Parallel Radiosity Techniques for Mesh-Connected SIMD Computers

TR91-028

July, 1991

Amitabh Varshney

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

DARPA Contract #DAEA18-90-C
NSF Grant #CCR-8609588
ONR Grant #N00014-86-K-0680
Acquisition of the MP-1 was supported by grants from ONR, NIH Contract #CA 47982 and the State of North Carolina.
UNC is an Equal Opportunity/Affirmative Action Institution.

PARALLEL RADIOSITY TECHNIQUES FOR MESH-CONNECTED SIMD COMPUTERS

by

Amitabh Varshney

A Thesis submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill

1991

Approved by:

[illegible]

Frederick P. Brooks, Jr. Advisor

Jan F. Prins

Jan F. Prins Reader

Jennifer L. Welch

Jennifer L. Welch Reader

Amitabh Varshney

PARALLEL RADIOSITY TECHNIQUES FOR MESH-CONNECTED SIMD COMPUTERS

PARALLEL RADIOSITY TECHS FOR MESH SIMD COMPUTERS

Department of Computer Science

Master of Science

Total pages: 163

MS thesis

AMITABH VARSHNEY. Parallel Radiosity Techniques for Mesh-Connected SIMD Computers (Under the direction of Professor Frederick P. Brooks, Jr.)

Abstract

This thesis investigates parallel radiosity techniques for highly-parallel, mesh-connected SIMD computers. The approaches studies differ along the two orthogonal dimensions: the method of sampling - by ray-casting or by environment-projection - and the method of mapping of objects to processors - by object-space-based methods or by a balanced-load method. The environment-projection approach has been observed to perform better than the ray-casting approaches. For the dataset studied, the balanced-load method appears promising. Spatially subdividing the dataset without taking the potential light interactions into account has been observed to violate the locality property of radiosity. This suggests that object-space-based methods for radiosity must take visibility into account during subdivision to achieve any speedups based on exploiting the locality property of radiosity.

This thesis also investigates the reuse patterns of form-factors in perfectly diffuse environments during radiosity iterations. Results indicate that reuse is sparse even when significant convergence is achieved.

Implementations of these approaches have been done on a 4K processor MasPar MP-1.

To my parents,

Acknowledgments

I would like to thank my thesis advisor Dr. Frederick P. Brooks, Jr., for his insightful comments, constant encouragement and very helpful advice during all stages of this work. His systematic, devoted, and well-organized approach to work has been and shall remain a major source of inspiration to me.

I would also like to thank Dr. Jan F. Prins for his valuable comments regarding both the form and the content of this thesis. I value highly the encouragement that he has given me in this field since Fall 1990 (which was when I did a Highly Parallel Computing course offered by him).

Thanks are also due to Dr. Jennifer Welch for her detailed review of this thesis and for the several useful and valuable suggestions that she provided.

I am also grateful to Dr. William Wright for agreeing to act as a substitute member on my committee.

John Airey gave me the first tutorial on radiosity. He was quite helpful in getting me started on radiosity. We had several sessions on radiosity together and each time I came out having gained something useful. I thank him for all this.

Working with Howard Good for the Highly Parallel Computing class project was an enjoyable experience. The balanced-load ray-casting apporach presented in this thesis was implemented in that project. I shall remember the long hours we spent together working on making the rays intersect those virtual walls as one of the more memorable experiences at UNC. I also thank him for his valuable comments on this thesis.

I would like to thank all members of the Walkthrough team - John Alspaugh, Randy Brown, Curtis Hill, Yulan Wang, and Xialin Yuan - for their infectious enthusiasm and good humor that has kept up my spirits all along. John Alspaugh, in particular, provided me with the model of our office, Room 365 Sitterson Hall, on which I have compared the different radiosity approaches studied herein.

Manish Pandey and Goopeel Chung have been very helpful in my efforts to LaTeXize this document.

My parents have taught me the importance of hard work, patience, and perseverance though example. I affectionately dedicate this thesis to them.

Contents

List of Figures

Chapter 1

Overview and Results

This thesis research has been motivated by the current lengthy radiosity computation times in the UNC Walkthrough project and by a personal desire to better understand the process of mapping computationally intensive problems on to the newly emerging class of highly parallel machines. To place this thesis in its proper perspective, it would serve well to start with an overview of the Walkthrough project and the role of radiosity in it.

1.1 The Walkthrough Project

The UNC Walkthrough Project aims at development of a system for creating virtual building environments. This is intended to help architects and their clients to explore a proposed building design prior to its construction, correcting problems on the computer instead of in concrete [Brooks88], [Airey90b].

Walkthrough is in a class of virtual-worlds systems in which users can actually (though maybe only later) make comparisons with the real world, and thus directly verify the veracity of the simulation. With the reality serving as a touchstone, one of the primary goals of this project is to strive for realism. This realism is sought along four different dimensions - realistic images, real-time update rates, real models and an intuitive interface. Efforts along any of these dimensions, oppose those along the other dimensions, making this project a challenging one.

Realistic Images: The natural world is everywhere dense with complexity. The real man-made world however is less so. This realization has fostered the efforts to first tackle the problem of realistic rendering of the man-made objects. Using a global diffuse illumination model of radiosity, a respectable level of realism in virtual building models can be achieved. The radiosity method realistically simulates the interaction of light between diffuse surfaces. Walkthrough uses this to enhance visual realism. Another technique employed to increase realism is the use of procedural textures. Having textures for bricks, wood, ceiling tiles etc. adds a whole new level of detail to the images. A daylight model is used for displaying the effects of a diffused sunlight.

Real-Time Update Rates: Interactive update rates are crucial for the illusion of virtual building to work [Airey90b]. This however is at odds with the aim of realistic images outlined above. Using high-end graphics machines, such as Pixel-Planes, for rendering partially solves this problem. Pre-computation of potentially visible sets to restrict the number of polygons that have to be transformed and rendered for any viewpoint [Airey90a] further helps here.

Real Models: To study first-hand the problems and challenges of simulating real buildings, Walkthrough uses models built from the architectural drawings of actual buildings. This affords an opportunity to work on sufficiently complicated and detailed models and offers a benchmark (the actual building once it is built) with which to compare qualitatively the places where the virtual-building simulation succeeds and the places where it falls short. A commercial software package, AutoCAD, is used as the modeling tool. Being designed for architectural description, AutoCAD has some specialized facilities for model maintaining and model refining. Further, since many architects use it, its use offers opportunities for the exchange of building datasets.

Intuitive Interface: In any virtual-worlds system, the degree to which a user can interact naturally with the virtual world plays an important role in determining its effectiveness. Ideally, this man-machine interface should afford as easy and intuitive an interaction as with the real object. For interacting with Walkthrough, different users have differing preferences. For a client who does not have sufficient experience in visualization of the dimensions of the building from its blueprints, actual pacing about the virtual-building would be more appropriate. For an architect, who already has a good feel for the physical size of the building, an interface that would enable him to quickly fly-through the model would be better. Keeping these in mind, the Walkthrough interface has provisions for using head-mounted displays, a treadmill, joysticks, and a bicycle.

1.2 The Need for Parallel Radiosity

Radiosity is a global-illumination model for modeling the interaction of light between diffuse surfaces. Pioneered at Cornell University [Goral84], this method has gained widespread acceptance in the last few years for providing a diffuse lighting model for architectural datasets. The reasons behind this are twofold. Firstly, it realistically simulates such diffuse lighting effects as soft shadows and the diffuse inter-reflections typically observed inside buildings and secondly, since the illumination solution thus computed is view-independent, it allows interactive viewing of the dataset once the process has finished. The property that the solution computed is view-independent gives this model an advantage over other global-illumination models. Designs of proposed buildings can thus be generated from the architectural drawings, radiosity solutions for these computed, and the building designs evaluated for usability, traffic, and aesthetic appeal, by navigating through these virtual buildings. Any shortcomings can then be corrected and new ideas tested out in the design phase itself. This design cycle can be iterated till a satisfactory result has been obtained. The design cycle outlined here appears in the Fig 1.1. Details about the radiosity algorithm can be found in Chapter 2.

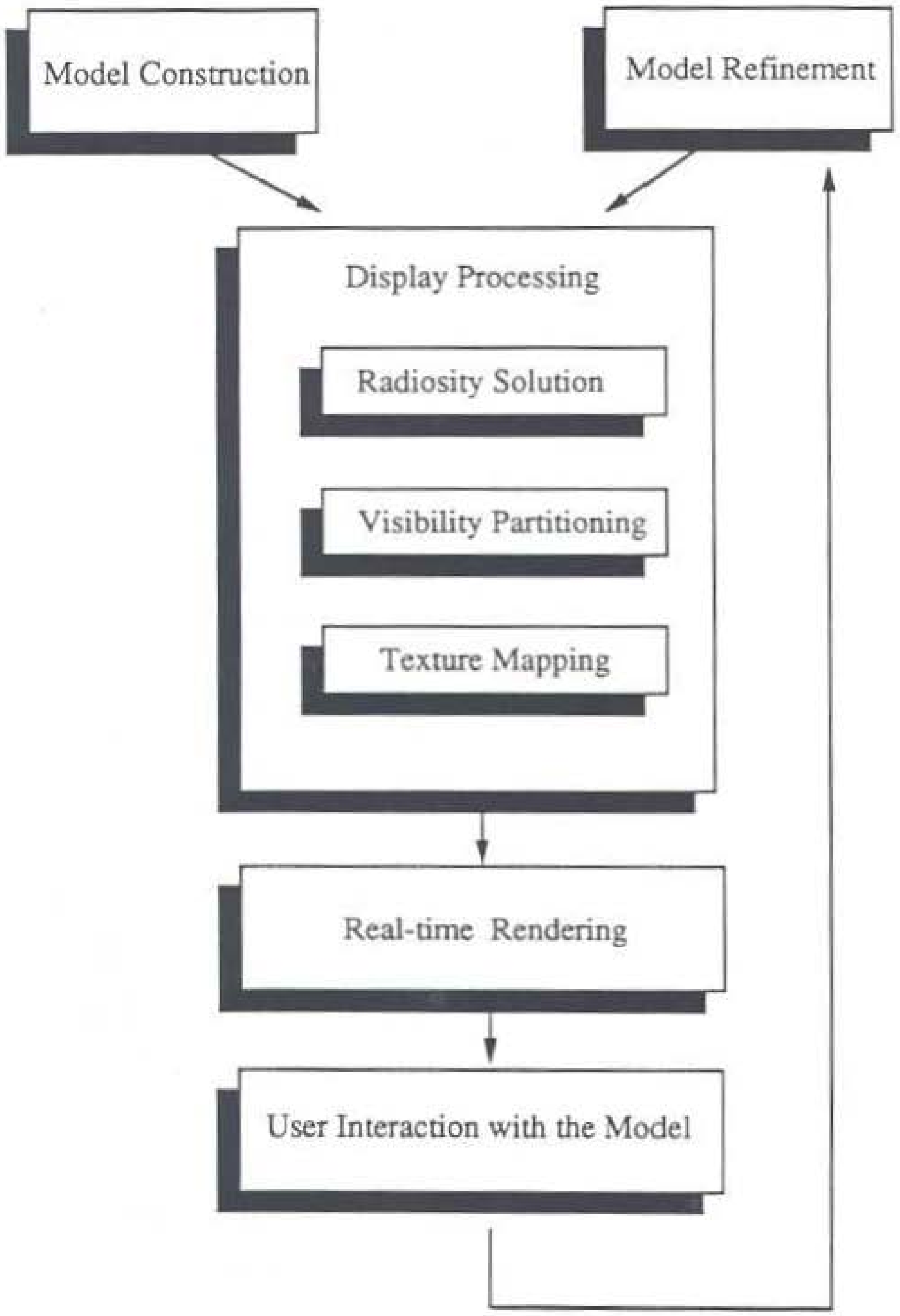


Figure 1.1 The Walkthrough Design Cycle

The radiosity method however is a computationally intensive process and generally requires hours of CPU time on standard workstations for moderately large datasets (on the order of ten thousand polygons). This then becomes the major bottleneck in the design cycle of an architectural model as outlined above. As radiosity-based shading depends on the model geometry, any change to the model necessitates recomputation of the radiosity solution - a very time-consuming task at present. This discourages free experimentation with the model and makes the model-modification process a conservative one. Right now, model modification is done by accumulating changes to the model over a period of days, if not weeks, and then the radiosity solution for it is computed over another one or two days. This is not to criticize the existing implementation, but just to give a feel for the amount of computation involved. In fact, the existing sequential implementation that we have is quite fast as compared to some other existing sequential implementations that we have come across. This speedup is gained partly by using ingenious data structures which exploit the property that most polygons in an architectural model are axially oriented, partly by using adaptive environment sampling techniques [Airey89], and partly from trading off lighting-model accuracy for speed.

Reduction of time in the radiosity computation stage would help bring down the design cycle times and allow for greater flexibility in experimentation with such models. The ability to freely experiment and learn in the process has been the motivating force in paying attention to the generation of such virtual environments in the first place.

With the availability of highly-parallel computers at modest prices and their expected widespread acceptance in the near future, it appears desirable and even necessary to try and devise methods to map such intensive applications on them. There have been documented efforts in the literature that deal with the parallelization of radiosity on shared-memory MIMD multiprocessors in which the number of processors is in the order of tens. However, this problem has not been studied for the emerging class of commercially available, highly-parallel SIMD machines in which the number of processors is in the thousands. Fine-grained parallelization of the radiosity method appears promising as an attempt to reduce radiosity solution times and thus has the potential to make the whole design process a more meaningful one.

The sections 1.3 and 1.4 are meant to serve as an extended abstract of this thesis, outlining the approaches and the results in brief. Readers wishing to get a detailed treatment could just skim through these sections and come back to them after studying Chapters 2 through 5.

1.3 Overview of Approaches Considered

In the design of parallel software, the first concern should be the identification and exploitation of parallelism. The type of parallelism being used depends on the target parallel architecture. We are considering radiosity algorithms for a mesh-connected SIMD architecture. For execution of the radiosity algorithm on this to be efficient, it should give due consideration to these features of the architecture.

Our implementations for the radiosity approaches were on the 4K processor MasPar MP-1 which has a mesh-connected SIMD architecture. The distributed-memory architecture of the MasPar MP-1 data parallel unit (DPU), with 16Kbytes of memory per processing element rules out a per-node duplication of anything but small datasets of around 100 polygons. Distributed memory also dictates that dynamic changes in the mapping of polygons to the processors be minimized. For unlike in a shared-memory system, where this change need only be reflected in a processor-polygon mapping table, in a distributed-memory system this would result in an increased interprocessor communication either due to actual movement of polygons or because of remote accessing of polygons across processor boundaries.

The fact that the target architecture is SIMD implies that for efficient utilization of parallel processors, the special cases to be handled be minimized. For every special case being executed, all those processors that do not qualify to execute that step are idle and thus lower the effective amount of processing being done. This realization goes a long way in helping design more efficient algorithms for SIMD machines.

