
/* links between patches. This routine is present for debugging
/* purposes only.
/**/
void
Writelinks(FILE *fp)
{
    Link *P;
    int i;
    static char *fmt4 =
        "%2d %4d ==> %4d %6.3f %6.3f %6.3f %6.3f %6.3f %6.3f %6.3f %d\n";

    for (i = 0, P = heap->p ; i < heap->tail ; i++) {
        fprintf(fp, fmt4, Getlevel(P->p) + Getlevel(P->q),
            P->p->id, P->q->id,
            P->p->center[0], P->p->center[1], P->p->center[2],
            P->q->center[0], P->q->center[1], P->q->center[2],
            P->cpq, P->vis);
        P++;
    }
}

double
When()
{
    struct timeval tp;
    gettimeofday(&tp, NULL);
    return ((double) tp.tv_sec + (double) tp.tv_usec * 1e-6);
}

```


A.4 Source file solver.c

```

/**/
/* File: solver.c
/* Link refinement, coupling factor estimation, and system solver for SLALOM93.
/* Version: %G% %W%
/**/

#include          "slal.h"
#include          "proto.h"

/**/
/* Refine Node p against Node q.  The two Nodes are adaptively subdivided
/* such that the requested number of interactions (reqlinks) are formed.
/* If reqlinks is passed in as -1, then subdivision is performed until all
/* coupling factors in excess of "big" are drained from the heap.
/**/
void
Refine(Node *moan, int *reqlinks, float epsilon, double *work, int stats[])
{
    float          cpq, epq,
                  cpq1, cpq2, cpq3,
                  Cp1q, Cp2q,
                  err1, err2, err3,
                  Ap, Aq;
    Node           *p, *q;
    Link           lnk, ltemp;
    Visible        vis, vis1, vis2, vis3;

    /* Split links until requested number of links is obtained or */
    /* the link error drops below that of the solver. i.e. epsilon. */
    while ((*reqlinks > 0 && heap->tail < *reqlinks) || (*reqlinks < 0)) {
        /* Take the Link from the heap with the largest estimated error. */
        lnk = Lhdequeue(heap, stats);
        p = lnk.p;
        q = lnk.q;
        cpq = lnk.cpq;
        epq = lnk.epq;
        Ap = p->area;
        Aq = q->area;
        vis = lnk.vis;

        /* If reqlinks is -1, and the links just removed from the heap */
        /* is not of ridiculous coupling, then terminate the subdivision. */
        if ((*reqlinks < 0 && epq < big) || epq < epsilon) {
            /* Put the link back onto the heap and break. */
            Lhenqueue(heap, lnk, stats);
            break;
        }

        if (debug & refine) {
            printf(
                "Refining (%2d=>%2d): Fpq=%g cpq=%g epq=%g Qlen=%d vis=%d.\n",
                p->id, q->id, cpq/Ap, cpq, epq, heap->tail, vis);
        }

        /* If p and q are the same node, subdivide 3 ways instead of 2. */
        if (p == q) {
            if (p->id <= 0) {
                vis1 = vis2 = vis3 = vis;
                cpq1 = Cfest(moan, p->left, p->left, &err1, &vis1, work, stats);
                cpq2 = Cfest(moan, p->left, p->right, &err2, &vis2, work, stats);
                cpq3 = Cfest(moan, p->right, p->right, &err3, &vis3, work, stats);
                if (debug & refine) {
                    printf(" Estimates: (%2d to %2d) = %g\n",
                        p->left->id, p->left->id, cpq1);
                }
            }
        }
    }
}

```

```

        printf("                (%2d to %2d) = %g\n",
               p->left->id, p->right->id, cpq2);
        printf("                (%2d to %2d) = %g\n",
               p->right->id, p->right->id, cpq3);
    }
    if (cpq1 > 0.0) {
        ltemp.p = p->left;
        ltemp.q = p->left;
        ltemp.cpq = cpq1;
        ltemp.err = err1;
        ltemp.vis = vis1;
        Lupdate(&ltemp);
        Lhenqueue(heap, ltemp, stats);
        (*work) += 22;
    }
    if (cpq2 > 0.0) {
        ltemp.p = p->left;
        ltemp.q = p->right;
        ltemp.cpq = cpq2;
        ltemp.err = err2;
        ltemp.vis = vis2;
        Lupdate(&ltemp);
        Lhenqueue(heap, ltemp, stats);
        (*work) += 22;
    }
    if (cpq3 > 0.0) {
        ltemp.p = p->right;
        ltemp.q = p->right;
        ltemp.cpq = cpq3;
        ltemp.err = err3;
        ltemp.vis = vis3;
        Lupdate(&ltemp);
        Lhenqueue(heap, ltemp, stats);
        (*work) += 22;
    }
}
else
    printf("Refine: p==q and is not composite!\n");
)

/* Subdivide p if it is larger or is the only composite of the pair. */
else if (
    (p->id <= 0 && q->id > 0) ||
    (Ap > Aq && p->id <= 0) ||
    (Ap > Aq && q->id > 0)) {
    Subdiv(p);
    (*work) += 96;                /* Subdiv */
    vis1 = vis2 = vis;
    Cp1q = Cfest(moan, p->left, q, &err1, &vis1, work, stats);
    Cp2q = Cfest(moan, p->right, q, &err2, &vis2, work, stats);
    if (debug & refine) {
        printf(" Split patch %d into patches %d and %d.\n",
               p->id, p->left->id, p->right->id);
        printf(" Cp1q = %g Cp2q = %g\n", Cp1q, Cp2q);
    }
    if (Cp1q > 0.0) {
        ltemp.p = p->left;
        ltemp.q = q;
        ltemp.cpq = Cp1q;
        ltemp.err = err1;
        ltemp.vis = vis1;
        Lupdate(&ltemp);
        Lhenqueue(heap, ltemp, stats);
        (*work) += 22;
    }
    if (Cp2q > 0.0) {
        ltemp.p = p->right;
        ltemp.q = q;

```

```

        ltemp.cpq = Cp2q;
        ltemp.err = err2;
        ltemp.vis = vis2;
        Lupdate(&ltemp);
        Lhenqueue(heap, ltemp, stats);
        (*work) += 22;
    }
}

/* Subdivide q because it has the larger area. */
else {
    Subdiv(q);
    (*work) += 96;          /* Subdiv */
    vis1 = vis2 = vis;
    cpq1 = Cfest(moan, p, q->left, &err1, &vis1, work, stats);
    cpq2 = Cfest(moan, p, q->right, &err2, &vis2, work, stats);
    if (debug & refine) {
        printf(" Split patch %d into patches %d and %d.\n",
               q->id, q->left->id, q->right->id);
        printf(" Cp1q=%g Cp2q=%g\n", cpq1, cpq2);
    }
    if (cpq1 > 0.0) {
        ltemp.p = p;
        ltemp.q = q->left;
        ltemp.cpq = cpq1;
        ltemp.err = err1;
        ltemp.vis = vis1;
        Lupdate(&ltemp);
        Lhenqueue(heap, ltemp, stats);
        (*work) += 22;
    }
    if (cpq2 > 0.0) {
        ltemp.p = p;
        ltemp.q = q->right;
        ltemp.cpq = cpq2;
        ltemp.err = err2;
        ltemp.vis = vis2;
        Lupdate(&ltemp);
        Lhenqueue(heap, ltemp, stats);
        (*work) += 22;
    }
}
}
}
*reqlinks = heap->tail;
}

/**/
/* Return a coupling factor estimate from patch p to patch q (symmetric).
**/
float
Cfest(Node *moan, Node *p, Node *q, float *err, Visible *vis, double *work,
      int stats[])
{
    Vector Rij;
    float Cest, Cpart,
          cpqmin, cpqmax,
          magsq,
          e1, e2;
    int i;

    /* Check for null pointer. Return 0.0 if null. */
    if (!p || !q) {
        *err = 0.0;
        return 0.0;
    }

    /* If either p or q is a composite, use the product */
    /* of their areas over total area as coupling factor. */

```

```

if (p->id <= 0 || q->id <= 0) {
    (*work) += 6;
    *err = Fmax(p->area, q->area);
    return p->area * q->area / moan->area;
}