1.3.1 Conceptual Overview

The basic radiosity method is an iterative one in which each iteration consists of two phases: form factor calculation and energy distribution. Looking at it from the viewpoint of an iterative solution of a linear system of equations $Ax = b$, one can find corresponding stages in the Gauss-Seidel method. The form factor calculation stage corresponds to the computation of one row, say k, of coefficients in the matrix A. The energy distribution stage corresponds to solving for x_k, using the previous values of other $x_i, i \neq k$. Of these two phases, previous studies have indicated that as much as 90% of the time is taken up by the first phase comprising calculation of the form-factor coefficient matrix [Cohen86]. Thus, special attention needs to be paid to calculation of form-factors. The form-factor calculation techniques that we have studied can be divided into two main categories: *ray-casting methods* and *environment-projection methods.*

Ray-Casting Methods fire off rays to sample the environment and thus determine the visibility of polygons. This has been a popular technique for sequential implementations as it gives the freedom of selecting *good* directions and amount of energy to be shot per ray.

Environment-Projection Methods involve projecting the polygons of the environment onto the shooting polygon and then z-buffering to find the visible polygons. These have the advantage of being able to use the z-buffer hardware which is available on some of the newer special-purpose graphics machines [Baum90].

Orthogonal to form-factor calculation by ray-casting or environment-projection is the issue of mapping polygons to processors. Here again we have considered methods that fall in two categories: *Object-Space methods* and *Balanced-Load methods* .

Object-Space Methods are methods in which the processor-polygon mappings are done on the basis of the geometrical distribution of the polygons. The entire dataset space is subdivided into mutually exclusive and collectively exhaustive volumes of space. These volume elements are assigned to the processors in some fixed order. The greatest problem here is ensuring load balancing. If explicit load balancing is not done, the polygonal dataset is often unbalanced across the processors, leading to waste of resources in terms of idle time of the processors that received lighter polygon allocations. If load balancing is attempted by modifying the boundaries of the volume elements then the shapes of the volume elements rapidly become complicated, making it more costly to detect their boundaries [Dippé84]. However, the advantage of these methods is that for a ray-casting approach any given ray only has to be intersected with a restricted set of polygons.

Balanced-Load Methods treat load balancing across processors as their first priority. Thus, these methods assign the polygons to the processors to ensure optimal load balancing. The advantage clearly is the minimization of the processor idle time. However, in absence of any structuring of data, these can be expected to perform poorly for applications where exploitation of geometrical proximity of input polygons is crucial to acceleration of the algorithm.

Conceptually then, the space of approaches considered for the problem appears as shown in Fig 1.2.

	Ray-Casting	Environment-Projection
Balanced-Load	Approach A	Approach B
Object-Space	Approach C	Approach D

Figure 1.2 The Conceptual Approaches

1.3.2 Implementation Overview

Of the four conceptual approaches outlined above, approaches A and C were partially implemented. On the basis of the preliminary results from these, approach B was fully implemented, and approach D was not implemented.

Approach A

In this approach, polygons are spread out evenly over all the processors on the mesh. The progressive refinement radiosity approach has been implemented. Thus, in each iteration a shooting polygon is determined which has the maximum unshot energy among all the polygons. Rays are then cast one at a time from this shooting polygon to sample the environment and transfer energy to the intersecting polygon. The orientation of these rays is chosen such that each of these carries an equal amount of energy. Details on computation of these orientations are given in [Airey89]. The ray that is to be fired is made available to all processors. Each processor then intersects this ray with the list of polygons which it has. The minimum distance of intersection for this ray is computed locally by each processor over all the polygons it has. After this, a global minimum over all these local minimum distances is computed. This gives the intersection distance for the ray. The polygon which had intersected this ray is then found. Once the intersected polygon is found, the energy being carried by the ray is transferred to this polygon. This process is repeated for all the rays from the shooting polygon. After this, the iteration repeats for the next shooting polygon. These iterations continue, till the unshot energy with the most recently selected shooting polygon is below a prespecified threshold value.

Approach B

Here again polygons are spread out equally over the DPU. The shooting polygon is calculated as in Approach A. This is then copied to the limited shared memory on the controller (ACU). All the polygons access this and compute their projection in parallel onto the projection plane of this shooting polygon. The projection plane for our implementation is a single plane that is able to catch 90% of the light energy emanating from the energy shooting polygon. This is described in greater details in [Recker90] and in Chapter 2 of this thesis. The projections are then z-buffered and scan-converted on the projection plane which is implemented to span the entire DPU array. Scan conversion itself can be done either globally (under the direct control of the PE which contains the polygon being scan converted), or locally (by neighbor

PEs). Both of these strategies have been compared in implementation. In contrast to approach A, where the sampling was being done *from* the shooting patch, this approach samples from the environment *to* the shooting patch. This guarantees that within an error tolerance of the resolution of the projection plane, no polygons will be missed in form-factor calculations.

Approach C

In the implementation of the object space based processor-polygon mapping, the spatial subdivision was done to a global grid that was orthogonally adjusted to provide as good a load balance as possible. A hybrid approach of processor-polygon mapping was tried in which the model was divided up into uniform global cuboidal cells. Each cell was mapped onto a whole row of the DPU array. Within each row of PEs, balanced load mapping was done, while across PE rows object space mapping was done. Thus, while each row of the DPU array was balanced, the columns were not. A ray-casting method was studied for this approach and results obtained for this indicated a very high level of interprocess communication requirements. These are summarized in Section 1.4 below.

Approach D

The initial load balancing results that were obtained from Approach C were quite discouraging. They demonstrated that there was a high degree of load imbalance and a high percentage of radiosity interactions were actually directed outside of the local cells (refer section 1.4). This suggested that attempts to subdivide the dataset in the manner done for Approach C to localize radiosity interactions would prove futile. Consequently no further efforts in investigating this approach were invested.

For further details on all of these approaches, the reader is referred to Chapter 4.

1.4 Results

The approaches A and B were tested out on a 3959 patch model of the dataset *Sitterson 365 office* (modeled by John Alspaugh).

In this section by the phrase *x% convergence* I mean the state of the environment when the brightest shooting patch in it has a delta-radiosity that is x% of the shooting radiosity for the first iteration.

Implementing approach A on the MasPar MP-1, the average time for one intersection cycle (intersection of one ray with all the polygons) for the 3959 patch dataset is 4.5 ms. Thus, the effective time for a single ray-polygon intersection is 1.13 μsecs. The total time for the patch radiosities to converge to their respective final values across the radiosity iterations is dominated by this time.

In approach B, a single plane of side-to-height ratio of 3 has been chosen as it permits 90% of the shooting patch's radiosity to be shot. Comparisons were made across different resolutions of the single plane. The following times were observed for the *Sitterson 365 office* model. For a single plane with 64×64 resolution, the average time per iteration is 0.24 secs. For a single-plane with 128×128 resolution, the average time per iteration is 0.45 secs. For a single-plane with 256×256 resolution, the average time per iteration is 2.15 secs. These results are for a local scan-conversion technique for generating the item-buffers.

If the number of rays to be fired per shooting polygon for approach A are n, then one iteration of approach B corresponds to n intersection cycles of approach A. Since the time for a 128×128 single-plane for approach B is 0.45 secs, it allows approach A to fire up to 100 rays (time per ray intersection cycle is 4.5 ms), before approach A starts losing out. Thus approach A is an order of magnitude slower.

A global positioning method for scan-conversion was also attempted with approach B, but here again the geometry dependency made load balancing a concern. For a 4K resolution single-plane with a side-to-height ratio of 1, the total number of tuples generated is 27K with a maximum of 1.4K tuples at a processor and a minimum of 0. All these tuples need to be z-buffered on the processor where they fall, making this a highly-imbalanced operation.

While studying approach B, tests were also done to determine what fraction of the form-factors are reused. This was done with a view to storing the calculated form-factors to avoid their recomputation. Unfortunately, most of the reuse in form-factors starts occurring quite late in the convergence. Thus, for the 3959 patch *Sitterson 365 office* model, only 6 form-factors are reused in the first 177 iterations in which convergence to 97.2% takes place. It is only later that a high fraction of form-factors begins to get reused, but by then the convergence is almost complete.

Approach C was tested on a $8 \times 8 \times 1$ subdivided model of the *Sitterson 365 office*. Geometry based distribution of these with orthogonal grid balancing into 64 cells yielded a poor load balance, with a minimum of 6 and a maximum of 208 polygons. Further refinement using subdivision and redistribution within the same DPU ray, brought down this imbalance to a minimum of 6 and maximum of 154. Using this data in the approach as outlined in Section 1.3.2 and firing one ray per polygon (along the polygon normal), it was observed that only 1983 rays intersect the polygons within the same cell, while 1353 do not and thus have to be passed onto other DPU rows. These 40% of the rays will contribute to a high inter-row communication and would also need to recompute the intersections with the polygons on the row where they reach. The problems of load imbalance and the expected high amount of interprocess communication suggested that we invest our effort elsewhere.

Whereas approach A requires fewer iterations, approach B has a smaller time per iteration. Overall, approach B is a winner in terms of total time required to achieve a given convergence.

Three levels of possible parallelism in radiosity have also been identified in this thesis. Thus, depending on the number of processors which are available one could choose to parallelize at different levels.

1.5 Guide to the Chapters

This chapter provided a brief overview of the problem investigated, the approaches taken, and the results that have been obtained. This chapter is meant to serve as an extended abstract, presenting the key aspects of the thesis in a nutshell.

Chapter 2 describes the radiosity method in general. This is followed by a discussion of issues relevant to its parallelization.

Chapter 3 reviews the parallel radiosity work that has been done in literature so far. Since computation of form-factors is often done using a ray-casting approach, parallel ray-tracing techniques can be used for this purpose. Therefore, a brief overview of parallel ray-tracing methods has also been done here.

Chapter 4 is devoted to the description of the radiosity algorithms and their implementations on the MasPar MP-1. This chapter discusses the various approaches that have been considered and the results that they yielded. A brief overview of the MasPar MP-1 is also provided here.

Chapter 5 discusses the possible extensions to this work.

Appendix A contains the source code listing of the implementation.

Chapter 2

The Radiosity Method

The radiosity method models the interaction of light between diffusely-reflecting surfaces. One of the strongest properties of this global-illumination model is the view-independent solution that this provides. This permits viewing of a geometric dataset at interactive rates from any viewpoint and direction, once the solution has been computed. Introduced from thermal engineering to computer graphics by Goral et al.[Goral84], this method realistically simulates diffuse lighting effects such as soft shadows and the diffuse inter-reflections. Such lighting effects are commonly found in building interiors, where most of the surfaces are diffuse reflectors and emitters. In this chapter, we will present an overview of the radiosity method, the bottlenecks involved, and an outline of stages in it that can be parallelized.

2.1 Overview

Radiosity B_j for a surface j is defined as the total rate at which radiant energy leaves that surface in terms of energy per unit time and per unit area [Goral84].

This flux B_j being emitted from the surface j is composed of two components:
(i) The rate of direct energy emission E_j from the surface j per unit time and per unit area.
(ii) The rate of reflected energy from the surface j per unit time and per unit area. If the reflectivity of the surface is ρ_j and the radiant energy incident at the surface j

per unit time and per unit area is H_j, then this rate of reflected energy is $\rho_j H_j$.

The radiosity equation for surface j is therefore:

$$B_j = E_j + \rho_j H_j \tag{2.1}$$

If it is assumed that the environment under consideration :

(i) is composed of only diffuse surfaces (say n in number) and

(ii) is *closed* in the sense that whatever flux is emitted from one surface is incident on one or more other surfaces within the environment ,

then the total incident flux at a surface j can be computed as follows:

$$H_j = \sum_{i=1}^{n} B_i F_{ij} \tag{2.2}$$

F_{ij} is the *form-factor* for surface i with respect to surface j. This denotes the fraction of radiant energy leaving surface i and incident on surface j. It is usually defined as the solid angle that the visible part of surface i subtends from the centroid of surface j divided by 2π.

Equation 2.1 can now be written as:

$$B_j = E_j + \rho_j \sum_{i=1}^{n} B_i F_{ij} \qquad for \quad j = 1, n \tag{2.3}$$

This is summarized in fig 2.1.

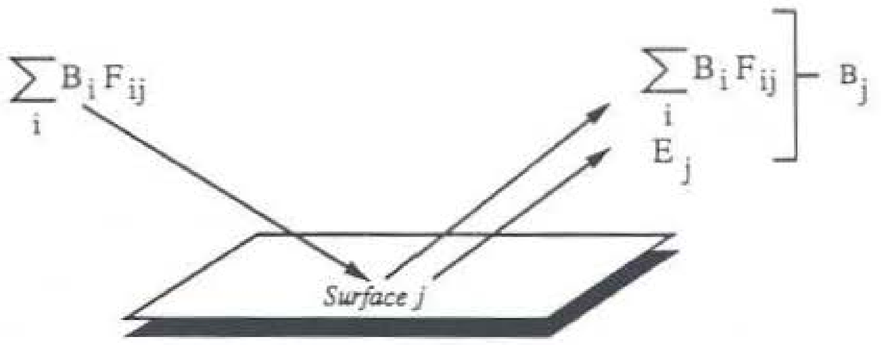


Figure 2.1 Interactions at Surface j

This yields a system of n linear simultaneous equations in n unknowns B_j. The F_{ij} are determined from the geometry of the environment and E_j the emittances of the light sources in the environment, are assumed given. This system is shown in fig 2.2.

$$\begin{bmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \cdots\cdots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \cdots\cdots & -\rho_2 F_{2n} \\ . & . & \cdots\cdots & . \\ . & . & \cdots\cdots & . \\ . & . & \cdots\cdots & . \\ -\rho_n F_{n1} & -\rho_n F_{n2} & & 1-\rho_n F_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ . \\ . \\ . \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ . \\ . \\ . \\ E_n \end{bmatrix}$$

Figure 2.2 The Radiosity System of Equations

An exact solution of this system, by methods such as Gaussian elimination, requires $O(n^2)$ space and is of $O(n^3)$ complexity, making it cumbersome to use for even small datasets (n of the order of hundreds). In most real applications n is of the order of thousands. The coefficient matrix being strictly diagonally dominant [1] (if the participating surfaces are assumed to be planar $F_{ii} = 0$ for all i), iterative techniques such as the Gauss-Seidel method fare much better for this system. In the *progressive refinement* approach [Cohen88], this solution is accelerated by choosing the variables B_j to be solved in the order of their contribution to the environment. This is the same as the notion of pivoting to move the bigger values to the upper left corner of the matrix to be solved for at each iteration. Thus, brightest surfaces are used earliest in the solution, so that the convergence to the exact solution is rapid in the initial stages. The time for convergence to an *acceptable* solution has been shown to be linear in the number of surfaces by using this approach [Cohen88]. The $O(n^2)$ space requirements for storing the coefficient matrix of this system are done away with by computing the form-factors on the fly [Cohen88]. Viewing this approach from the viewpoint of energy distribution, at the beginning of each iteration the polygon with maximum

[1]The diagonal element is greater than the sum of other elements that lie either along its row or along its column.

energy, say i is chosen as the shooting polygon. The amount of energy with this polygon is referred to as the *unshot energy* for the shooting polygon. This energy is added to the unshot energies of all other polygons j, based on the form-factor values F_{ij}.

Calculation of form-factors is done either by using analytical methods or, more commonly, by using sampling methods. The latter approach offers the flexibility of varying the speed of computation by choosing the accuracy desired. The overall convergence to a result within some prespecified tolerance can be accelerated by suitably trading-off accuracy for speed as the computation proceeds.

The form-factor F_{ij} is equal to the fraction of the base of a unit-radius hemisphere centered on surface j that is covered by the projection of surface i on that hemisphere [Cohen85]. The fig 2.3 adapted from that paper explains this.

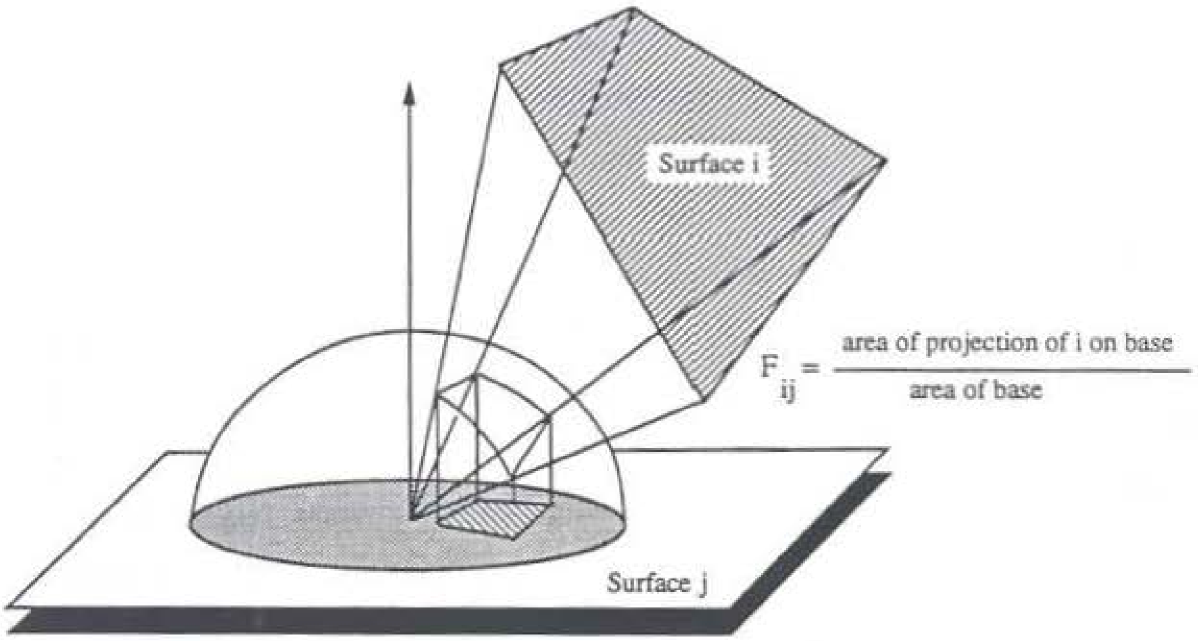

Surface i
$F_{ij} = \frac{\text{area of projection of i on base}}{\text{area of base}}$
Surface j

Figure 2.3 The Form-Factor Analog

However, projection of the environment on a quadratic surface, such as a hemisphere, is costly, unless there are special quadratic primitives provided in the machine hardware (Pixel-Planes 5, for instance, has an ability to render on quadratic surfaces). Thus, the notion of a *hemi-cube* was introduced [Cohen85]. A hemi-cube is a five-sided half cube that fits atop the energy distributing hemisphere to approximate its energy distribution effects and at the same time provide planar projection surfaces. Each surface of the hemi-cube is divided into square pixels. The *delta form-factor* Δf_{iq} for one of these pixels q is defined as the form-factor between that pixel and the surface i. The total form-factor F_{ij} for surface i with respect to the surface j is the sum of all the delta form-factors of the pixels comprising the hemi-cube for the surface j which cover the projection of surface i on surface j. This can be specified as:

$$F_{ij} = \sum_{q \epsilon Q} \Delta f_{iq} \qquad \forall \; hemi-cube \;\; pixels \;\; q \;\; on \;\; j \ni \Delta f_{qi} = 1 \qquad (2.4)$$

Here Q is the set of hemi-cube pixels covered by projection of surface i onto hemi-cube.

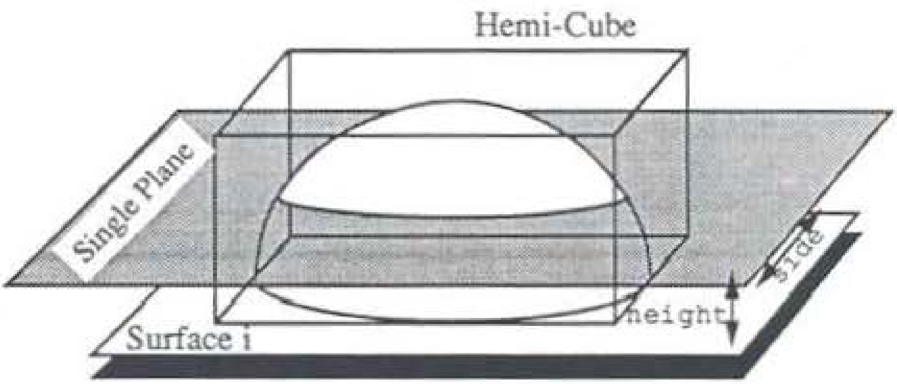


Figure 2.4 Hemi-Cube and Single Plane for Form-Factor Calculations

A step further in acceleration of form-factor calculation was taken by using the notion of a *single-plane* [Recker90]. The idea here is to approximate the effects of a five-plane unit hemi-cube by a sufficiently large single plane. Depending on how large this plane is and how much above the shooting polygon it is placed, the ratio of its side to its height (as shown in fig 2.4), henceforth referred to as the *side-to-height* ratio, can be computed. By increasing the side-to-height ratio of this single-plane to around 3, almost 90% of the energy can be shot [Recker90]. Thus, instead of five planes on which to project the environment, a single plane of projection can be used with 90% accuracy. This suffices for most practical applications. For better accuracy (and lesser aliasing), the pixels on the single-plane that are *near* the center should be of a smaller size, because most of the energy would be expected to be radiated from here. This follows from the cosine distribution of the energy being radiated from a diffuse surface. To understand this better, one should refer fig 2.3. The fraction of the base of the energy-distributing hemisphere that is covered by the projection of a unit area on the top of the hemisphere would be more than the corresponding fraction for a unit area located on the side of the hemisphere. This technique of having pixels of two different resolutions on the single-plane is the *modified single-plane* algorithm. Using the ideas in [Airey89], this idea can be extended so that we have the sizes of pixels on the single-plane of height h such that the areas which they subtend on the hemisphere of radius h are all equal.

Calculation of form-factor sampling is done in two ways. The first method is by casting rays from the centers of each of these pixels (on hemi-cube surfaces or on the single-plane) along the direction vector from the center of the shooting polygon (which could be jittered). The second method is by projecting the environment on the projection-surface (planes of the hemi-cube or the single-plane) for the shooting polygon. The former method, henceforth referred to as the *ray-casting method* , is the method of choice for most software implementations. The latter method, henceforth referred to as the *environment-projection method*, is preferred for graphics-hardware-oriented implementations. The reason is that the process of finding the delta form-factors using this method is quite similar to the process of rasterization of 3-D geometry with hidden surface removal [Baum90]. Thus, the graphics pipeline well-tuned for this purpose can be used to advantage in calculating the form-factors.

For the purposes of this thesis we will refer to the plane on which the environment is projected, be it a side of the hemi-cube or the single-plane, as the *projection-plane*

To enhance realism, one needs to be able to faithfully reproduce the light gradient across a surface. However, if this gradient is large at some places, then to realistically simulate light across such a surface, one needs to approximate it by smaller polygons. Thus, the original polygons are subdivided to get more realistic light gradients. We will refer these smaller subdivided polygons by the name *patches*.

2.2 The Bottleneck

The radiosity process can be subdivided into two main phases: the *form-factor calculation* phase and the *energy-distribution* phase. The former corresponds to the calculation of the coefficient matrix elements F_{ij} in the linear simultaneous system of equations, and the latter corresponds to its solution. These two phases are conceptually distinct; for ease of understanding it is preferable to treat them as such.

These phases are easy to identify in environment-projection implementations where they remain distinct. However, in ray-casting implementations, these phases are often tightly interleaved, blurring this distinction. To clarify this further, let us consider the implementation of the ray-casting approach. Here, the most common technique is to treat the ray fired from a polygon as laden with a certain amount of energy that is transferred to the polygon with which the ray intersects. Thus, in this case the two phases of form-factor computation and energy distribution proceed hand-in-hand right down to the level of delta form-factor computation. Treating the identification of the polygon with which the ray intersects as being in the form-factor computation phase and the energy transfer to the polygon as being in the energy-distribution phase, it is easy to imagine that it is the form-factor computation stage which takes up most of the time.

Once the form-factors have been computed, the energy distribution step becomes trivial. This is supported by statistics from past studies [Cohen85], [Cohen86] which show that form-factor computation takes up around 90% of the time. The form-factor computation being the main bottleneck in the radiosity process, faster ways of calculating this must be found. As will be discussed in the next section, this stage turns out to be parallelizable, offering a good opportunity to reduce the overall radiosity solution times.

2.3 Parallelism

Solution of a system of linear simultaneous equations by a method of successive displacements is inherently sequential. If viewed in this light, the solution of the radiosity system of equations using the Gauss-Seidel method is a sequential one, at least at a macro level. However, there are sub-stages within this overall process that can be parallelized.

The radiosity process can be parallelized at the level of form-factor calculations. As has been mentioned before in Section 2.1, the form-factor F_{ij} is the sum of delta form-factors Δf_{iq} such that $\Delta f_{qi} = 1$. Each of these delta form-factors can be computed in parallel.

If the delta form-factors are computed by using the ray-casting approach, then the intersection of each ray with the polygons in the environment can be computed in parallel. The distance from the ray-origin to the intersection point on the polygon can be computed in parallel for all polygons and then the polygon with the minimum distance from the ray-origin can be selected as the polygon intersected by the ray.

Apart from these places, parallelism can also be exploited at a macro level, if one is prepared to move away from the Gauss-Seidel method. Using a simultaneous displacement method for solving the radiosity system of equations, such as the Jacobi method, one could carry out the solution for every variable in parallel. If a hybrid approach between the Gauss-Seidel and Jacobi methods is chosen, then that too holds promise of parallelism. In this approach, a set of k out of a total of n variables are chosen as the variables to be solved for in the current iteration. Each of these k variables is solved for in parallel using the values of the previous iteration for all variables. Then in the next iteration, these new values of k variables are simultaneously used. This idea, though not explicitly stated in this fashion, is used by Chen in his distributed radiosity approach [Chen89] (ref subsection 3.1.1).

Selection of the variable in the radiosity system of equations that needs to be solved next, the shooting polygon, itself can be done in *O(log n)* time in parallel over $O(n)$ processors, instead of $O(n)$ time in a sequential method.

Once the form-factors have been computed, the distribution of energy across all the polygons can be done in parallel.

To summarize, if we look at the process of radiosity solution as solving of a system of n linear simultaneous equations $Ax = b$, then parallelism can be exploited in form-factor calculations at several levels as shown in Fig. 2.5.

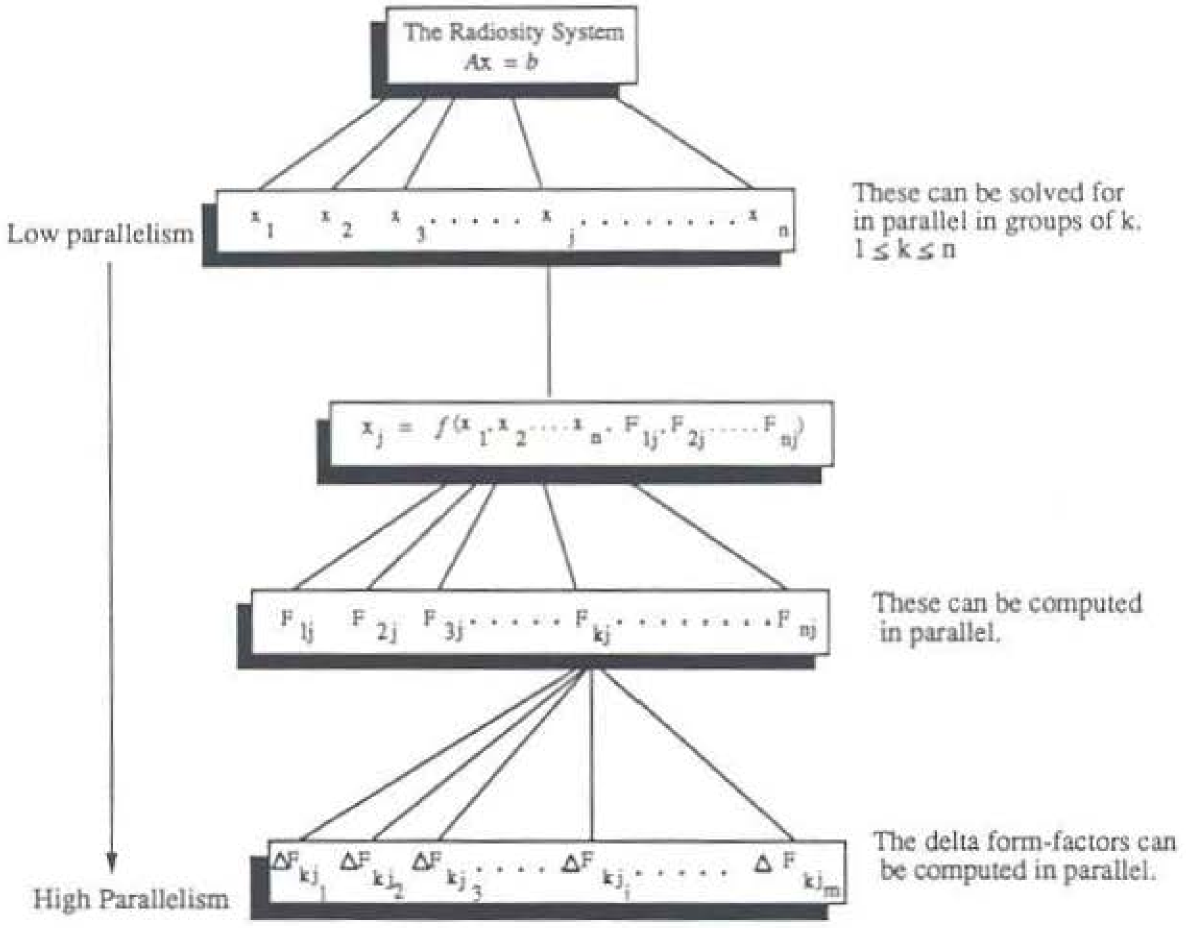

The Radiosity System $Ax = b$
Low parallelism
x_1 x_2 x_3 . . . . . x_j . . . . x_n
These can be solved for in parallel in groups of k. $1 \le k \le n$
$x_j = f(x_1, x_2, \ldots, x_n, F_{1j}, F_{2j}, \ldots, F_{nj})$
F_{1j} F_{2j} F_{3j} . . . . . F_{kj} . . . . F_{nj}
These can be computed in parallel.
ΔF_{kj_1} ΔF_{kj_2} ΔF_{kj_3} . . . ΔF_{kj_i} . . . . . ΔF_{kj_m}
The delta form-factors can be computed in parallel.
High Parallelism

Figure 2.5 Parallelism in Radiosity

These levels are:

(i) Finding the new values of some k variables in parallel in each iteration. $k = 1$ for Gauss-Seidel, $k = n$ for Jacobi, and $1 \le k \le n$ for the hybrid method outlined above.

(ii) Computing the rows of the coefficient matrix A in parallel.