/* Both p and q are polygons. Check to see if Cfest has been */
/* called with p == q. If so, return 0. */
if (p == q) {
    *err = 0.0;
    return 0.0;
}

/* If the link inherited partial visibility, do an occlusion test. */
/* Return early if occluded. */
if (*vis == partial) {
    *vis = Occlusion(moan, p, q, work);
    if (debug & refine) {
        printf (" Occlusion(%3d->%3d) returns %s\n", p->id, q->id,
            (*vis == partial) ? "partial" :
            ((*vis == visible) ? "visible" : "blocked"));
    }
    if (*vis == blocked) {
        stats[blocked]++;
        *err = 0.0;
        return 0.0;
    }
}

/* Calculate some geometry constants. */
Cest = 0.0;
cpqmax = 0.0;
cpqmin = big;

/* Do the two center-to-center estimates. */
Vdiff(Rij, q->center, p->center);
magsq = Vmagsq(Rij);
e1 = Vdot(p->normal, Rij);
e2 = - Vdot(q->normal, Rij);
Cpart = e1 * e2 / (magsq * (magsq * PI));
cpqmin = Fmin(cpqmin, Cpart);
cpqmax = Fmax(cpqmax, Cpart);
if (Cpart > 0.0)
    Cest += Cpart;
Cpart = e1 * e2 / (magsq * (magsq * PI));
cpqmin = Fmin(cpqmin, Cpart);
cpqmax = Fmax(cpqmax, Cpart);
if (Cpart > 0.0)
    Cest += Cpart;

/* Do the eight off-center estimates. */
for (i = 0 ; i < 4 ; i++) {
    Vdiff (Rij, q->vertex[i], p->center);
    magsq = Vmagsq(Rij);
    if (magsq < eps * eps)
        magsq = big;
    (*work) += 8;

    /* Finally, do the coupling estimate using disk method. */
    e1 = Vdot(p->normal, Rij);
    e2 = - Vdot(q->normal, Rij);
    Cpart = (e1 * e2) / (magsq * (magsq * PI));
    cpqmin = Fmin(cpqmin, Cpart);
    cpqmax = Fmax(cpqmax, Cpart);
    if (Cpart > 0.0)
        Cest += Cpart;
    (*work) += 21;
}

```

```

}

for (i = 0 ; i < 4 ; i++) {
  Vdiff (Rij, q->center, p->vertex[i]);
  magsq = Vmagsq(Rij);
  if (magsq < eps * eps)
    magsq = big;
  (*work) += 8;

  /* Finally, do the coupling estimate using disk method. */
  e1 = Vdot(p->normal, Rij);
  e2 = - Vdot(q->normal, Rij);
  Cpart = (e1 * e2) / (magsq * (magsq * PI));
  cpqmin = Fmin(cpqmin, Cpart);
  cpqmax = Fmax(cpqmax, Cpart);
  if (Cpart > 0.0)
    Cest += Cpart;
  (*work) += 21;
}

(*work) += 3;
if (*vis == partial)
  cpqmin = Fmin(cpqmin, 0.0);
*err = cpqmax - cpqmin;
return Cest * (float) 0.1;
}

float
Solver(Node *moan, int rho, int emiss, int x, float epsilon,
        int area, int p, int rowsums, double *work)
{
  int          i,
              nleaf, nlink;
  float        resid;
  int          rhoinv, t;

  rhoinv = Halloc();
  t       = Halloc();
  Hgetarea(moan, area);
  Hfill(moan, p, 1.0);
  Matvecmult(moan, rowsums, p);
  Hinvert(moan, rowsums, rowsums);
  Hinvert(moan, rhoinv, rho);
  nleaf = Numleaves(moan);
  nlink = heap->tail;
  (*work) += (10 * nlink + 14 * nleaf) + 6 * nleaf;

  /**/
  /* Use Jacobi iteration to solve the system.
  /**/
  i = 0;
  do {
    /**/
    /* The following C++ expresses the Jacobi iteration.
    /* x = (moan * x) * rowsums * rho + emiss;
    /* resid = (rhoinv*area*(x-emiss) - area*rowsums*(moan*x)).infnorm();
    /**/

    /* Next iterate */
    Matvecmult(moan, x, x);
    /**/
    /* Hmult(moan, x, x, rowsums);
    /* Hmult(moan, x, x, rho);
    /* Hadd (moan, x, x, emiss);
    /**/
    Hmma (moan, x, rowsums, rho, emiss);

    /* Residual calculation */

```

```

Matvecmult(moan, p, x);
/**/
/* Hmult(moan, t, rowsums, p);
/* Hsub(moan, p, x, emiss);
/* Hmult(moan, p, p, rhoinv);
/* Hsub(moan, t, p, t);
/* Hmult(moan, t, t, area);
/* resid = Hinfnorm(moan, t);
/**/
Hmsmsmn(moan, &resid, rowsums, p, x, emiss, rhoinv, area);

putchar(planechar);
fflush(stdout);
i++;
} while (resid > epsilon);
(*work) += i * (2 * (12 * nlink + 14 * nleaf) /* Matrix-vector multiplies */
+ 9 * (nleaf)); /* Vector operations */

Hfree(rhoinv);
Hfree(t);
return resid;
}

```

A.5 Source file patch.c

```

/*****/
/* File: patch.c
/*
/* SLALOM 93
/* Functions which operate on Node type.
/*
/* The data structure constructed by this program is a
/* binary tree which consists of three distinct strata. Composite nodes form
/* the top of the patch subdivision hierarchy. These nodes are groupings of
/* polygons which are specified by the user. Next is the single level of
/* Polygons representing the scene geometry specified by the user. Finally,
/* the third and bottom-most stratum is comprised of Polygons representing
/* the subdivided user-specified polygons.
/*
/* Each Node in the patch hierarchy may have some number of "links" or
/* "interactions" with other patches in the hierarchy. Each link represents
/* the (constant) content of a single block in the coupling matrix. Thus, the
/* hierarchy forms an induced coupling matrix. Vectors with the same
/* hierarchical structure are allocated in the patch tree. These vectors are
/* called HierVecs.
/*
/* Version: %G% %W%
/*****/

#include      <stdio.h>
#include      <math.h>
#include      "slal.h"
#include      "proto.h"

/**/
/* Static declarations and functions.
/**/
static int      vecalloc = 0;
int             solelemR = 0;
int             solelemG = 0;
int             solelemB = 0;
int             rhoelemR = 0;
int             rhoelemG = 0;
int             rhoelemB = 0;
LinkHeap       *heap = NULL;
static void     Indent(int level);

/*****/
/* Node Functions */
/*****/

/**/
/* Constructor. Initialize vectors with zero.
/**/
Node *
Nodealloc(NodeType type)
{
    static int  nextid = 0; /* Composites have ID <= 0; patches have ID >= 1 */
    Node       *nd;

    nd = (Node *) malloc(sizeof(Node));
    nd->left = nd->right = nd->parent = (Node*) NULL;
    nd->t = 0.0;
    if (type == composite)
        nd->id = - (nextid++);
    else
        nd->id = nextid++;
    Vzero(nd->center);
    return nd;
}

```

```

}

/**/
/* Copy other patch geometry.
/**/
void
Nodecopy(Node *dest, Node *nd)
{
    int    i, mask;

    dest->left = dest->right = dest->parent = (Node*) NULL;
    dest->t = 0.0;
    for (i = 0 ; i < maxhiervec ; i++) {
        mask = (1 << i);
        if (vecalloc & mask)
            dest->e[i] = nd->e[i];
    }
    Vzero(dest->center);
}

/**/
/* Destructor. Delete subtrees recursively.
/**/
void
Nodefree(Node *nd)
{
    if (!nd)
        return ;
    Nodefree(nd->left);
    Nodefree(nd->right);
    free(nd);
}

/**/
/* Allocate storage for a HierVec. This method allocates a slot for storing
/* a hierarchical vector in an existing patch hierarchy. When each node in
/* the patch hierarchy is created, an array of 'maxhiervec' floats is also
/* allocated in which to store the elements of hierarchical vectors. This
/* method searches for an open slot in this array. If an open slot is found,
/* then it is mark as being in use, and the slot number is returned. If no
/* open slot is found, an error message is printed, and -1 is returned. This
/* method, since it relies on the static member 'vecalloc' to hold the
/* allocation table, may be called on any node in the hierarchy.
/**/
int
Halloc(void)
{
    int    i, mask;

    /* Find an empty vector slot, and return its index. */
    for (i = 0 ; i < maxhiervec ; i++) {
        mask = (1 << i);
        if (!(vecalloc & mask) ){
            vecalloc |= mask;
            return i;
        }
    }

    /* All vectors in use. */
    fprintf(stderr, "Halloc: All vector slots full!\n");
    return -1;
}

/**/
/* Free up a HierVec slot.
/**/
void

```



```

Hfree(int d)
{
    int    mask;

    /* Make sure the vector slot is actually in use. */
    mask = (1 << d);
    if (!(vecalloc & mask)) {
        fprintf(stderr, "Hfree: Vector not allocated to begin with!\n");
        abort();
    }

    /* Mark the vector slot as free. */
    vecalloc &= ~mask;
}

/**/
/* Hierarchical matrix-vector multiply. This method multiplies the matrix
/* induced by the nodes' link tables by a hierarchical vector, and returns
/* the hierarchical product vector. Note that the matrix which is implied
/* is the FORM FACTOR matrix, not the ACCEPTANCE FACTOR matrix. This
/* method must be called on the root node of a hierarchy. The multipli-
/* cand vector slot number is passed in 'x', and the desired product vector
/* slot number is given in 'b'. Matrix-vector multiply is accomplished in
/* four phases, which follow this method.
/**/
void
Matvecmult(Node *nd, int b, int x)
{
    int    i;
    Link   *P;

    /* Check that this is the root node of the hierarchy. */
    if (nd->parent != NULL) {
        fprintf(stderr, "Matvecmult() only works on root node.\n");
        exit(1);
    }

    /* Collapse vector leaf elements up the hierarchy. */
    Hprep(nd, x);

    /* Add up link contributions. */
    P = heap->p;
    for (i = 0 ; i < heap->tail ; i++) {
        P->p->t += P->cpq * P->q->e[x] / P->p->area;
        if (P->p != P->q)
            P->q->t += P->cpq * P->p->e[x] / P->q->area;
        P++;
    }

    Prop(nd, b); /* Collapse link contributions up and down the hier. */
}

/**/
/* Prepare the multiplicand vector for matrix-vector multiply by collapsing
/* its leaf nodes up the hierarchy. Note that since the multiplicand vector
/* is a vector of radiosities, a parent's radioactivity is the area-weighted
/* average of its children's radiosities.
/**/
void
Hprep(Node *p, int x)
{
    Node    *sstak[128],
            **ssp = sstak;
    char    pstak[128],
            *csp = pstak,
            c;

    if (!p)

```

```

    return;

    /* Collapse vector elements upwards. */
    *ssp++ = p;
    *csp++ = 0;
    while (ssp != sstak) {
        p = *--ssp;
        c = *--csp;
        p->t = 0.0;
        if (p->left) {
            if (c == 0) { /* Left subtree not visited. */
                ssp++;
                *csp++ = 1;
                *ssp++ = p->left;
                *csp++ = 0;
            }
            else if (c == 1) { /* Right subtree not visited. */
                ssp++;
                *csp++ = 2;
                *ssp++ = p->right;
                *csp++ = 0;
            }
            else { /* Visit this node. */
                p->e[x] = (p->left->area * p->left->e[x] +
                    p->right->area * p->right->e[x]) /
                    (p->left->area + p->right->area);
            }
        }
    }
}

/**/
/* Complete the answer by propagating partial dot-product sums down the
/* hierarchy from the root.
/**/
void
Prop(Node *p, int b)
{
    Node    *sstak[128],
            **ssp = sstak;
    char    pstak[128],
            *csp = pstak,
            c;
    float   temp;

    if (!p)
        return;

    /* Collapse vector elements upwards. */
    *ssp++ = p;
    *csp++ = 0;
    while (ssp != sstak) {
        p = *--ssp;
        c = *--csp;
        if (c == 0) {
            p->e[b] = p->t;
            if (p->parent)
                p->e[b] += p->parent->e[b];
        }
        if (p->left) {
            if (c == 0) { /* Left subtree not visited. */
                ssp++;
                *csp++ = 1;
                *ssp++ = p->left;
                *csp++ = 0;
            }
            else if (c == 1) { /* Right subtree not visited. */
                ssp++;

```

```

        *csp++ = 2;
        *ssp++ = p->right;
        *csp++ = 0;
    }
}

void
Propup(Node *p, int b)
{
    Node    *sstak[128],
            **ssp = sstak;
    char    pstak[128],
            *csp = pstak,
            c;

    if (!p)
        return;

    /* Collapse vector elements upwards. */
    *ssp++ = p;
    *csp++ = 0;
    while (ssp != sstak) {
        p = *--ssp;
        c = *--csp;
        if (c == 0)
            p->e[b] = p->t;
        if (p->left) {
            if (c == 0) { /* Left subtree not visited. */
                ssp++;
                *csp++ = 1;
                *ssp++ = p->left;
                *csp++ = 0;
            }
            else if (c == 1) { /* Right subtree not visited. */
                ssp++;
                *csp++ = 2;
                *ssp++ = p->right;
                *csp++ = 0;
            }
        }
        else { /* Visit this node. */
            p->e[b] = p->left->e[b] + p->right->e[b];
        }
    }
}

/*****/
/* All the following vector operations take slot numbers as their arguments.
/*****/

/**/
/* Hierarchical dest = a + b.
/**/
void
Hadd(Node *nd, int dest, int a, int b)
{
    if (!nd)
        return;
    if (!nd->left)
        nd->e[dest] = nd->e[a] + nd->e[b];
    Hadd(nd->left, dest, a, b);
    Hadd(nd->right, dest, a, b);
}

/**/
/* Hierarchical dest = a - b.

```

```

/**/
void
Hsub(Node *nd, int dest, int a, int b)
{
    if (!nd)
        return;
    if (!nd->left)
        nd->e[dest] = nd->e[a] - nd->e[b];
    Hsub(nd->left, dest, a, b);
    Hsub(nd->right, dest, a, b);
}

/**/
/* Hierarchical dest = -a.
/**/
void
Hneg(Node *nd, int dest, int a)
{
    if (!nd)
        return;
    if (!nd->left)
        nd->e[dest] = - (nd->e[a]);
    Hneg(nd->left, dest, a);
    Hneg(nd->right, dest, a);
}

/**/
/* Hierarchical dest = a * b.
/**/
void
Hmult(Node *nd, int dest, int a, int b)
{
    if (!nd)
        return;
    if (!nd->left)
        nd->e[dest] = nd->e[a] * nd->e[b];
    Hmult(nd->left, dest, a, b);
    Hmult(nd->right, dest, a, b);
}

/**/
/* Hierarchical dest = a * s.
/**/
void
Hscale(Node *nd, int dest, int a, float s)
{
    if (!nd)
        return;
    if (!nd->left)
        nd->e[dest] = nd->e[a] * s;
    Hscale(nd->left, dest, a, s);
    Hscale(nd->right, dest, a, s);
}

/**/
/* Hierarchical dot product of a and b.
/**/
float
Hdot(Node *nd, int a, int b)
{
    /* If called with NULL pointer, just return 0. */
    if (!nd)
        return 0.0;
    return (nd->left)
        ? (Hdot(nd->left, a, b) + Hdot(nd->right, a, b))
        : (nd->e[a] * nd->e[b]);
}

```

```

/**/
/* Hierarchical dest = 1.0 / a.
/**/
void
Hinvert(Node *nd, int dest, int a)
{
    if (!nd)
        return;
    if (!nd->left && nd->e[a] != 0.0)
        nd->e[dest] = 1.0 / nd->e[a];
    Hinvert(nd->left, dest, a);
    Hinvert(nd->right, dest, a);
}

/**/
/* Vector d = vector s.
/**/
void
Hcopy(Node *nd, int d, int s)
{
    if (!nd)
        return;
    if (!nd->left)
        nd->e[d] = nd->e[s];
    Hcopy(nd->left, d, s);
    Hcopy(nd->right, d, s);
}