(iii) Computing the value of each element in a row of the coefficient matrix A by parallel methods.

These three levels of parallelism are nested. One can go down these levels with an increasing availability of processors to exploit increasing parallelism. Since computation of the form-factors takes up almost 90% of the time [Cohen85], [Cohen86], more effort should be invested more in exploiting parallelism at the levels (ii) and (iii) as outlined above.

Chapter 3

Previous Work

Since the introduction of radiosity as a global illumination method in 1984 [Goral84], various attempts have been made to improve and accelerate it. Most of them, [Cohen85], [Cohen86], [Cohen88], [Airey89], [Chen90] to name a few, have been presented with uniprocessor environments in mind. Only recently has attention been focussed on solving this problem in multiprocessor environments. This chapter provides a brief overview of the previous work done in parallel radiosity.

Form-factor calculation being computationally the most significant phase of the radiosity method, and ray-casting being a popular approach in this phase, one can only expect that most of the available parallel ray-tracing techniques will be of use in the endeavors to parallelize radiosity. Taking this into account, this chapter also provides an overview of some parallel ray-tracing techniques that have been investigated in the past.

3.1 Parallel Radiosity

Parallel radiosity is a relatively new field, no more than three years old. Most of the attempts at radiosity parallelization, that have been documented in the literature so far, have focussed either on loosely-coupled multiprocessing or on coarse-grained parallelization. This section presents a synopsis of the work that has been done in the past in this area.

3.1.1 Radiosity on Loosely-Coupled Systems

Beginnings in parallelization of radiosity were first made in 1989 when Chen [Chen89] exploited coarse-grain parallelism over a network of Hewlett-Packard 835 workstations via ethernet using Unix sockets. The process model used was the client-server paradigm.

One of the workstations is designated as the server and the rest as clients. Each client has a complete copy of the geometry information. The server has the energy information for each element of the dataset. If there are n clients , then at the beginning of each iteration the server selects n shooting patches and distributes them among the clients. In the progressive refinement approach, these would be the n brightest patches. This is equivalent to exploiting the first-level parallelism described in Section 2.3.

Utilizing the dataset geometry information available locally, each client then computes the form-factors for the shooting patch assigned to it. These form-factors are sent back to the server by the clients. Using this information, the server distributes the energy. Thus, the form-factor calculation proceeds in parallel and the energy distribution is sequential.

Performance results from this approach show that for a 180×180 hemi-cube resolution for a 5196 patch dataset, average time per iteration is around one second for five or more client processors.

This was an important first step in parallelization of radiosity and it proved successful in reducing the radiosity calculation times. However, as is true for most pioneering efforts, this did not have the benefit of deriving insights from previous work. Its biggest drawback is the possibility of the server becoming the bottleneck with increasing number of clients. Thus, the method is not scalable. It also involves duplication of the entire dataset at each client processor. Besides the memory costs, duplication raises the problems of consistency in an interactively changing environment. However, these problems are becoming apparent only now, when we have reached the threshold of interactive radiosity.

3.1.2 Radiosity on Tightly-Coupled Systems

Parallel radiosity algorithms have been developed and implemented for tightly coupled multiprocessors too: [Baum90], [Drettakis90]. These attempts have focussed primarily on coarse-grain parallelization with the number of processors used being in the order of tens. Further, the target architectures for these were primarily shared-memory MIMD systems.

Baum and Winget [Baum90] implemented their algorithm on the Silicon Graphics line of graphics workstations. The radiosity process is viewed by them as a producer-consumer problem.

The consumer first selects the shooting patch. The producer then is responsible for generating the delta form-factors. The consumer collates these to find the actual form-factors and distributes the energy using these.

The delta form-factor computation is done by hemi-cube item-buffer generation. This involves projecting the environment on the faces of the hemi-cube and z-buffering for each pixel of the hemi-cube. This process is the same as hidden-surface removal and scan-conversion for 3-D objects. Graphics hardware, finely tuned for this purpose, that is available on these workstations can be used for this purpose. Thus, the producer task is implemented on the graphics hardware.

The consumer task is parallelized over the multiple host processors that are available on these workstations. The item-buffers generated by the producer are partitioned into blocks that are distributed over the host processors using dynamic scheduling to maintain a good load balance. Each host processor computes local form-factors from these item-buffers. These local form-factor values are then summed over the host processors to find the actual form-factor values. Energy distribution is then done in parallel over the host processors.

Using an eight processor system and a hemi-cube resolution of 150×150 and a model size of 8247 patches, this approach computed each radiosity iteration in one second.

This has been integrated with a walkthrough program so that the user can interactively walk through a dataset while the solution refines.

The idea of using the graphics hardware to solve for the delta form-factors is a novel and useful one. This also, however, limits the target architectures to multiprocessor machines with specialized 3-D graphics capability. Baum and Winget's integration of the radiosity solution with the walkthrough program is an important step towards developing user-modifiable, radiosity-lighted systems.

Drettakis et al. [Drettakis90] present a parallel method for calculating generalized global illumination. They have designed their algorithm for common-bus shared-memory MIMD architectures having about 10 - 30 processors. Their method involves recursively subdividing a given dataset into cells in an octree fashion till the number of objects within each cell becomes sufficiently small. Energy is then distributed in parallel with one direction of energy beams devoted to each processor.

3.2 Parallel Ray-Tracing

Ray-tracing is an inherently parallel technique. Rays are cast corresponding to each pixel of the screen. These rays are traced through the dataset, reflecting and refracting according to the properties of the surfaces that these intersect. Parallelism in this method can be exploited in two ways. First, the rays can be cast in parallel. Thus, at a given time several rays could be traversing the dataset. Second, the intersection calculations for each ray to determine the surfaces that intersect it (and their relative order), can be done in parallel. An overview is given here of two parallel ray-tracing papers, one designed for MIMD computers and the other for a SIMD computer.

3.2.1 Ray-tracing on MIMD systems

A spatial-subdivision based algorithm for MIMD systems has been proposed by Dippé and Swensen [Dippé84]. The three-dimensional space of the dataset is subdivided into several volume elements or voxels. Each voxel is assigned to a processor. This subdivision is adaptively changed at run time to maintain uniformity of load. Rays originate from the voxel that contains the screen upon which the scene has to be ray-traced. These rays pass through all the voxels, and hence the processors corresponding to them, that are along their direction of travel till they find suitable intersections. At each processor, intersection calculations are done for all the objects that lie in the voxel corresponding to this processor, with the incoming rays. A ray that does not find any intersection is passed along to the appropriate processor corresponding to next voxel along the ray's path. Each ray is traced in this fashion up to its intersection. Thus, each voxel is processed independently and in parallel.

To alleviate load-balancing problems, the voxel shapes are changed dynamically to maintain a good load balance of objects and rays. Neighboring processors share load information and processors that are more heavily loaded transfer their load to lightly loaded processors. The voxel shapes that have been proposed here are orthogonal parallelopipeds, general cubes and tetrahedra - all of which are volumes bounded by planar surfaces. These have been preferred over other surfaces for the ease of their boundary detection and hence the voxel identification. Two-dimensional analogs of these shapes, adapted from [Dippé84], are shown in fig 3.1.

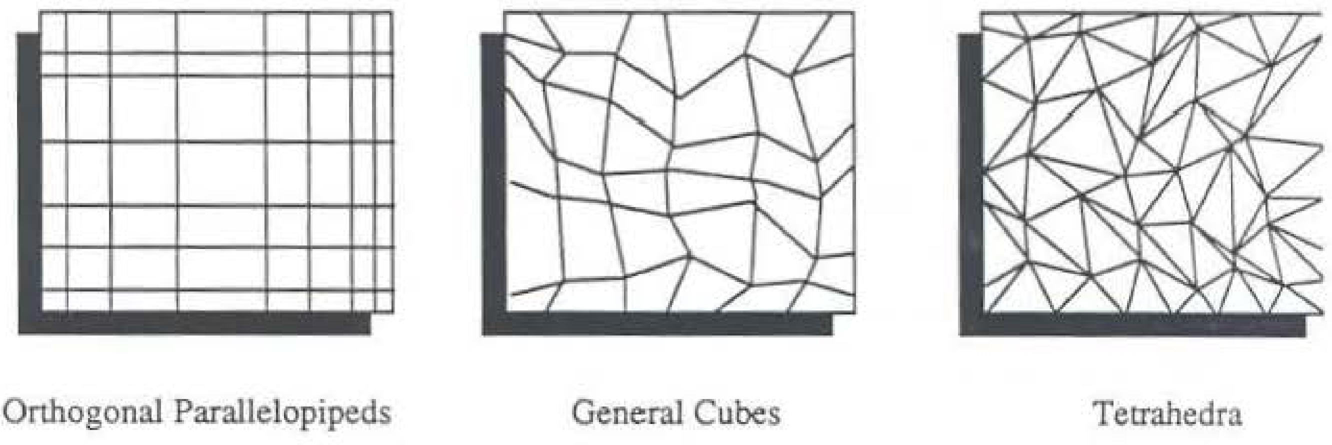


Figure 3.1 2-D Analogs of 3-D Spatial Subdivision Schemes

Planar surfaces however cause splitting of objects across the voxel boundaries. The authors have proposed trading-off the amount of splitting necessary with the degree to which the load is balanced.

3.2.2 Ray-tracing on SIMD systems

Ray-tracing on highly-parallel hypercube connected processors has been studied by Delany [Delany88]. This paper presents a very interesting and elegant way of mapping ray-tracing on hypercube connected SIMD processors. The implementation was done on a 16K processor Connection Machine CM-1. At a conceptual level, this technique also uses incremental voxel traversal as described in Section 3.2.1.

The three-dimensional space of the dataset is subdivided in an octree fashion. The objects and the rays are assigned xyz-triples, called *key words*, based on their spatial location. Ray origins are used for assigning the initial key words to the rays.

Voxels are considered over edge lengths varying in powers of 2 from a scale of unit-length to the largest side of the dataset (this corresponds to the voxel that encompasses the whole dataset).

As before, rays are traced incrementally, one voxel at a time. The scale of the voxel traversed however varies. If there is a large open space with no objects immediately ahead of the current ray point in the direction of the ray, then the voxel traversed by that ray would be the largest one (with the constraint that its edge length be a power of 2), that would fit inside that space. This is the basic idea which causes the ray-tracing time for a particular ray to be logarithmic in its free path length (length of the ray-path that has no objects to be tested for intersection).

For each voxel traversal, the whole set of the object and ray point key words are sorted across the entire hypercube. Objects at a given scale and location are spread out over the consecutive processors along with the rays. A worthwhile point to note here is that since objects are sorted within each voxel, we have available here an induced octree ordering on the key words. Each ray-point can look at the key words of the preceding and succeeding objects which would be on *nearby* (if not the same) processors, and from them determine the smallest volume of space such that there are no objects within it. If this volume is *small* then ray-object intersection calculations need to be performed. If this is *large* , then the ray-point is simply moved to the far edge of this volume. This completes one voxel traversal.

Voxel traversal is done either till the ray intersects or it is found to be outside of the entire dataset space (in which case it did not intersect any objects).

Using a CM-1 with 16K processors, time to fire 307200 rays in a scene containing 8000 objects, with 3 orders of reflection was 347.2 CPU seconds with this approach.

Chapter 4

Parallel Radiosity Techniques

4.1 MasPar MP-1 Overview

Even though the algorithms described in this chapter are applicable to all mesh-connected SIMD computers, their implementation has been done only on the MasPar MP-1. It would be therefore in order to give a brief overview of the MasPar MP-1 before describing the approaches. A more detailed description of the MasPar MP-1 can be found in [Maspar90].

4.1.1 Processor Architecture

The MasPar MP-1 has a front-end which is currently a VAX 3520. The parallel processing is done on the data parallel unit (DPU), which comprises a processor element array (PE array) and a controller for this array, ACU (Array Control Unit).

Each processor element (PE) is a 1.8 MIPS 32-bit control processor with 16Kbytes of RAM and 1500 bits of register space. The PEs are organized in a 2D mesh with direct connections to 8 nearest neighbors. This is true even for the boundary PEs, so the PE array layout is really toroidal. The MasPar MP-1 configuration on which the implementations for this thesis are done has 4K PEs arranged in a toroidal square mesh of 64×64 processors.

The ACU has a 14 MIPS control processor with 128Kbytes of data memory and a total of 1Mbyte of program memory. The ACU is responsible for sending instructions and data to the PE array.

4.1.2 Communication Architecture

There are two major kinds of communications possible on the DPU in the MasPar MP-1.

The first deals with ACU-PE array communications. These take place over a special ACU-PE bus. This is a two-way communication path where the ACU can send data and instructions to the PE array and get back data results from the PEs.

The second type of communication is among the PEs within the PE array. This can be one of the following two major types:

a) X-Net: These are directional communications that can take place between PEs which are located along one of the eight nearest-neighbor directions from one another. These have low latency and high bandwidth.

b) Global Router: These are general communications that are possible between any PE and any group of PEs. Although useful and convenient, general communication has higher latency and lower bandwidth than nearest neighbor communication.

4.2 Algorithms

A comparative study of three radiosity algorithms has been done on the MasPar MP-1. Although the interconnection network of the MasPar MP-1 permits direct-connection to eight nearest neighbors, only four (east, west, north, and south) of these are used in the implementations described below. Thus, these methods are applicable to all those mesh-connected SIMD computers in which the interior nodes have interconnection degrees of four or more.

The approaches studied differ in the processor-polygon mappings used and in the method of delta form-factor computation.

4.2.1 A Balanced-Load Ray-Casting Approach

In this approach, the patches are uniformly distributed over the processor array, and the form-factors are computed by a ray-casting approach.