/**/
/* Vector d = constant.
/**/
void
Hfill(Node *nd, int d, float s)
{
    if (!nd)
        return;
    if (!nd->left)
        nd->e[d] = s;
    Hfill(nd->left, d, s);
    Hfill(nd->right, d, s);
}

/**/
/* Vector one norm.
/**/
float
Hnorm(Node *nd, int d)
{
    if (!nd)
        return 0.0;
    return (nd->left ? 0.0 : fabs((double) nd->e[d])) +
        Hnorm(nd->left, d) +
        Hnorm(nd->right, d);
}

/**/
/* Vector infinity norm.
/**/
float
Hinfnorm(Node *nd, int d)
{
    float mleft, mright;

    if (!nd)
        return 0.0;

    if (nd->left) {
        mleft = Hinfnorm(nd->left, d);

```

```

        mright = Hinfnorm(nd->right, d);
        return Fmax(mleft, mright);
    }
    else
        return fabs((double) nd->e[d]);
}

void
Hmma (Node *p, int x, int rowsums, int rho, int emiss)
{
    Node    *sstak[128],
            **ssp = sstak;

    if (!p)
        return;

    /* Collapse vector elements upwards. */
    *ssp++ = p;
    while (ssp != sstak) {
        p = *--ssp;
        if (!p->left)
            p->e[x] = p->e[x] * p->e[rowsums] * p->e[rho] + p->e[emiss];
        if (p->left) {
            *ssp++ = p->left;
            *ssp++ = p->right;
        }
    }
}

void
Hmsmsmn (Node *p, float *resid, int rowsums, int tmp, int x,
         int emiss, int rhoinv, int area)
{
    Node    *sstak[128],
            **ssp = sstak;
    float   t;

    if (!p)
        return;

    /* Collapse vector elements upwards. */
    *resid = 0.0;
    *ssp++ = p;
    while (ssp != sstak) {
        p = *--ssp;
        if (!p->left) {
            t = p->e[area] * (p->e[rhoinv] *
                (p->e[x] - p->e[emiss]) - p->e[rowsums] * p->e[tmp]);
            *resid = Fmax(*resid, fabs((double) t));
        }
        if (p->left) {
            *ssp++ = p->left;
            *ssp++ = p->right;
        }
    }
}

/**/
/* Get Area vector.  Set each element of the hierarchical vector to the
/* geometrical area of the node it is associated with.
/**/
void
Hgetarea(Node *nd, int d)
{
    if (!nd)
        return;
    if (!nd->left)
        nd->e[d] = nd->area;
}

```

```

    if (nd->left) {
        Hgetarea(nd->left , d);
        Hgetarea(nd->right, d);
    }
}

/**/
/* Pretty-print a vector.
/**/
void
Hprint(Node *nd, int d)
{
    /* Check for null pointer. */
    if (!nd)
        return;

    /* Recur if body node, print if leaf node. */
    if (nd->left) {
        Hprint(nd->left , d);
        Hprint(nd->right, d);
    }
    else
        printf("%7g ", nd->e[d]);

    /* Newline if done with root node. */
    if (!nd->parent)
        printf("\n");
}

/**/
/* Pretty print the current patch hierarchically. If the verbose flag
/* is greater than zero, the patch geometry is printed in addition to
/* the patch number. If the verbose flag is greater than one, then the
/* patch center is also printed.
/**/
void
Nodeprint(Node *nd, int verbosity, int level)
{
    int          i, mask;

    if (!nd)
        return;

    Indent(level);
    printf("Node #%d:\n", nd->id);

    if (verbosity > 0) {
        if (nd->id <= 0) {
            Indent(level);
            printf(" Bbox   = [%g - %g][%g - %g][%g - %g]\n",
                nd->vertex[0][0], nd->vertex[1][0],
                nd->vertex[0][1], nd->vertex[1][1],
                nd->vertex[0][2], nd->vertex[1][2]);
        }
        Indent(level);
        printf(" Center = ");
        Vprint(nd->center);
        printf("\n");
        Indent(level);
        printf(" Normal = ");
        Vprint(nd->normal);
        printf("\n");
        Indent(level);
        printf(" Area   = %g\n", nd->area);
    }
    if (verbosity > 1) {
        Indent(level);
        printf(" Vector elements: ");
    }
}

```

```

        for (i = 0 ; i < maxhiervec ; i++) {
            mask = (1 << i);
            if (vecalloc & mask)
                printf("%d:%f ", i, nd->e[i]);
        }
        printf("\n");
    }

    if (nd->left)
        Nodeprint(nd->left , verbosity, level + 1);
    if (nd->right)
        Nodeprint(nd->right, verbosity, level + 1);
}

/**/
/* Convenience routine to indent some number of spaces.
/**/
static void
Indent(int level)
{
    while (level-- > 0)
        printf(" ");
}

/*****
/* Polygon Functions */
*****/

/**/
/* Constructor from four vertices.
/**/
void
Makepoly(Node *nd, Vector v0, Vector v1, Vector v2, Vector v3)
{
    int          i0, i1;
    float        tmp;
    Vector        vtmp1, vtmp2, edge[4];

    /* Install vertices, construct the center, and set the normal vector. */
    Vcopy (nd->vertex[0], v0);
    Vcopy (nd->vertex[1], v1);
    Vcopy (nd->vertex[2], v2);
    Vcopy (nd->vertex[3], v3);
    Vdiff (edge[0],    v1,    v0);
    Vdiff (edge[1],    v2,    v1);
    Vdiff (edge[2],    v3,    v2);
    Vdiff (edge[3],    v0,    v3);
    Vdiff (vtmp1,      v2,    v0);
    Vdiff (vtmp2,      v3,    v1);
    Vcross (nd->normal, vtmp1, vtmp2);
    Vscale (nd->normal, nd->normal, 0.5);
    Vsum   (nd->center, v0,    v1);
    Vsum   (nd->center, nd->center, v2);
    Vsum   (nd->center, nd->center, v3);
    Vscale (nd->center, nd->center, 0.25);
    nd->area = Vmag(nd->normal);

    /* Now make sure that the polygon is convex and planar. */
    for (i0 = 0 ; i0 < 4 ; i0++) {
        i1 = (i0 + 1) & 3;
        Vcross(vtmp1, edge[i0], edge[i1]);
        tmp = fabs(Vdot(vtmp1, nd->normal) - (nd->area * Vmag(vtmp1)));
        if (tmp > eps * nd->area * Vmag(vtmp1)) {
            Vzero(nd->vertex[0]);
            Vzero(nd->vertex[1]);
            Vzero(nd->vertex[2]);
            Vzero(nd->vertex[3]);
            Vzero(nd->center);
        }
    }
}

```



```

        Vzero(nd->normal);
        nd->area = 0.0;
        return;
    }
}

/**/
/* Initialize this node and its subtree. Initialization consists of setting
/* normal vectors, and in the case of composite nodes, the center as well.
/**/
void
Nodeinit(Node *nd)
{
    Vector vtmp1, vtmp2;
    Vector lbmin, lbmax,
           rbmin, rbmax;
    Node *p, *q;
    float tmp;
    int i, j;

    /* Set center and normal vectors for a polygon. */
    if (nd->id > 0) {
        Vsum (nd->center, nd->vertex[0], nd->vertex[1]);
        Vsum (nd->center, nd->center, nd->vertex[2]);
        Vsum (nd->center, nd->center, nd->vertex[3]);
        Vscale (nd->center, nd->center, 0.25);
        Vdiff (vtmp1, nd->vertex[2], nd->vertex[0]);
        Vdiff (vtmp2, nd->vertex[3], nd->vertex[1]);
        Vcross (nd->normal, vtmp1, vtmp2);
        Vscale (nd->normal, nd->normal, 0.5);
        nd->area = Vmag(nd->normal);
    }

    /* Recur for all daughters. */
    if (nd->left)
        Nodeinit(nd->left);
    if (nd->right)
        Nodeinit(nd->right);

    /* Set area, center, normal, bbox for composites. */
    if (nd->id <= 0) {
        /* Set area. */
        nd->area = nd->left->area + nd->right->area;

        /* Set center. */
        Vscale(vtmp1, nd->left->center, nd->left->area);
        Vscale(vtmp2, nd->right->center, nd->right->area);
        Vsum (nd->center, vtmp1, vtmp2);
        Vscale(nd->center, nd->center,
              1.0 / (nd->left->area + nd->right->area));

        /* Set normal. */
        Vsum (nd->normal, nd->left->normal, nd->right->normal);
        tmp = Vmag(nd->normal);
        if (tmp != 0.0)
            Vscale (nd->normal, nd->normal, nd->area / tmp);

        /* Set bounding box. */
        /* Extract the bounding boxes for the left and right daughters. */
        p = nd->left;
        q = nd->right;
        if (nd->left->id <= 0) {
            Vcopy(lbmin, p->vertex[0]);
            Vcopy(lbmax, p->vertex[1]);
        }
        else {
            /* Get the bounding box around p and q. */

```

```

    for (j = 0 ; j < 3 ; j++) {
        lbmin[j] = p->vertex[0][j];
        lbmax[j] = p->vertex[0][j];
        for (i = 1 ; i < 4 ; i++) {
            lbmin[j] = Fmin(lbmin[j], p->vertex[i][j]);
            lbmax[j] = Fmax(lbmax[j], p->vertex[i][j]);
        }
    }
}
if (nd->right->id <= 0) {
    Vcopy(rbmin, q->vertex[0]);
    Vcopy(rbmax, q->vertex[1]);
}
else {
    /* Get the bounding box around p and q. */
    for (j = 0 ; j < 3 ; j++) {
        rbmin[j] = q->vertex[0][j];
        rbmax[j] = q->vertex[0][j];
        for (i = 1 ; i < 4 ; i++) {
            rbmin[j] = Fmin(rbmin[j], q->vertex[i][j]);
            rbmax[j] = Fmax(rbmax[j], q->vertex[i][j]);
        }
    }
}

/* Merge the left and right bounding boxes. */
for (j = 0 ; j < 3 ; j++) {
    nd->vertex[0][j] = Fmin(lbmin[j], rbmin[j]);
    nd->vertex[1][j] = Fmax(lbmax[j], rbmax[j]);
}
}

/**/
/* Return number of ACTIVE leaves in hierarchy.
/**/
int
Numleaves(Node *nd)
{
    return nd->left ? Numleaves(nd->left) + Numleaves(nd->right) : 1;
}

/**/
/* Use a bounding box check to see if anything is between p and q.
/**/
Visible
Occlusion (Node *O, Node *P, Node *Q, double *work)
{
    Vector      r, s, t, u,
               vtmp1,
               p[4], q[4], o[4];
    Visible
    float      vis1, vis2;
    float      tmp,           /* Scratch scalar */
               pqbox[6];     /* Bbox about polygons p and q */
    int        i, j, k, v,   /* Loop counters */
               k1, k2, k3, k4, /* Hit-miss counters */
               ip, iq,      /* Hull plane counters */
               ip0, iq0,
               incflag;
    static Vector hullnm[8],
                  hullpt[8];
    static int    ih,
                  hullp[8], /* Hull edge vertex index. */
                  hullq[8]; /* Hull edge vertex index. */
    static Node   *lastP = NULL,
                  *lastQ = NULL;
}

```

```

/* Copy p, q, and o vertices into convenience variables. */
for (i = 0 ; i < 4 ; i++) {
    Vcopy(o[i], O->vertex[i]);
    Vcopy(p[i], P->vertex[i]);
    Vcopy(q[i], Q->vertex[i]);
}

/**/
/* PQ VISIBILITY AND SUPPORT PLANE SPLITTING TEST */
/* Find the number of vertices of p in q's half plane, and vice */
/* versa. */
/* If p behind q or q behind p, then they do not see each other. */
/* If p straddles q or q straddles p, they may be partly visible. */
/**/
k1 = k2 = k3 = k4 = 0;
for (i = 0 ; i < 4 ; i++) {
    Vdiff(r, q[i], p[i]);
    tmp = Vdot(r, P->normal);
    if (tmp > eps)
        k1++;
    if (tmp < -eps)
        k2++;
    tmp = Vdot(r, Q->normal);
    if (tmp < -eps)
        k3++;
    if (tmp > eps)
        k4++;
}
(*work) += 68;
/* At least one polygon can't see the other. */
if (k1 == 0 || k3 == 0)
    return blocked;
/* One polygon splits the other's support plane. */
if (k2 > 0 || k4 > 0)
    return partial;

/**/
/* Test for a COMPOSITE between two polygons. */
/**/
if (O->id <= 0) {
    /* Get the bounding box around p and q. */
    for (i = j = 0 ; i < 6 ; i += 2, j++) {
        pqbox[i+0] = Fmin(P->vertex[0][j], Q->vertex[0][j]);
        pqbox[i+1] = Fmax(P->vertex[0][j], Q->vertex[0][j]);
        for (v = 1 ; v < 4 ; v++) {
            pqbox[i+0] = Fmin(pqbox[i+0],
                Fmin(P->vertex[v][j], Q->vertex[v][j]));
            pqbox[i+1] = Fmax(pqbox[i+1],
                Fmax(P->vertex[v][j], Q->vertex[v][j]));
        }
    }

    /* Test if this polygon's bbox lies completely to one side of */
    /* pq's bbox. If so, then there is surely no occlusion. */
    if ((O->vertex[0][0] < pqbox[0] && O->vertex[1][0] < pqbox[0] ||
        O->vertex[0][0] > pqbox[1] && O->vertex[1][0] > pqbox[1]) ||
        (O->vertex[0][1] < pqbox[2] && O->vertex[1][1] < pqbox[2] ||
        O->vertex[0][1] > pqbox[3] && O->vertex[1][1] > pqbox[3]) ||
        (O->vertex[0][2] < pqbox[4] && O->vertex[1][2] < pqbox[4] ||
        O->vertex[0][2] > pqbox[5] && O->vertex[1][2] > pqbox[5])) {
        if (debug & isect)
            printf("Occlusion: Polygon outside bounding box.\n", O->id);
        return visible;
    }
}

vis1 = Occlusion(O->left, P, Q, work);
if (debug & isect) {

```

```

printf("Occlusion(%d): Left returned %s\n", O->id,
      ( vis1 == partial) ? "partial" :
      ((vis1 == visible) ? "visible" : "blocked"));
}
if (vis1 == blocked)
  return vis1;
vis2 = Occlusion(O->right, P, Q, work);
if (debug & isect) {
  printf("Occlusion(%d): Right returned %s\n", O->id,
        ( vis2 == partial) ? "partial" :
        ((vis2 == visible) ? "visible" : "blocked"));
}
if (vis2 == blocked)
  return vis2;
if (vis1 == partial || vis2 == partial)
  return partial;
return visible;
}
/**/
/* Test for a POLYGON between two polygons. */
/**/
else {
  /**/
  /* ENDCAP TEST */
  /* Check if o lies at least partly in p and q half planes. Return 2 */
  /* if all vertices of o are behind p or behind q, or if p and q are */
  /* on the same side of o. */
  /**/

  k1 = k2 = 0;
  for (i = 0 ; i < 4 ; i++) {
    Vdiff(vtmp1, o[i], p[i]);
    tmp = Vdot(vtmp1, P->normal);
    if (tmp > eps) /* o vertex is strictly in front of p. */
      k1++;
    Vdiff(vtmp1, o[i], q[i]);
    tmp = Vdot(vtmp1, Q->normal);
    if (tmp > eps) /* o vertex is strictly in front of q. */
      k2++;
  }
  (*work) += 72;
  if (k1 == 0 || k2 == 0) /* No vertices are strictly in front */
    return visible; /* of the p and q planes. */

  k1 = k2 = k3 = k4 = 0;
  for (i = 0 ; i < 4 ; i++) {
    Vdiff(vtmp1, p[i], o[i]);
    tmp = Vdot(vtmp1, O->normal);
    if (tmp > -eps) /* p vertex is on or in front of o. */
      k1++;
    if (tmp < eps) /* p vertex is on or behind o. */
      k2++;
    Vdiff(vtmp1, q[i], o[i]);
    tmp = Vdot(vtmp1, O->normal);
    if (tmp > -eps) /* q vertex is on or in front of o. */
      k3++;
    if (tmp < eps) /* q vertex is on or behind o. */
      k4++;
  }
  (*work) += 80;
  if ((k1 == 4 && k3 == 4) || (k2 == 4 && k4 == 4))
    return visible;