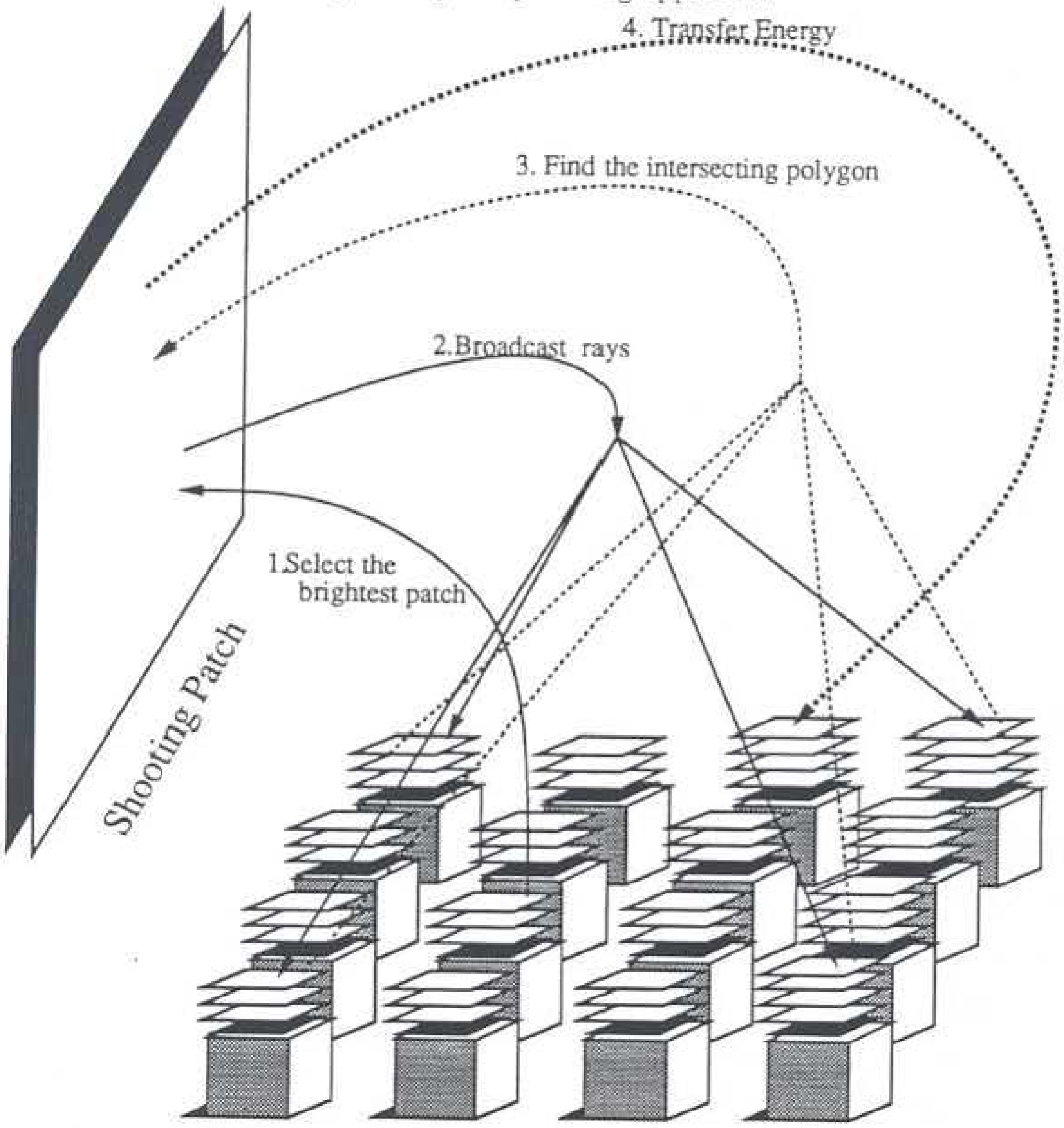

4. Transfer Energy
3. Find the intersecting polygon
2.Broadcast rays
1.Select the brightest patch
Shooting Patch

Figure 4.1 The Balanced-Load Ray-Casting Approach

Outline

Here, the patches derived from the polygons are spread out over the entire mesh of processors in a uniform fashion. This ensures that the processors have an even load.

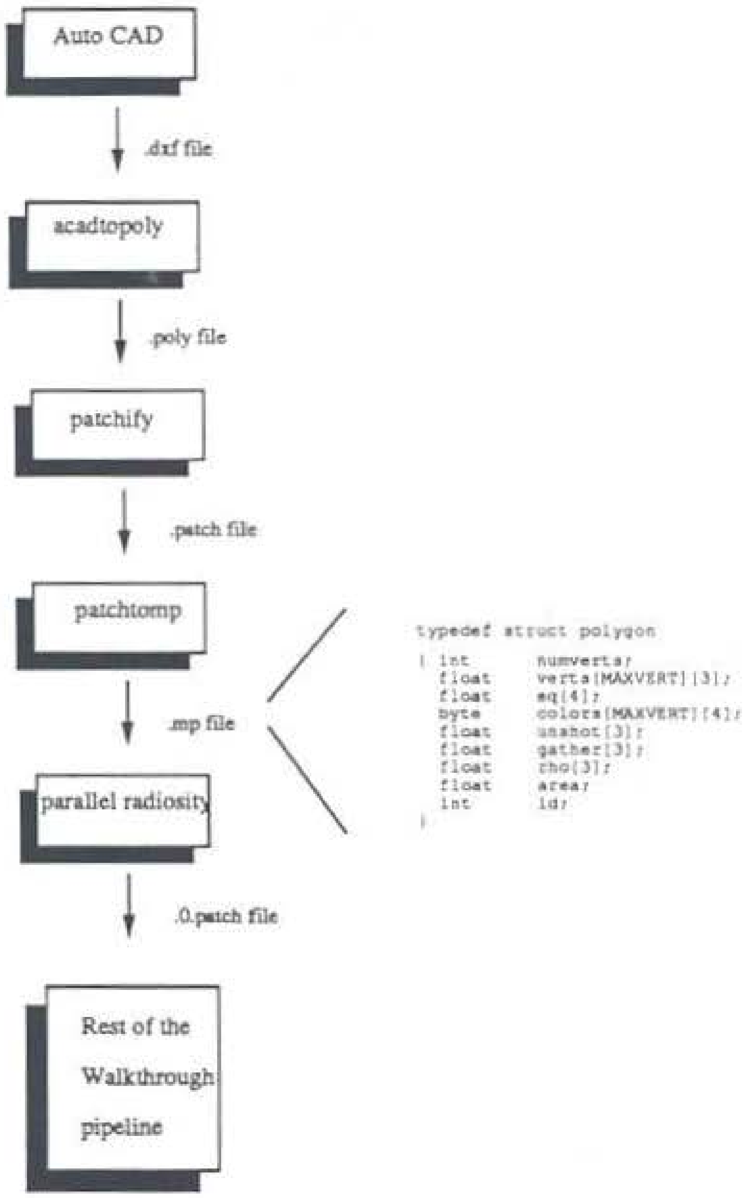
The aim here is to exploit the parallelism as much as possible, at the cost of ignoring the locality property of potential interactions in the radiosity method.

Following the progressive-refinement approach, at the beginning of each radiosity iteration, the patch with the highest unshot energy is selected as the *shooting patch* The origin and orientation of the rays to be fired from this patch are then determined. Intersection testing is done one ray at a time by all processors. Each processor tests for an intersection of the ray with the patches that it has and computes the minimum distance of intersection. By finding the global minimum over these local minimum distances on all the processors, the patch that is intersected by the ray is determined.

The energy carried by the ray, determined by the orientation of the ray with respect to the shooting patch's normal and the radiosity of the shooting patch, is transferred to the patch intersected. This process is repeated for all the rays from the current shooting patch. After that, the next shooting patch is selected as described before and the iteration repeats. These iterations continue till the energy of the shooting patch selected is below a certain minimum threshold.

Implementation Notes

The polygons that have been output from the AutoCAD are converted to a `.poly` file format from a `.dxf` format as in the usual Walkthrough pipeline. These are then subdivided into patches based on a global grid by the program `patchify.c`. The resolution of the grid along either of the dimensions can be specified by means of command-line arguments. The output of the `patchify` program is a `.patch` file which is in ASCII. This is converted to a `.mp` file format by the program `patchtomp`. The `.mp` file is in VAX binary format to enable faster loading times. The type-structure of the patches here is the same as used on the MasPar. The description of this patch format and this sequence of stages is shown in the figure 4.2. The output of the radiosity runs is in the `.0.patch` format file that is consistent with the rest of the Walkthrough pipeline.



```
typedef struct polygon
{ int         numverts;
  float       verts[MAXVERT][3];
  float       eq[4];
  byte        colors[MAXVERT][4];
  float       unshot[3];
  float       gather[3];
  float       rho[3];
  float       area;
  int         id;
}
```

Figure 4.2 The Modified Walkthrough Pipeline

The `.mp` file is read on to the MP-1 DPU array by the `p_read()` function call. Selection of the brightest patch on the DPU is done by performing a global `reduceMax()` operation over the shooting energies of all patches. This returns the shooting energy of the brightest patch. From this the brightest patch is found and is designated as the shooting patch for the current iteration.

The rays to be shot from the shooting patch are determined next. In the current implementation, the origin of the rays is fixed at the center of the shooting patch, but it can be jittered using techniques described in [Airey90a]. The direction of each ray is along the vector from the center of the shooting patch to the center of one of the subdivisions of the unit hemisphere on the shooting patch. These subdivisions are the non-uniform subdivisions along radius and theta dimensions of the shooting hemisphere as described in [Airey89]. This approach ensures that the areas projected by such subdivisions onto the base of the hemisphere are equal and hence that the energy carried by each ray is the same.

The code for finding the ray-polygon intersection has been taken from the book Graphics Gems [Glassner90]. The ray is tested against all the patches at every processor in parallel. The MasPar library function `reduceMin()` is then used to find the minimum distance of intersection. After the patch that intersects the current ray is located, the energy carried by the ray is transferred to it.

All the variables that are ray-independent in the ray-polygon intersection code are precomputed and stored with the patch in the data-structure `poly_tag` . Some other useful data such as the patch center and the major axes along which the patch is oriented are also precomputed and stored in `poly_tag`.

Results

The results of this give one a good idea of the amount of computation involved in the radiosity process if coherence is not exploited in one form or the other. The average time for one intersection cycle (intersection of one ray with all the polygons available), in this implementation and for the *Sitterson 365 model* dataset with 3959 patches is 4.5ms. This gives the effective time for a single ray-polygon intersection as 1.13 μsec. The current implementation does not make use of ray-tracing acceleration techniques such as bounding volume hierarchies, object bounding volumes, generalized rays etc. It should be possible to accelerate these times further using these techniques. This will be discussed in Chapter 5.

The rate of energy distribution in sampling with different numbers of rays for the

Sitterson 365 office model with 3959 patches is shown in figure 4.3. Shooting a larger number of rays drops the unshot energy of the brightest patch more rapidly than does shooting with a fewer number of rays. The reason is that sampling the environment coarsely causes small polygons to be missed, and the energy that is transferred to the polygons that are hit is more than their due share. Thus, the brightest patch in the environment is likely to be brighter than what it really should be.

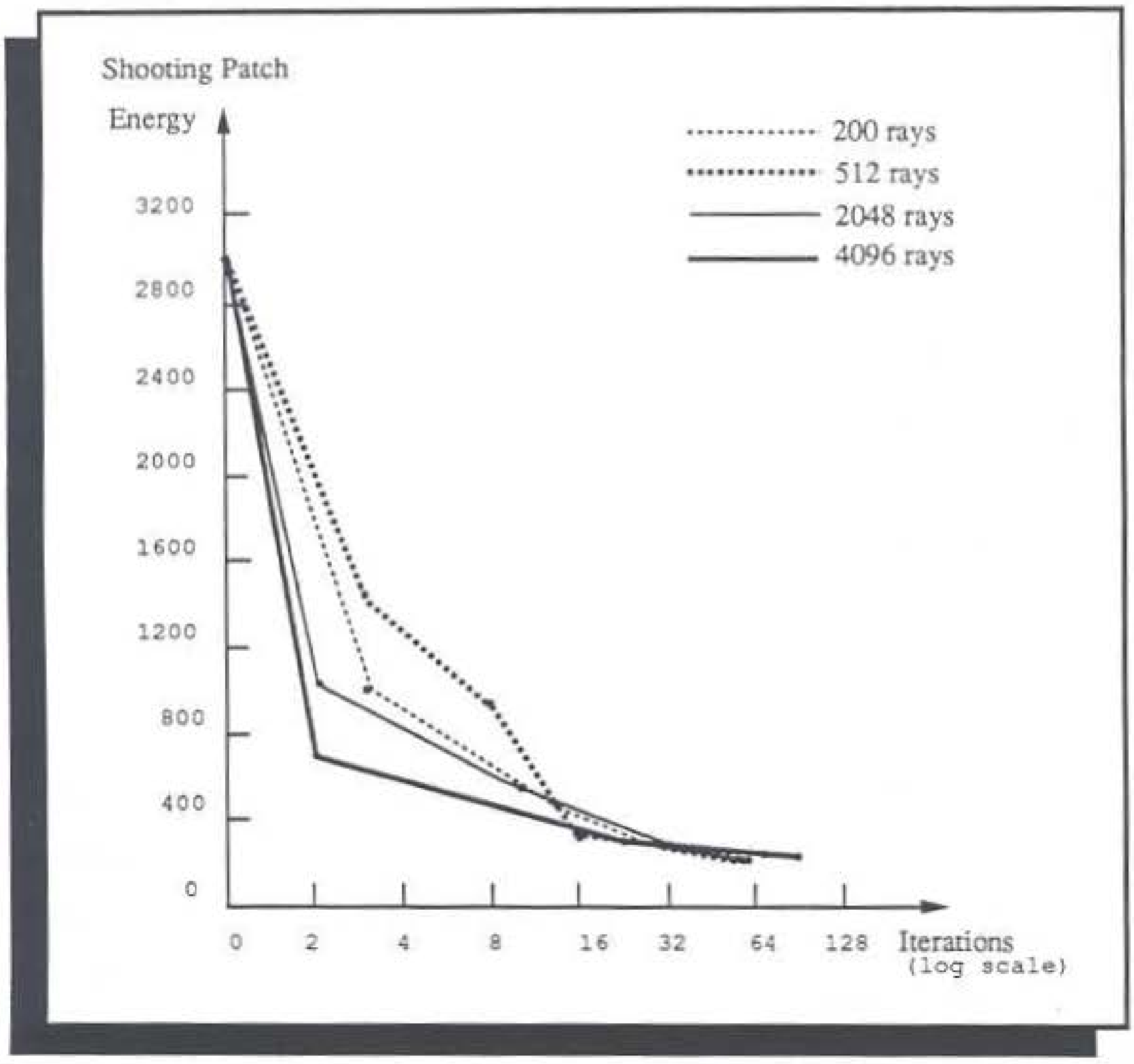


Figure 4.3 The Convergence for different number of rays in the Balanced-Load Ray-Casting Approach

A criticism of this approach could be that whereas the intersections are computed in parallel, finding the patch that intersected the ray and updating that patch's radiosity energy is done once per every ray. Thus, over the whole sequence of iterations these steps proceed sequentially. If multiple rays are fired simultaneously (one per processor) and made to circulate around the processors to determine the patch with which they intersect, then this would be a totally parallel approach. There would be no stages where any processor would be idle. A similar strategy could be used for energy distribution too. These ideas have been implemented for the approach described in subsection 4.2.3, but they have not been implemented for this approach. The reason was that most of the time with the current MasPar configuration is not taken up in these stages, but in the ray-polygon intersection routine. That routine is totally parallelized already and implementing the ideas suggested as above would not have changed the results much.

4.2.2 A Balanced-Load Environment-Projection Approach

In this approach, the patches are uniformly distributed over the processor array. The form-factors are computed by projecting the environment on to the shooting patch and then z-buffering the projected patches.

Outline

After the patches are distributed evenly over the processor mesh, the shooting patch is found as described in Section 4.2.1. This information about the shooting patch selection is then made available to all the other processors. Every patch in the environment is projected on the single-plane corresponding to the current shooting patch. This operation is totally parallel, being limited by the number of processors available. After this, z-buffering of the resulting projections needs to be done to determine which patch is actually visible from a given pixel on the single-plane.

To parallelize the z-buffer operation, the single-plane is mapped on the processor mesh in a hierarchical fashion. Thus, neighboring pixels on the single-plane fall onto either the same processor (if the resolution of the single-plane is greater than the number of processors available) or onto the immediately neighboring processor. This mapping has been chosen to exploit the coherency expected among the nearby pixels on the single-plane.

The projected patches now have to be scan-converted and z-buffered on this mapping of the single-plane over the processor mesh. For this, each projected patch is sent to the processor corresponding to the lower-left corner (minimum x, minimum y) of its projection's bounding-box on the single-plane. After this, each patch is spread out in both the dimensions using the neighbor-to-neighbor inter-processor communication primitives. This is shown in fig 4.4. During this phase, appropriate buffering is required to store multiple patches that might be reaching a particular pixel on the single-plane. After the spread-out operation is complete, z-buffering is done to determine the patch *seen* by a given pixel. For speed, and to reduce the number of patches that need to be buffered for a given pixel, we can make the z-buffer operation proceed concurrently with the spreading-out of the patch along the second dimension. This is detailed in the *Implementation Notes* below.