  /**/
  /* WAIST PLANE CONSTRUCTION */
  /* Finally, check whether o lies in the half planes that define the */
  /* "waist" of the convex hull of p and q. Form the 8 planes defined */
  /* by an edge of p with the "rearmost" vertex of q and vice versa, */

```

```

/* storing them as point-normal pairs. Test planes against all 4 */
/* points of o. If o lies strictly behind any half plane, return 2. */
/* If o lies strictly within at least one (open) half plane, */
/* return 1. */
/* If o contains every intersection of hull with o plane, return 0.*/
/**/
if (P != lastP || Q != lastQ) {
    lastP = P;
    lastQ = Q;

    ip = ip0 = 0;
    Vdiff(s, p[(ip+1)&3], p[ip]); /* Form directed edge of patch p.*/
    for (i = 0 ; i < 4 ; i++) {
        Vdiff(r, q[i], p[ip]); /* Vector from anchor point on p */
                                /* to trial point on q. */
        Vcross(u, r, s);
        if (Vmagsq(u) < eps * eps)
            Vcopy(u, P->normal);

        /* Now check that all other points on q */
        /* lie in front of this plane. */
        for (k = 0 ; k < 4 ; k++) {
            if (k == i)
                continue;
            Vdiff(vtmp1, q[k], q[i]);
            tmp = Vdot(vtmp1, u); /* Dot with vector from ref */
                                /* point to test vertex */
            (*work) += 9;
            /* If vertex lies behind plane, end loop. */
            if (tmp < -eps)
                break;
        }
        (*work) += 19;

        if (k == 4) {
            iq = iq0 = i;
            break;
        }
    }
    (*work) += 3;

    ih = 0;
    incflag = 0;
    do {

        /* Find out which vertex can be incremented. */
        /* First, try incrementing ip. */
        Vdiff(s, p[(ip+1)&3], p[ip]); /* Vector along edge on p from*/
                                    /* vertex i to vtx i+1 mod 4. */
        Vdiff(r, q[iq], p[ip]); /* Vector from trial pt on q */
                                /* to head point of edge on p.*/
        Vcross(u, r, s); /* Normal to plane. r X s. */
        if (Vmagsq(u) < eps * eps) /* Check if p[ip] and q[iq] */
                                    /* are identical. */
            (*work) += 22;

        /* Now check that all other points on q */
        /* lie in front of this plane. */
        for (k = 0 ; k < 4 ; k++) {
            if (k == iq)
                continue;
            Vdiff(vtmp1, q[k], q[iq]);
            tmp = Vdot(vtmp1, u); /* Dot with vector from ref */
                                /* point to test vertex */
            (*work) += 9;
            if (tmp < -eps) /* If vertex lies behind plane*/
                                /* end loop. */
                break;
        }
    }
}

```

```

/* If all points in q lie behind the plane, accept ip+1. */
if (k == 4) {
    /* Store a line segment from vertex ip to iq. */
    hullp [ih] = ip;
    hullq [ih] = iq;
    Vcopy(hullnm[ih], u);
    Vcopy(hullpt[ih], p[ip]);
    ip = (ip + 1) & 3;
    incflag = 0;
    ih++;
    continue;
}

/* Next, try decrementing iq. */
Vdiff(s, q[(iq-1)&3], q[iq]); /* Vector along edge on q from*/
/* vertex i to vertex i-1mod4.*/
Vcross(u, r, s); /* Normal to plane. r X s. */
if (VmagSq(u) < eps * eps)
    Vcopy(u, Q->normal);
(*work) += 19;

/* Now check that all other points on p */
/* lie in front of this plane. */
for (k = 0 ; k < 4 ; k++) {
    if (k == ip)
        continue;
    Vdiff(vtmp1, p[k], p[ip]);
    tmp = Vdot(vtmp1, u); /* Dot with vector from ref */
/* point to test vertex */
    (*work) += 9;
    if (tmp < -eps) /* If vertex lies behind plane*/
        break; /* end loop. */
}

/* If all points in p lie behind the plane, accept iq-1. */
if (k == 4) {
    /* Store a line segment from vertex ip to iq. */
    hullp [ih] = ip;
    hullq [ih] = iq;
    Vcopy(hullnm[ih], u);
    Vcopy(hullpt[ih], p[ip]);
    iq = (iq - 1) & 3;
    incflag = 1;
    ih++;
    continue;
}

/* Disaster if we get to here. */
if (incflag == 0) {
    incflag = 1;
    iq = (iq - 1) & 3;
}
else {
    incflag = 0;
    ip = (ip + 1) & 3;
}
} while ((ip != ip0 || iq != iq0) && ih < 8);
}

/**/
/* WAIST PLANE TESTS.
/* Now, actually check the vertices of
/* o against the waist planes of pq.
/**/
for (i = 0, k2 = 0 ; i < ih ; i++) {
    for (j = 0, k1 = 0 ; j < 4 ; j++) {

```

```

        /* Bump k1 if point j lies on the outside the half space. */
        Vdiff(vtmp1, o[j], hullpt[i]);
        tmp = Vdot(vtmp1, hullnm[i]);
        if (tmp < eps)
            k1++;
    }
    (*work) += 36;
    if (k1 == 4) /* o lies completely behind hull plane. */
        return visible;
    if (k1 == 0) /* o lies completely inside hull plane. */
        k2++;
}
if (k2 == ih) /* o is strictly in front of every hull plane. */
    return partial;

/* Test if the intersection of waist and support plane of o lies */
/* completely inside o. If so, then return total occlusion. */
for (i = 0 ; i < ih ; i++) {
    /* Vector along the i'th "waistline". */
    Vdiff(r, q[hullq[i]], p[hullp[i]]);

    /* Intersect the vector with o support plane. */
    tmp = Vdot(r, O->normal);
    (*work) += 8;
    if (tmp != 0.0) { /* SHOULD THIS BE EPSILON? */
        Vdiff(vtmp1, o[0], p[hullp[i]]);
        tmp = Vdot(vtmp1, O->normal) / tmp;
        Vscale(s, r, tmp);
        Vsum(s, s, p[hullp[i]]);
        (*work) += 18;
    }
    else {
        Vcopy(s, p[hullp[i]]);
        break; /* FIX FOR ONORML PERP. TO WAISTLINE */
    }
}

/* Check if the IP lies inside o. */
for (j = 0 ; j < 4 ; j++) {
    Vdiff(r, o[(j+1)&3], o[j]); /* Construct an edge of o. */
    Vdiff(t, s, o[j]); /* Vector from o[j] to IP */
    Vcross(u, r, t); /* u is perpendicular to r and t */

    /* Check the sign of the dot product of u with onormal. */
    tmp = Vdot(O->normal, u);
    (*work) += 21;
    if (tmp < -eps)
        break;
}

/* If this point does not lie inside o, then quit looking. */
if (j != 4)
    break;
}

/* If all points lie inside o, then return blocked. */
if (i == ih)
    return blocked;

return partial;
}

/**/
/* Subdivide a patch into two subpatches. Split patch by halving the
/* longest side, and the side opposite the longest side. This function
/* always succeeds and returns TRUE.
/**/
void

```

```

Subdiv(Node *nd)
{
    Vector      edge[4],
               nvertex0,
               nvertex1;
    float      len[4],
               temp;
    int        i,
               longest,
               opposite;

    /* Return if patch has already been subdivided. */
    if (nd->left)
        return;

    /* Allocate the new daughter patches. */
    nd->left = Nodealloc(patch);
    nd->right = Nodealloc(patch);
    Nodecopy(nd->left, nd);
    Nodecopy(nd->right, nd);
    nd->left->parent = nd;
    nd->right->parent = nd;

    /* Compute the edge vectors. */
    Vdiff(edge[0], nd->vertex[1], nd->vertex[0]);
    Vdiff(edge[1], nd->vertex[2], nd->vertex[1]);
    Vdiff(edge[2], nd->vertex[3], nd->vertex[2]);
    Vdiff(edge[3], nd->vertex[0], nd->vertex[3]);
    len[0] = Vmagsq(edge[0]);
    len[1] = Vmagsq(edge[1]);
    len[2] = Vmagsq(edge[2]);
    len[3] = Vmagsq(edge[3]);

    /* Find the longest side. */
    longest = 0;
    temp = len[0];
    for (i = 1 ; i < 4 ; i++) {
        if (len[i] > temp) {
            temp = len[i];
            longest = i;
        }
    }
    opposite = (longest + 2) & 3;

    /* Split the longest side, and the side opposite it. */
    Vscale(nvertex0, edge[longest], 0.5);
    Vsum (nvertex0, nvertex0, nd->vertex[longest]);
    Vscale(nvertex1, edge[opposite], 0.5);
    Vsum (nvertex1, nvertex1, nd->vertex[opposite]);
    Makepoly(nd->left, nd->vertex[longest], nvertex0, nvertex1,
             nd->vertex[(longest + 3) & 3]);
    Makepoly(nd->right, nd->vertex[opposite], nvertex1, nvertex0,
             nd->vertex[(opposite + 3) & 3]);
}

/*****
/* Composite Functions */
*****/

/**/
/* Constructor from two other nodes.
/**/
void
Makecomp(Node *nd, Node *p, Node *q)
{
    int      i, j;
    Vector   lbmin, lbmax,
             rbmin, rbmax,

```



```

        vtmp1, vtmp2;

/* Link p and q to nd. */
nd->left = p;
nd->right = q;
p->parent = nd;
q->parent = nd;

/* Update the area and center of the composite. */
nd->area = p->area + q->area;
Vscale(vtmp1, p->center, p->area);
Vscale(vtmp2, q->center, q->area);
Vsum (nd->center, vtmp1, vtmp2);
Vscale(nd->center, nd->center, 1.0 / (p->area + q->area));

/* Extract the bounding boxes for the left and right daughters. */
if (nd->left->id <= 0) {
    Vcopy(lbmin, p->vertex[0]);
    Vcopy(lbmax, p->vertex[1]);
}
else {
    /* Get the bounding box around p and q. */
    for (j = 0 ; j < 3 ; j++) {
        lbmin[j] = p->vertex[0][j];
        lbmax[j] = p->vertex[0][j];
        for (i = 1 ; i < 4 ; i++) {
            lbmin[j] = Fmin(lbmin[j], p->vertex[i][j]);
            lbmax[j] = Fmax(lbmax[j], p->vertex[i][j]);
        }
    }
}
if (nd->right->id <= 0) {
    Vcopy(rbmin, q->vertex[0]);
    Vcopy(rbmax, q->vertex[1]);
}
else {
    /* Get the bounding box around p and q. */
    for (j = 0 ; j < 3 ; j++) {
        rbmin[j] = q->vertex[0][j];
        rbmax[j] = q->vertex[0][j];
        for (i = 1 ; i < 4 ; i++) {
            rbmin[j] = Fmin(rbmin[j], q->vertex[i][j]);
            rbmax[j] = Fmax(rbmax[j], q->vertex[i][j]);
        }
    }
}

/* Merge the left and right bounding boxes. */
for (j = 0 ; j < 3 ; j++) {
    nd->vertex[0][j] = Fmin(lbmin[j], rbmin[j]);
    nd->vertex[1][j] = Fmax(lbmax[j], rbmax[j]);
}
}

/**/
/* Return this Node's level in hierarchy.
/**/
int
Getlevel(Node *p)
{
    return p->parent ? Getlevel(p->parent) + 1 : 0;
}

/* Return the number of nodes in
/* the patch hierarchy starting
/* here and going down.
*/
int
Numelem(Node *p)

```

```
{  
  return p ? Numelem(p->left) + Numelem(p->right) + 1 : 0;  
}
```

A.6 Source file heap.c

```

/**/
/* File: heap.c
/* SLALOM 93 adaptive patch subdivision testbed program.
/* Queue and Heap function definitions.
/* Version: %G% %W%
/**/

#include          "slal.h"
#include          "proto.h"

/*****/
/* Queue Functions */
/*****/

/**/
/* Constructor. Initialize to zero length and allocate no storage.
/**/
LinkQueue *
Lqalloc(void)
{
    LinkQueue *lq;
    lq = (LinkQueue *) malloc(sizeof(LinkQueue));
    lq->alloclen = lq->head = lq->tail = 0;
    lq->p = NULL;
    return lq;
}

/**/
/* Destructor. Free up any allocated storage.
/**/
void
Lqfree(LinkQueue *lq)
{
    if (lq->p)
        free(lq->p);
    if (lq)
        free(lq);
}

/* Return the number of Links in Q */
int
Lqlength(LinkQueue *lq)
{
    return (lq->head - lq->tail + lq->alloclen) % lq->alloclen;
}

/**/
/* Enqueue the element lnk at the head of the queue.
/**/
void
Lqenqueue(LinkQueue *lq, Link lnk)
{
    /* Check for overflow. */
    if (!lq->p ||
        lq->head == ((lq->tail - 1 + lq->alloclen) % lq->alloclen))
        Lqextend(lq, 0);

    /* Insert the new element. */
    lq->p[lq->head] = lnk;

    /* Bump the head index. */
    lq->head++;
    if (lq->head >= lq->alloclen)

```

```

        lq->head = 0;
    }

/**/
/* Dequeue an element from the tail of the queue.
/**/
Link
Lqdequeue(LinkQueue *lq)
{
    Link lnk;

    /* Check for underflow. */
    if (lq->head == lq->tail) {
        fprintf(stderr, "LinkQueue: Queue underflow.\n");
        lnk.p = lnk.q = NULL;
        lnk.cpq = lnk.epq = 0.0;
        return lnk;
    }

    /* Extract the element. */
    lnk = lq->p[lq->tail];

    /* Bump the tail index. */
    lq->tail++;
    if (lq->tail >= lq->alloclen)
        lq->tail = 0;

    return lnk;
}

/**/
/* Allocate another unit of "defchunksize" queue elements and copy the old
/* elements in. This is rather inefficient, so it deserves future attention.
/**/
void
Lqextend(LinkQueue *lq, int newsize)
{
    Link      *newp;
    int       i;

    /* Check argument passed. If zero, then extend one chunksize. */
    if (newsize == 0 || newsize < lq->alloclen)
        newsize = lq->alloclen + defchunksize;

    /* Allocate the new space. */
    newp = (Link *) malloc(newsize * sizeof(Link));

    /* Copy old queue contents into new queue. */
    if (lq->tail < lq->head) {
        for (i = lq->tail ; i < lq->head ; i++)
            newp[i-lq->tail] = lq->p[i];
        lq->head = lq->head - lq->tail;
        lq->tail = 0;
    }
    else if (lq->alloclen > 0) {
        for (i = lq->tail ; i < lq->alloclen ; i++)
            newp[i-lq->tail] = lq->p[i];
        for (i = 0 ; i < lq->head ; i++)
            newp[lq->alloclen-lq->tail+i] = lq->p[i];
        lq->head = ((lq->head - lq->tail + lq->alloclen) % lq->alloclen);
        lq->tail = 0;
    }

    /* Free up the old queue storage. */
    if (lq->p)
        free(lq->p);

    /* Update queue management information. */

```

```

    lq->p = newp;
    lq->alloclen = newsize;
}

/**/
/* Pretty print the contents of the queue.
/**/
void
Lqprint(LinkQueue *lq)
{
    int    i;

    printf("Queue dump: ");
    for (i = lq->tail ; i != lq->head ; i = (i + 1) % lq->alloclen)
        printf("[p=%d q=%d cpq=%g epq=%g]\n",
            lq->p[i].p, lq->p[i].q, lq->p[i].cpq, lq->p[i].epq);
    printf("\n");
}