An alternative strategy for scan-conversion could have been to do global placement of the item-buffers instead of performing positioning by spreading out the polygons locally. In this strategy, the processor having the polygon to be projected computes the item-buffer elements corresponding to the projection of the polygon on the projection-plane. This processor then routes the item-buffers directly to those processors that map on to the locations corresponding to these item-buffers. This is shown in fig 4.5. Although this obviates the need for scan-conversion in the fashion described before, it has two drawbacks. The first is the cost of the communication. The router would provide slower communication for this case than the local neighbor-to-neighbor commmunication. Secondly, results reproduced in *Results* below indicated that the imbalance in this approach is high. The processors whose polygons cover a large fraction of the projection plane become heavily loaded while there are some processors that do not have any load. Thus, this strategy was not used in the final implementation.

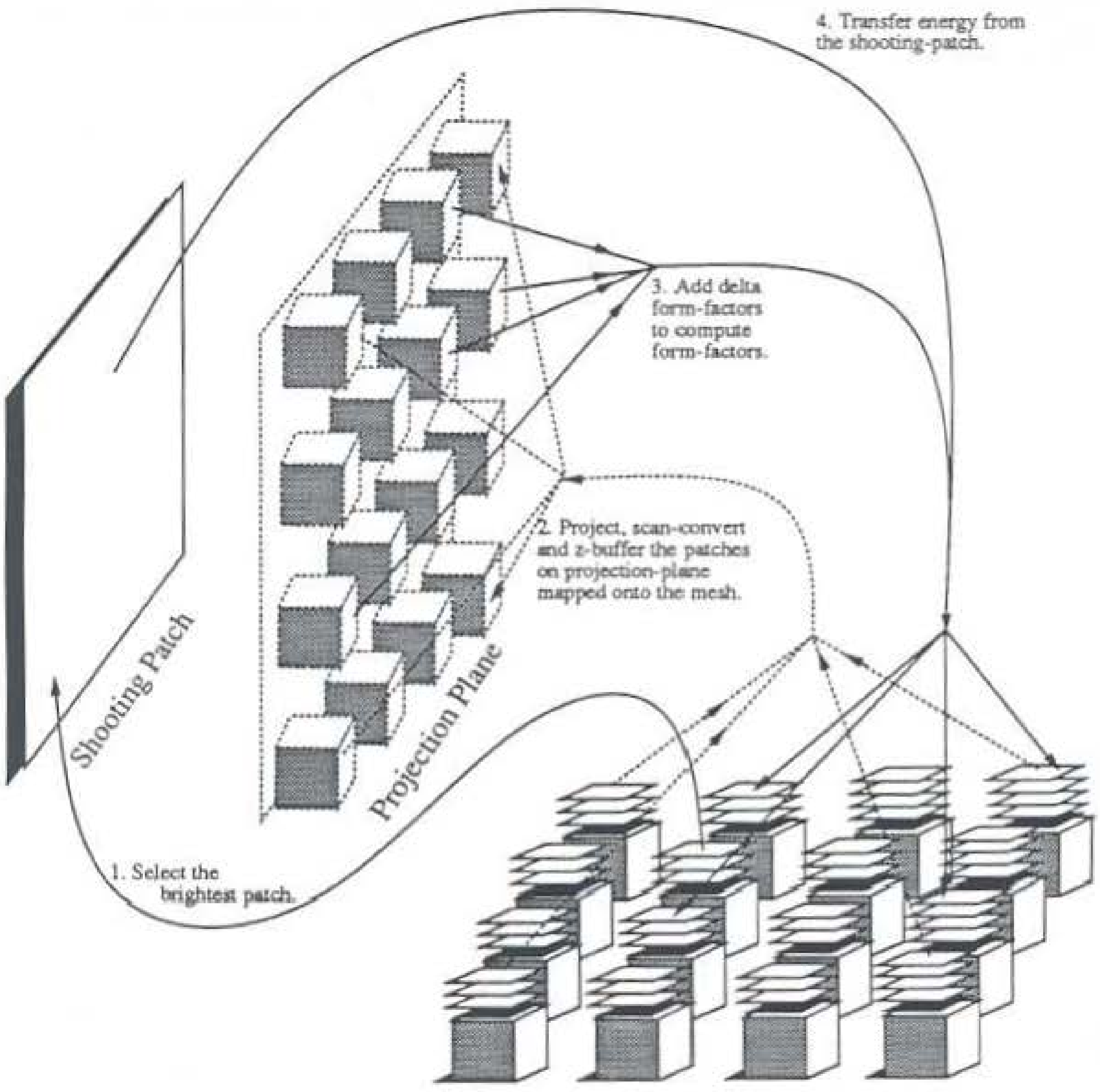
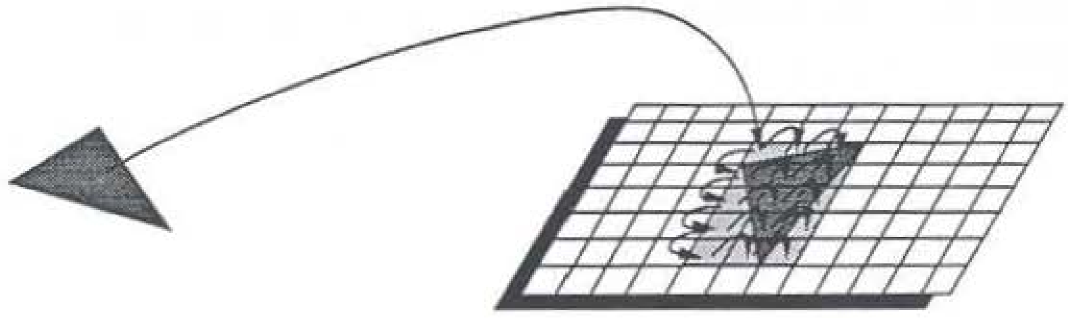

4. Transfer energy from the shooting-patch.
3. Add delta form-factors to compute form-factors.
2. Project, scan-convert and z-buffer the patches on projection-plane mapped onto the mesh.
Shooting Patch
Projection Plane
1. Select the brightest patch.

Figure 4.4 The Balanced-Load Environment-Projection Approach

Once the z-buffering is done, the delta form-factor corresponding to the pixel is routed back to the processor containing the patch that is *seen* by this pixel. The sum of all such delta form-factor values corresponding to a given patch gives the contribution of energy that will be received by it.

The energy-distribution phase is quite simple. Each patch receives energy proportional to its form-factor value and the radiosity of the shooting patch.


Scan Conversion by Local Placement

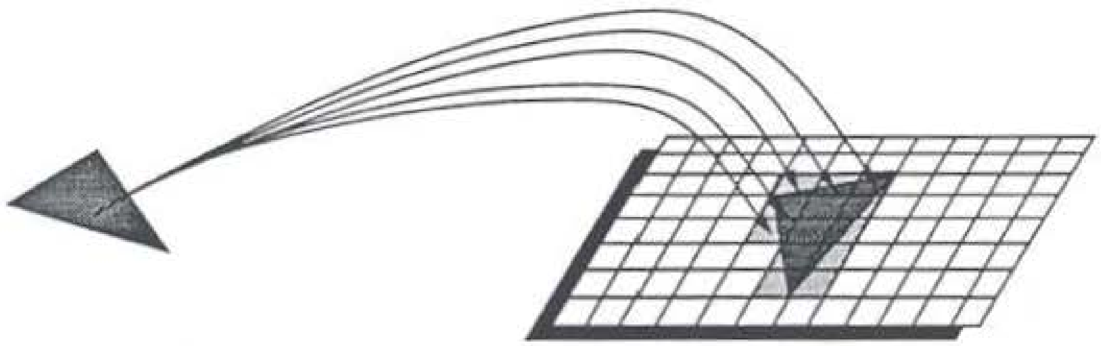

Scan Conversion by Global Placement

Figure 4.5 Scan-Conversion on the Projection-Plane

A study was also done within this approach to investigate the fraction of the form-factors that are reused as the radiosity solution proceeds. These are summarized graphically in fig 4.7 which appears in the subsection 4.2.2.

Implementation Notes

After patchification and conversion to the `.mp` file format, as described in the subsection 4.2.1, the patches are allocated to the DPU in a uniform fashion.

During the initialization phase, the matrix that would transform from the world-coordinate system to the patch's local-coordinate system is computed and stored with every patch. This transformation matrix of a patch is used while projecting the environment on that patch.

Given that the mapping of single-plane to the processors is static, the delta form-factor of each of the pixels is fixed throughout the radiosity solution. Thus, these too are precomputed and stored during the initialization phase.

The shooting patch is determined as in subsection 4.2.1 by a global `reduceMax()` operation over the shooting energies of all the patches. This is followed by transformation of every patch using the transformation matrix of the shooting patch.

In our implementation, the mapping of the single-plane to the processor mesh is done to maintain an orthogonal and monotonic correspondence between the x and y in the single-plane space and the system-defined constants *ixproc* and *iyproc* that define the location of a processor in the processor space. Thus, if $\mathcal{R}$ is the single-plane space and $\mathcal{P}$ is the processor space then this mapping $\mathcal{F} : \mathcal{R} \rightarrow \mathcal{P}$, is such that

$\forall x_1, x_2 \epsilon \mathcal{R} \ni x_1 \leq x_2, \mathcal{F}(x_1) \leq \mathcal{F}(x_2), (\mathcal{F}(x_1), \mathcal{F}(x_2) \epsilon \mathcal{P})$

Given the manner of numbering of *ixproc* and *iyproc*, this means that what corresponds to the lower-left corner on the single-plane space now corresponds to the upper-left corner on this processor space. After the patches are routed to the upper-left corner (by the `router` command), they are spread-out first along the *south* (increasing y) by a sequence of `xnet` commands to give single-processor thick strips. Each of these strips (corresponding to one patch), are then spread-out along *east* (increasing x) again by `xnet` commands. The pixels that fall within the bounding-box but outside of the actual projection are not considered for z-buffering.

To conserve space and speed, storing of the item-buffers is done only during the south-spread. During the spread in the east direction, z-buffering is done on the fly as the item-buffer tuples travel across the processor mesh.

Delta form-factors are added together using the combining send provided by the `sendwithAdd` library function.

Results

The results for this approach on the *365 Sitterson office* model with 3959 patches are summarized graphically in figure 4.6. These are summarized in terms of number of iterations and up to a 95% convergence of the solution. The single-plane used had a side-to-height ratio of 3, allowing 91.7% of the shooting-patch's energy to be shot out per iteration.

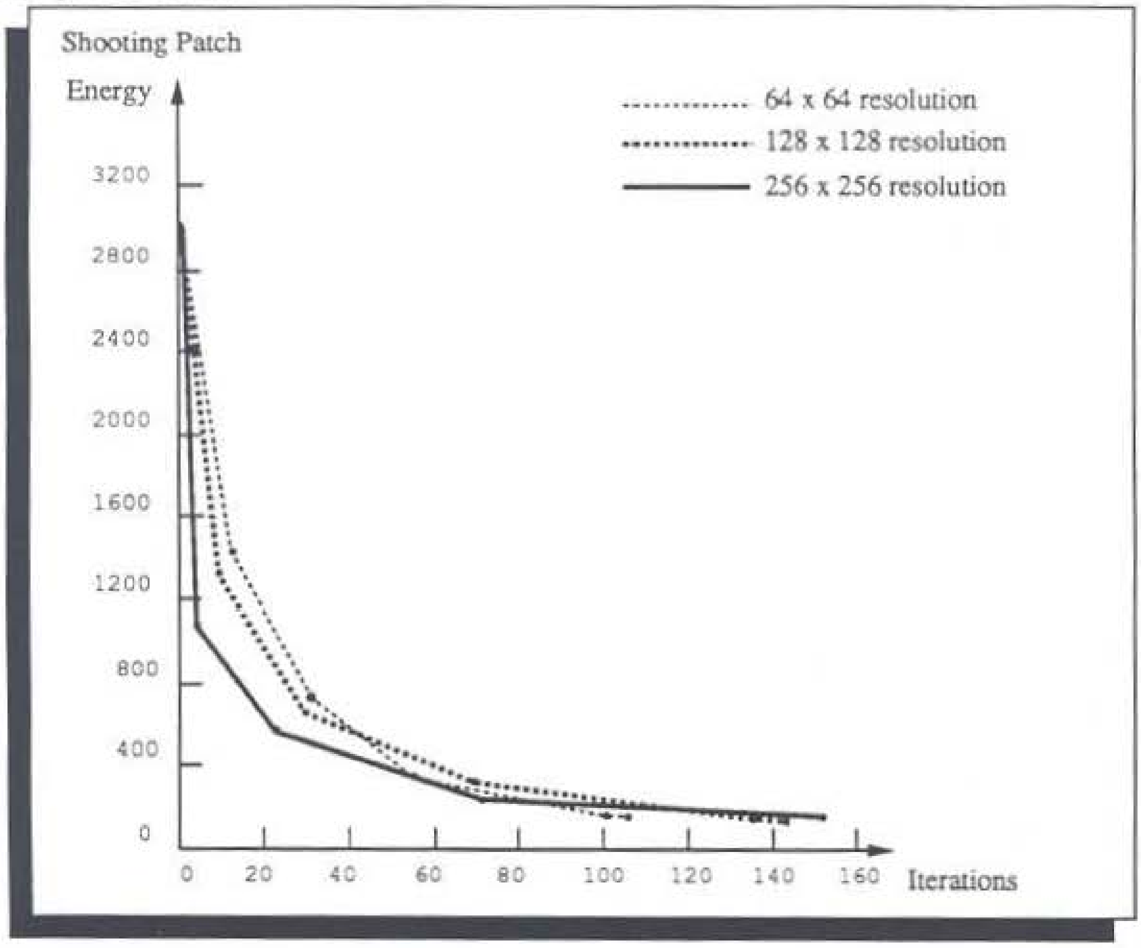


Figure 4.6 The Convergence for different Single-Plane resolutions for Balanced-Load Environment-Projection Approach

The rates of energy distribution in sampling with different resolutions of the single-plane follow the same pattern, in general, as those for energy distribution rates with different number of rays cast. Using a finer projection-plane resolution causes the unshot energy of the brightest patch to decrease faster than it does with a coarser resolution. The reason however is the opposite as that for the previous approach. In this approach, since polygons are being z-buffered, the small polygons that are in front of other polygons would not be missed. However, if the resolution is coarse enough, they would now be covering the whole pixels on the projection-plane whereas they should have been covering only fractions of these. Thus, these polygons get a higher share of energy than is due to them. This energy imbalance is corrected as the resolution becomes finer.

For this approach, the times for each iteration varied depending upon the resolution of the single-plane being used. The following times are for the *365 Sitterson office* model with 3959 patches. For a single-plane with 64×64 resolution, the average time per iteration is 0.24 secs. For a single-plane with 128×128 resolution, the average time per iteration is 0.45 secs. For a single-plane with 256×256 resolution, the average time per iteration is 2.15 secs.