/*****
/* Heap Functions */
*****/

/**/
/* Constructor. Allocate no initial storage.
/**/
LinkHeap *
Lhalloc(void)
{
    LinkHeap *lh;

    lh = (LinkHeap *) malloc(sizeof(LinkHeap));
    lh->alloclen = lh->tail = 0;          /* Point one past last element. */
    lh->p = NULL;
    return lh;
}

/**/
/* Destructor.
/**/
void
Lhfree(LinkHeap *lh)
{
    if (lh->p)
        free(lh->p);
    if (lh)
        free(lh);
}

/**/
/* Add an element to the heap, then rebuild the heap.
/**/
void
Lhenqueue(LinkHeap *lh, Link lnk, int stats[])
{
    int    i, j;
    Link   temp;

    /* Update the stats structure, add element to the end of the heap, and */
    /* index to the new element and its parent. */
    stats[lnk.vis]++;
    /* Check for overflow. */
    if (lh->tail >= lh->alloclen)
        Lhextend(lh, 0);
    lh->p[lh->tail] = lnk;
    i = lh->tail++;
    j = (i - 1) >> 1;
}

```

```

/* While not at the root... */
while (i) {
    /* If the daughter is greater than the parent, swap them. */
    if (lh->p[i].epq > lh->p[j].epq) {
        temp = lh->p[j];
        lh->p[j] = lh->p[i];
        lh->p[i] = temp;
    }
    else
        break;

    /* If there was a swap, move a level up the heap, and repeat. */
    i = (i - 1) >> 1;
    j = (j - 1) >> 1;
}
}

/**/
/* Remove the root of the heap, and reheapify.
/**/
Link
Lhdequeue(LinkHeap *lh, int stats[])
{
    Link    lnk;

    /* Check for underflow. */
    if (lh->tail <= 0) {
        fprintf(stderr, "Lhdequeue(): Heap underflow.\n");
        lnk.p = lnk.q = NULL;
        lnk.cpq = lnk.epq = 0.0;
        return lnk;
    }

    /* Remove the root element, and replace it with the tail element. */
    lnk = lh->p[0];
    if (--lh->tail > 0)
        lh->p[0] = lh->p[lh->tail];

    /* Heapify from the root down since we replaced the root element. */
    Lhheapify(lh, 0);

    /* Update the stats structure. */
    stats[lnk.vis]--;
    return lnk;
}

/**/
/* Update the epq member of each element and reheapify.
/**/
void
Lhrebuild(LinkHeap *lh)
{
    int    i;

    for (i = 0 ; i < lh->tail ; i++)          /* Update the epq field. */
        Lupdate(&lh->p[i]);

    for (i = (lh->tail >> 1) - 1 ; i >= 0 ; i--) /* Build a new heap. */
        Lhheapify(lh, i);
}

void
Lupdate(Link *lnk)
{
    float s, t;

    if (lnk->p && lnk->q) {
        /**/

```

```

    /* Compute the 1-norm of the maximum amount of light
    /* emitted from patch p and reflected from patch q, or vice versa.
    /**/
    s = lnk->p->e[rhoelemR] * lnk->q->e[solelemR]
      + lnk->p->e[rhoelemG] * lnk->q->e[solelemG]
      + lnk->p->e[rhoelemB] * lnk->q->e[solelemB];
    t = lnk->q->e[rhoelemR] * lnk->p->e[solelemR]
      + lnk->q->e[rhoelemG] * lnk->p->e[solelemG]
      + lnk->q->e[rhoelemB] * lnk->p->e[solelemB];
    lnk->epq = Fmax(s, t) * lnk->err * 0.001;
  }
  else
    lnk->epq = 0.0;
}

/**/
/* Update the epq member of each element and reheapify.
/**/
void
Lhheapify(LinkHeap *lh, int root)
{
  int    i, j, k;
  Link   temp;

  i = root;
  j = (i << 1) + 1;
  k = j + 1;

  while (j < lh->tail) {
    if (lh->p[j].epq > lh->p[i].epq &&
        (k < lh->tail && lh->p[j].epq > lh->p[k].epq || k >= lh->tail)) {
      temp = lh->p[i];
      lh->p[i] = lh->p[j];
      lh->p[j] = temp;
      i = j;
    }
    else if (lh->p[k].epq > lh->p[i].epq && k < lh->tail) {
      temp = lh->p[i];
      lh->p[i] = lh->p[k];
      lh->p[k] = temp;
      i = k;
    }
    else
      break;
    j = (i << 1) + 1;
    k = j + 1;
  }
}

/**/
/* Clear all elements from the heap.
/**/
void
Lhclear(LinkHeap *lh)
{
  lh->tail = 0;
}

/**/
/* Allocate more space for the heap.
/**/
void
Lhextend(LinkHeap *lh, int newsz)
{
  Link    *newp;
  int     i;

  /* Check argument passed.  If zero, then extend one chunksize. */

```

```

if (newsize == 0 || newsize < lh->alloclen) {
    newsize = lh->alloclen + defchunksize;
}

/* Allocate a bigger storage area. */
newp = (Link *) malloc(newsize * sizeof(Link));

/* Copy the old elements to the new area. */
for (i = 0 ; i < lh->alloclen ; i++)
    newp[i] = lh->p[i];

/* Delete the old storage area. */
if (lh->p)
    free(lh->p);

/* Update heap management members. */
lh->p = newp;
lh->alloclen = newsize;
}

/**/
/* Pretty print the contents of the heap.
/**/
void
Lhprint(LinkHeap *lh)
{
    int    i;

    printf("Heap dump:\n");
    for (i = 0 ; i < lh->tail ; i++) {
        printf("[p=%4d q=%4d cpq=%10.4f epq=%10.4f]\n",
            lh->p[i].p->id, lh->p[i].q->id, lh->p[i].cpq, lh->p[i].epq);
    }
    printf("\n");
}

```


A.7 Source file matvec.c

```

/**/
/* File: matvec.c
/* 4x4 and 4x1 Matrix and vector manipulation and transformation code.
/**/

#include          <stdio.h>
#include          <math.h>
#include          "slal.h"
#include          "proto.h"

/**/
/* Vector Functions */
/**/

/* Zero a vector. */
void
Vzero(Vector v)
{
    v[0] = v[1] = v[2] = 0.0;
}

/* Copy a vector. */
#ifdef Vcopy
void
Vcopy(Vector dest, Vector a)
{
    dest[0] = a[0];
    dest[1] = a[1];
    dest[2] = a[2];
}
#endif

/* Pretty-print the vector. */
void
Vprint(Vector v)
{
    printf("[%g %g %g] ", v[0], v[1], v[2]);
}

/* Return the magnitude of a vector. */
float
Vmag(Vector v)
{
    return sqrt((double) Vdot(v, v));
}

/* Return the magnitude squared of a vector. */
#ifdef Vmagsq
float
Vmagsq(Vector v)
{
    return Vdot(v, v);
}
#endif

/* Add two vectors. */
#ifdef Vsum
void
Vsum(Vector dest, Vector a, Vector b)
{
    dest[0] = a[0] + b[0];
    dest[1] = a[1] + b[1];
    dest[2] = a[2] + b[2];
}
#endif

```

```

#endif

/* Subtract two vectors. */
#ifndef Vdiff
void
Vdiff(Vector dest, Vector a, Vector b)
{
    dest[0] = a[0] - b[0];
    dest[1] = a[1] - b[1];
    dest[2] = a[2] - b[2];
}
#endif

/* Scale vector by a scalar. */
void
Vscale(Vector dest, Vector a, float s)
{
    dest[0] = a[0] * s;
    dest[1] = a[1] * s;
    dest[2] = a[2] * s;
}

/* Vector cross product */
#ifndef Vcross
void
Vcross(Vector dest, Vector a, Vector b)
{
    Vector vtemp;

    vtemp[0] = (a[1] * b[2]) - (b[1] * a[2]);
    vtemp[1] = -((a[0] * b[2]) - (b[0] * a[2]));
    vtemp[2] = (a[0] * b[1]) - (b[0] * a[1]);
    Vcopy(dest, vtemp);
}
#endif

/* Vector dot product */
#ifndef Vdot
float
Vdot(Vector a, Vector b)
{
    return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
}
#endif

```