The global positioning method for scan-conversion on a 4K resolution single-plane with a side-to-height ratio of one, yielded a total of 27K item-buffer elements. The maximum number of item-buffers on a single processor was 1.4K and the minimum was zero.

Figure 4.7 shows the results of the study done to ascertain the number of form-factors being reused. This was done on the *Sitterson 365 office* model with 3959 patches. As can be seen from the table, most form-factors are not reused till a very late stage in the iteration process. By then the convergence is almost complete and advantage if any to be gained from storing the form-factors are minimal. A small cache for storing the form-factors might be useful. However, since for most patches, the chance to reshoot energy comes only after several other patches have shot their energy in between, the size of the cache will have to be much larger than the number of form-factors reused. For instance, from the table it is clear that to reuse 6 form-factor rows, one would need to store 177 such entries.

Iteration	No of form-factors reused	Convergence %
13	2	53.03
32	3	76.57
59	3	88.76
101	3	94.40
177	6	97.20
392	39	98.61
1105	216	99.30
2462	810	99.65

Figure 4.7 Form-Factor Reuse

4.2.3 An Object-Space Ray-Casting Approach

This approach has been studied in a class project for the Highly Parallel Computing course in Fall 1990 by Varshney and Good [Varshney90].

Outline

Here a hierarchical spatial subdivision approach has been considered. The dataset is subdivided into some number of *virtual cells*. At the boundaries of each virtual cell, there are *virtual walls*. The aim here is to take advantage of the spatial coherence of the radiosity method: each surface interacts *mostly* with the nearby surfaces. These clusters of nearby surfaces should then be mapped to processors that are near each other on the processor mesh to minimize the communication costs. Each virtual wall must keep a record of information on light emanating from its virtual cell, and pass it to the appropriate neighboring cell.

To generate virtual cells that are approximately load-balanced, the orthogonal-load-balancing strategy is used [Dippé84]. Other strategies, such as general-cube subdivision and tetrahedron subdivision, fail to guarantee that the polygons would not need to be subdivided further during the cell-formation stage. The orthogonal-load-balancing scheme, however, does not result in a very good load balance because there is no local control of subdivision. Thus, if there is a dense cluster of polygons then the scheme would force all the cells that lie along the same orthogonal coordinates along the three axes to be narrow. This could cause some of such cells to receive fewer than the average number of polygons. This is observed in the load balancing results for this approach. An alternative could have been to use the one-dimensional version of this approach and divide up the dataset into *slabs* of varying thickness. That would have given a good load balance but could have potentially resulted in a higher amount of interactions that are directed outside of such a cell.

Further refinement of the balance for these load distributions is done by subdividing the polygons in the cells which are lightly loaded. This was admittedly a debatable decision. In retrospect, it seems more appropriate to subdivide the polygon on the basis of the light gradient across the parent polygon than on the basis of any load balancing scheme being used.

The virtual cells can be assigned as one cell per PE or per group of PEs. For highly parallel architectures such as the MasPar MP-1, if the one-cell-per-PE approach is taken then it would result in a large number of cells with relatively few polygons per cell. This would cause most of the interactions of a polygon to take place outside of the cell containing it. This undermines the advantages to be gained from localization of the interactions in a spatial-subdivision approach. Thus, each cell is assigned over a group of PEs. This also alleviates the load balancing problem to a certain extent, as within the group of PEs to which a cell is assigned, optimal load balancing is trivial.

Once the assignment of the patches to the processors is done, the radiosity iterations for each cell proceed in parallel. For the purposes of capturing the inter-cell interactions, each cell is bounded by virtual walls. These walls are called *virtual* as they do not form a part of the dataset but are used by the algorithm to just store and propagate any energies that are incident on them. The rays that do not intersect the polygons in the current cell are intersected with the virtual walls and their energies are stored at the virtual wall with which they intersect. After local convergence within the cells, the energy-laden virtual walls are exchanged between neighboring cells to facilitate inter-cell interactions. Each cell then distributes the energy thus obtained amongst its patches. After this, the local intra-cell iterations again start.

Thus, the whole process can be thought of as alternating local and global iterations of the radiosity process.

Implementation Notes

After patchification and conversion to the `.mp` file format, the patches are allocated to the DPU in a uniform fashion. The subdivision of polygons is done to a global grid that is finer than the number of cells desired along that axis. In the sample dataset for instance, the spatial subdivision was $8 \times 8 \times 1$ along x, y, and z axes respectively. The global grid used then was such that it partitioned the model into $16 \times 16 \times 2$ sub-cells. The sum of polygons in each of the 16 strips along the x axis is computed first. Then these 16 strips are coalesced as evenly, in number of polygons, as possible into 8 groups. The boundaries of these groups correspond to the virtual walls along the x axis. The y and z axes are divided similarly. Each virtual cell is then sent to a row of the data-parallel unit of the MasPar as shown in the figure 4.8 The $8 \times 8 \times 1$ subdivision yields one virtual cell per row of the MasPar. The patches within each cell are distributed evenly across all the processors of the corresponding row.

Each processor maintains a partial form-factor matrix for the virtual cell on its row. Each row of the matrix corresponds to how much each patch can *see* of the other patches. Thus, the partial matrix has roughly one by sixty four of the rows and all the columns of the virtual cell's form-factor matrix.

Each processor shoots rays from each of its patches and sends them to the neighboring processor in the same row. Each ray is intersected with all the local patches and then passed to the next processor. After cycling through all the 64 processors in the row, the ray has been intersection-tested against every patch in the cell. The identification of the patch intersected by the ray is recorded in the ray's structure and the appropriate form-factor matrix entry on the ray's originated processor is updated. If the ray did not intersect any patch, then it is tested against the six virtual walls of the cell and is added to the buffer for the appropriate wall.

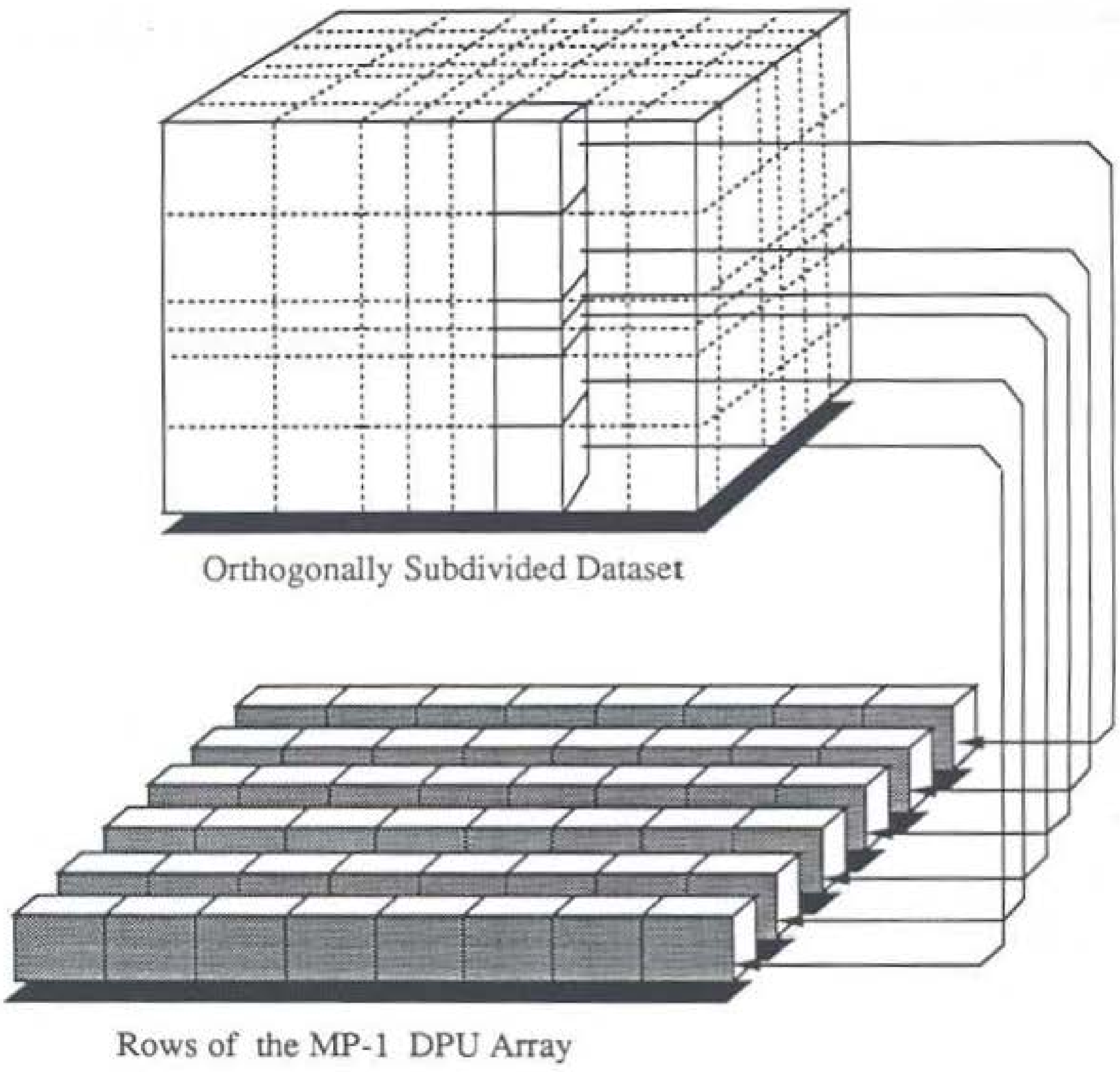


Figure 4.8 Mapping of Virtual Cells to Processors

Once the form-factors are calculated, the energy is distributed. Each processor selects its brightest patch (that with the most unshot energy) and creates an energy packet. That packet is then cycled through all the processors of that row. As each processor receives a packet, it uses its partial form-factor matrix to update the energies of all its local patches. Once the cycle is completed, new shooting patches are chosen and the cycle is repeated. When the unshot energy of the brightest patch is below a threshold value, the process ends. This is the *patch-to-patch* energy distribution stage.

After the patch energies have stabilized, the energy for each ray stored at each virtual wall (from the form-factor calculation stage), is updated based on the energy of the ray's originating patch. This is the *patch-to-wall* energy distribution stage.

Once the ray energies at the walls have been updated, the neighboring cells need to be updated with these energies. Each virtual wall's ray buffer is sent to the row corresponding to the virtual cell on the other side of the wall. This the *wall-to-wall* energy exchange stage.

The wall-rays are propagated through all the polygons on the row much as in the form-factor calculation step, except that patch energies are updated now (instead of testing for ray-patch intersection). This is the *wall-to-patch* energy distribution stage.

After the wall-ray energies have been redistributed to the patches, the energy distribution process repeats with patch-to-patch distribution.

This implementation has been done up to the stage of form-factor computation. It has not been carried out further since the results of this stage indicated that this approach is not well-suited to the problem.

Results

The sample dataset for this approach was the *365 Sitterson office* model. This was divided by a global grid to $8 \times 8 \times 1$ subdivisions along the x, y, and z axes, yielding a total of 64 cells. During the form-factor calculation stage, one ray per patch was fired (for test purposes). Of a total of 3336 rays thus fired, 1983 of these intersect the patches within the cell from which they originated. The remaining 1353 rays had to be stored at the virtual wall buffers. This indicated that our assumption that most of the interactions could be expected to be localized within each cell was wrong and that as much as 40% of all interactions were actually directed outside of these virtual cells. This suggests that spatial subdivision to a global grid is not a good strategy for radiosity. Using PVS cells instead [Airey90a] for spatial subdivision might be a better approach. This is again discussed in Chapter 5.

4.3 Discussion of Results

Let us first compare the two balanced-load approaches: ray-casting and environment-projection. The ray-casting approaches take up fewer iterations for a given convergence than do the environment projection techniques. However, the time per iteration is substantially more for the ray-casting approaches. For instance, for the dataset of *365 Sitterson office* studied, the time per intersection cycle is 4.5 ms whereas the iteration time for a 128×128 resolution single-plane is 450 ms. This allows the ray-casting approach only 100 rays per iteration before it starts becoming slower than the environment-projection approach, clearly an order of magnitude coarser sampling.

As is evident from the results given previously, balanced-load environment-projection approach described above appears promising. As suggested in [Airey89], it would make more sense to switch to a coarse ray-casting approach from the high-resolution environment-projection approach in the later iterations. In later iterations, the unshot energies left with the shooting patches are small enough that the high resolution of the single-plane is of much consequence. Alternatively, the resolution of the single-plane could be adaptively varied depending upon the energy of the shooting patch.

The basic difficulty in using the object-space based polygon-processor mappings has been the load imbalance of the resulting distribution and the fewer local radiosity interactions than expected.

At this point it would be interesting to assess the object-space environment-projection approach that has not been implemented. One way to implement this could be by using a spatial subdivision technique such as the one used for the object-space ray-casting approach described above. Instead of storing rays at the virtual walls, however, one could store the entire projection-planes. Intra-cell local radiosity iterations could then be followed by inter-cell global radiosity iterations. During the global iterations, these projection-planes could be exchanged across the neighboring cells. Two main issues would need to be addressed in following such an approach. The first one is the problem of correcting the load-imbalance that has been described in subsection 4.2.3. The second one is the problem of large amounts of interprocessor communication involved. Assuming a 128×128 resolution single-plane, one would have to store one such plane for each of the six virtual walls (to preserve the direction information of incident energy). Assuming that form-factors are being stored as 4-byte floating-point numbers, this would require 384Kbytes of memory per cell. On the MasPar implementation, if a cell is stored across a row of the DPU array, this would require 6Kbytes of memory per PE. This much information would need to be exchanged with other PEs at the end of each global iteration. As observed from the results of subsection 4.2.3, we should therefore be expecting a *high* share of all iterations to be global. The overall process would then be *slow*. Some other alternatives to this approach are discussed in Chapter 5

In brief then, of the three approaches implemented, the balanced-load environment-projection approach appears most promising.

Chapter 5

Contributions and Future Work

This thesis has surveyed three parallel radiosity techniques for mesh-connected SIMD computers. The contributions of this thesis include:

- Formulation of a balanced-load environment-projection radiosity approach for mesh-connected SIMD computers.
- A relative comparison of three parallel radiosity techniques on mesh-connected SIMD computers.
- A classification of possible parallelism levels in radiosity.
- Spatial subdivision without considering visibility has been observed to violate the locality property of radiosity.
- Form-factor reuse study shows a low reuse of form-factors while a high convergence is achieved.

This area is so rich that there is ample scope for future work both within and outside of these approaches. A brief overview of these possibilities is provided next.

5.1 Future Work

5.1.1 Potentially Visible Cells

Following the ideas of Airey [Airey90a], one could consider potentially visible cells as the basis for spatial subdivision rather than subdivision to a global grid. To describe the concept of potentially visible cells by an example: rooms in a building are prime candidates for potentially visible cells for its model. Considering potentially visible cells promises minimal light interaction across cells, thus removing the main problem of excessive inter-cell communication that was observed for the object-space ray-casting approach in Section 4.2.3. Besides, this also ties in very nicely with the ideas of incremental radiosity calculations and interactive walkthroughs of the architectural datasets. The incremental radiosity techniques study the changes in the radiosity shading with changes to the dataset. Most of these changes are local, such as moving a chair and observing its shadow change. Therefore, it makes sense to exploit this locality property by considering only the polygons that fall within the current cell. As far as interactive walkthroughs are concerned, the idea of potentially visible cells is already being used to determine the polygons to be displayed. This idea can be extended to consider this set of polygons for radiosity calculation purposes.

An approach similar to that used in Section 4.2.3 could then be used. Local iterations would proceed for the cell in which the user is currently in and once the solution in the current cell refines to suitable levels, a global iteration could be done. Energy across cells could now be stored on the open portals for the cell instead of storing them on the six virtual walls as done in Section 4.2.3. When a user exits a cell and enters a new cell, the polygons on the processor grid corresponding to the current cell could be swapped out, those corresponding to the new cell be swapped in. The polygons corresponding to the portals between the two cells would remain on the processor mesh.

Any one of the approaches studied in this thesis would be a viable alternative for computing the radiosity solution within each such potentially visible cell.

5.1.2 Accelerated Ray-Tracing

Extensions to the ray-casting approaches in this thesis could be made by incorporating one or more of the several well known ray-tracing techniques. Thus, one could use bounding volume hierarchies, object bounding volumes, generalized rays etc. A implementation similar to [Delany88] could also be studied for mesh-connected SIMD computers. Rough calculations are however not very promising. From the results given in that paper, a rate of roughly 0.3 ms per ray intersection cycle is feasible using that approach on a 16K CM-1. For a 128×128 projection-plane, that would give roughly 5 seconds per iteration, which is by itself an order of magnitude costlier than the balanced-load environment-projection approach considered in this thesis. However, these are just hypothetical figures, and these should be checked out against an actual implementation, now that fast virtual sorting and routing functions are available on the MasPar MP-1 [Prins90].

5.1.3 Coherency in Scan-Conversion

There exists scope for exploiting more coherency in the scan-conversion of the polygons in the environment-projection approach studied in Section 4.2.2. Right now, while the polygon is being spread out along the second dimension, no use is made of information about the other polygons that have been spread out before. Consider for instance, a polygon, say A, that is smallest in z (and therefore will be finally chosen) already spread out and another polygon, say B, that is within the extents of A, and has yet to be spread out. Then after comparing the z-extents of the two polygons it should be possible for one to discard B before spreading it out and save this extra work. This is an extreme example, but still there are several cases in which using the information of already spread out polygons, one could speed up the scan-conversion of the current polygon. However, trying to take all these special cases into account would also reduce the homogeneity - the very basis of all efficient SIMD algorithms - in this approach, and it remains to be seen how these expected gains balance off the losses arising from the new irregularities introduced.

Bibliography

[Airey90a] Airey, J.M., "Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculation", PhD Dissertation, 1990, Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, N.C.

[Airey90b] Airey, J.M., J.H. Rohlf and F.P. Brooks, Jr., "Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments" *ACM Computer Graphics (Proceedings 1990 Symposium on Interactive 3D Graphics)* ,Vol. 24, No. 2, pp 41-50.

[Airey89] Airey, J. and M. Ouh-young, "Two Adaptive Techniques Let Progressive Radiosity Outperform the Traditional Radiosity Algorithm", Department of Computer Science, University of North Carolina at Chapel Hill, TR89-020.

[Almasi89] Almasi, G.S. and A. Gottlieb, "Highly Parallel Computing", *The Benjamin/Cummings Publishing Company Inc.*, Redwood City, California, 1989.

[Baum90] Baum D.R. and J.M. Winget, "Real Time Radiosity Through Parallel Processing and Hardware Acceleration", *ACM Computer Graphics (Proceedings 1990 Symposium on Interactive 3D Graphics)*, Vol. 24, No. 2, pp 67-76.

[Brooks88] Brooks, F.P., Jr., "First Annual Technical Report Walkthrough Project", Department of Computer Science, University of North Carolina at Chapel Hill, TR88-035.

[Chen90] Chen S.E., "Incremental Radiosity: An Extension of Progressive Radiosity to an Interactive Image Synthesis System", *ACM Computer Graphics (Proceedings SIGGRAPH '90)*, Vol. 24, No. 4, pp 135-144.

[Chen89] Chen S.E., "A Progressive Radiosity Method and its Implementation in a Distributed Processing Environment", Master's Thesis, Program of Computer Graphics, Cornell University, Ithaca, N.Y., Jan 89.

[Cohen88] Cohen, M.F., S.E. Chen, J.R. Wallace and D.P. Greenberg, "A Progressive Refinement Approach to Fast Radiosity Image Generation", *ACM Computer Graphics (Proceedings SIGGRAPH '88)*, Vol. 22, No. 4, pp 75-84.

[Cohen86] Cohen, M.F., D.P. Greenberg, D.S. Immel and P.J. Brock, "An Efficient Radiosity Approach for Realistic Image Synthesis", *IEEE CG&A*, Vol. 6, No. 2, pp 26-35.

[Cohen85] Cohen, M.F. and D.P. Greenberg, "The Hemi-Cube: A Radiosity Solution for Complex Environments", *ACM Computer Graphics (Proccedings SIGGRAPH '85)*, Vol. 19, No. 3, pp 31-40.

[Delany88] Delany, H.C., "Ray Tracing On A Connection Machine", 1988 International Conference on Supercomputing Proceedings, St. Malo, France, July 4-8, 1988, ACM Press, pp 659-664.

[Dippé84] Dippe, M., and J. Swensen, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis", *ACM Computer Graphics*, Vol. 18, No. 3, July 1984, pp 149-158.

[Drettakis90] Drettakis, G., E. Fiume and A. Fournier, "Tightly Coupled Multiprocessing for a Global Illumination Algorithm", *Eurographics 90*, Proceedings of the European Computer Graphics Conference and Exhibition, Montreux, Switzerland, September 4-7, 1990, pp 387-398.

[Glassner90] ed. Glassner, A. "Graphics Gems", *Academic Press, Inc.*, San Diego, California, 1990.

[Good91] Good, H.R., Personal Communication.

[Goral84] Goral, C.M., K.E. Torrance and D.P. Greenberg, "Modeling the Interaction of Light Between Diffuse Surfaces", *ACM Computer Graphics (Proceedings SIGGRAPH '84)*, Vol. 18, No. 3, pp. 213-222.

[Maspar90] "MasPar MP-1 Standard Programming Manuals", *MasPar Computer Corporation*, Sunnyvale, California.

[Prins90] Prins, J.F. and Smith J.B., "Parallel Sorting of Large Arrays on the MasPar MP-1", *Proceedings, Third Symposium on Frontiers of Massively Parallel Computation*, College Park, MD, IEEE 1990.

[Recker90] Recker, R.J., D.W. George and D.P. Greenberg, "Acceleration Techniques for Progressive Refinement Radiosity", *ACM Computer Graphics (Proceedings 1990 Symposium on Interactive 3D Graphics)*, Vol. 24, No. 2, pp 59-66.

[Varshney90] Varshney, A. and H.R. Good, "Highly Parallel Radiosity Using Spatial Subdivision", Class Project Report *Highly Parallel Computing*, Comp 290-2, Fall 1990.

[Xu89] Xu, H., Q. Peng and Y. Liang, "Accelerated Radiosity Method for Complex Environments", *Eurographics 89*, Proceedings of the European Computer Graphics Conference and Exhibition, Hamburg Federal Republic of Germany , September 4-8, 1989, pp 51-59.

Appendix A

Source Code

patchify.c

```
/**************************************************************************/

          Amitabh Varshney                    Howard Good

/**************************************************************************/
 This program patchifies the input dataset into desired size patches
 The patches can be specified on a global grid with possibly all the
 three dimensions specified. This grid is aligned about the three
 principal axes.
/***************************************************************************/

#include <stdio.h>
#include <math.h>
#include <signal.h>
#include "host.h"
#include "error.h"

#define   MAXVERT 20 /* redefine MAXVERT */

double PatchSide[3]; /* Stores the three sides of the global grid*/


/***********************************************************************
                      Option Handler

   Handles all the options for the program specified in command line
   The options are '?' : learn the usage
                   'S' : have all the grid sides as equal
```

```
    'C' : have all the grid sides specified

*********************************************************************/
int option_handler(ac, av)
int  ac;
char *av[];
{ register int          i, ok = 1;
  register char         *c;

  /* Take the default patchside as the CELLLENGTH consts from host.h */
  PatchSide[X] = XCELLLENGTH;
  PatchSide[Y] = YCELLLENGTH;
  PatchSide[Z] = ZCELLLENGTH;

  for (i = 1; i < ac && av[i][0] == '-'; i++)
      for (c = &(av[i][1]); *c; c++) switch (*c) {
         case '?':
              fprintf(stderr, "S D  { Equal Sized Patchsize }\n");
              fprintf(stderr, "C D D D { Cuboidal Patchsize }\n");
              fprintf(stderr, "fname
{ < fname.poly > fname.patch }\n");
              break;
   case 'S':
     PatchSide[X] = PatchSide[Y] = PatchSide[Z] = atof(av[++i]);
     break;
 case 'C':
     PatchSide[X] = atof(av[++i]);
     PatchSide[Y] = atof(av[++i]);
     PatchSide[Z] = atof(av[++i]);
     break;
         default:
             fprintf(stderr, "%s: unknown option -%c\n", av[0], *c);
             ok = 0;
             break;
   }
   fprintf(stderr,"Patchsides being used X %f Y %f Z %f \n",
   PatchSide[X], PatchSide[Y],PatchSide[Z]);
   return (ok ? i : 0);
}


/*********************************************************************
                         write_patch

   Writes out a patch to the patch file fp being constructed
   The header specifies things like patch colors, number of verts
   etc. and vertices is an array containing all the vertices.
********************************************************************/
write_patch(fp,header,vertices)
FILE* fp;
int header[10];
float vertices[MAXVERT][3];
{ int k,n;

  n = header[6];
  fprintf(fp,"%d %d %d %d %d %d %d %d %d %d\n",
    header[0],header[1],header[2],header[3],header[4],
    header[5],header[6],header[7],header[8],header[9]);

  for (k = 0; k < n; k++)
    fprintf(fp,"%f %f %f\n",vertices[k][X],vertices[k][Y],vertices[k][Z]);
  fflush(fp);
}


/********************************************************************
                         patchification
   Writes out a patch to the patch file fp being constructed
   The header specifies things like patch colors, number of verts
   etc. and vertices is an array containing all the vertices.
   All the output patches are either triangles or quads.
 *******************************************************************/
void patchification(poly_fp,patch_fp)
FILE* poly_fp;
FILE* patch_fp;
{
   Poly p; /* Polygon being processed */
   int  i,j,jj,k,m,n; /* Miscellaneous counters */
   char  line[512]; /* Line from a poly file */
   int h[10]; /* Polygon header */
   int polysread = 0; /* Number of polygons read so far*/
   int pwritten = 0; /* Number of patches written so far*/
   int x0, x1, x2, x1i, x2i, x1dim, x2dim;
   float wb[MAXVERT][3]; /* Buffers to store the vertices */
   float new_verts[4][3];
   float ex[3*2];                   /* Min and max extents in 3D */
   float  clipbox[3*2];   /* Current clipping box for polygon*/
   float normal[3]; /* Plane normal */
   int orientation; /* Orientation of the polygon */
```

```
while (fgets(line, 511, poly_fp)) /* Read in the header line */
{ if(++polysread % 1000 == 0) fprintf(stderr,"Read poly #%d\n",polysread);
  if (sscanf(line,"%d %d %d    %d %d %d     %d %d %d %d\n",
     &h[0],&h[1],&h[2],&h[3],&h[4],&h[5],&h[6],&h[7],&h[8],&h[9]) != 10)
     die("patchification","bad header read",1);

 /* Determine the extents of the current polygon */
 ex[MINEX(X)] = ex[MINEX(Y)] = ex[MINEX(Z)] = HUGE;
 ex[MAXEX(X)] = ex[MAXEX(Y)] = ex[MAXEX(Z)] = -HUGE;

 for (i=0; i<h[6]; i++) /* h[6] is the number of vertices */
 { if (fscanf(poly_fp,"%f %f %f\n",&p.verts[i][X],&p.verts[i][Y],
        &p.verts[i][Z])!=3)
     die("read_poly_format","bad vert read",1);
   else
   { ex[MINEX(X)] = MIN(ex[MINEX(X)],p.verts[i][X]);
     ex[MINEX(Y)] = MIN(ex[MINEX(Y)],p.verts[i][Y]);
     ex[MINEX(Z)] = MIN(ex[MINEX(Z)],p.verts[i][Z]);
     ex[MAXEX(X)] = MAX(ex[MAXEX(X)],p.verts[i][X]);
     ex[MAXEX(Y)] = MAX(ex[MAXEX(Y)],p.verts[i][Y]);
     ex[MAXEX(Z)] = MAX(ex[MAXEX(Z)],p.verts[i][Z]);
   }
 }

 p.numverts = h[6];
 planeEq(&p,p.verts,p.numverts);

 if (h[7] > 0) /* this is an emitter polygon so do not split */
 {
   write_patch(patch_fp,h,p.verts);
   pwritten++;
   continue;
 }

 normal[X] = fabs(p.eq[A]);
 normal[Y] = fabs(p.eq[B]);
 normal[Z] = fabs(p.eq[C]);

 /* Determine the orientation of the polygon - ie the pair of
    orthogonal axes along which its projection would be maximum */

 if (normal[X] > normal[Y])
 { if (normal[X] > normal[Z]) orientation = SKEWX;
   else orientation = SKEWZ;
  }
  else
  { if (normal[Y] > normal[Z]) orientation = SKEWY;
   else orientation = SKEWZ;
  }

  if ((normal[Y] == 0)&&(normal[Z] == 0)) orientation = X;
  if ((normal[X] == 0)&&(normal[Z] == 0)) orientation = Y;
  if ((normal[Y] == 0)&&(normal[X] == 0)) orientation = Z;

  switch (orientation)
  { case X: case SKEWX: x1 = Y; x2 = Z; x0 = X; break;
    case Y: case SKEWY: x1 = Z; x2 = X; x0 = Y; break;
    case Z: case SKEWZ: x1 = X; x2 = Y; x0 = Z; break;
  }

  /* snap to global grid */

  x1i =  floor(ex[MINEX(x1)]/PatchSide[x1]);
  x2i =  floor(ex[MINEX(x2)]/PatchSide[x2]);
  x1dim = ceil(ex[MAXEX(x1)]/PatchSide[x1]) - x1i;
  x2dim = ceil(ex[MAXEX(x2)]/PatchSide[x2]) - x2i;

  clipbox[MINEX(x0)] = ex[MINEX(x0)];
  clipbox[MAXEX(x0)] = ex[MAXEX(x0)];

  clipbox[MINEX(x1)]= x1i*PatchSide[x1];
  clipbox[MAXEX(x1)]= clipbox[MINEX(x1)] + PatchSide[x1];

  /* Clip the current polygon into desired sized patches */
  for (i = 0; i < x1dim; i++)
  { clipbox[MINEX(x2)] = x2i*PatchSide[x2];
    clipbox[MAXEX(x2)] = clipbox[MINEX(x2)] + PatchSide[x2];
    for (j = 0; j < x2dim; j++)
    { if ((n = clip_face_to_box(clipbox[MINX],clipbox[MAXX],
clipbox[MINY],clipbox[MAXY],
clipbox[MINZ],clipbox[MAXZ],
&p,wb,orientation,ex)) > 2)
      { if (n <= 4)
        { h[6] = n;
  write_patch(patch_fp,h,wb);
          pwritten++;
        }
else
        { h[6] = 4;
      write_patch(patch_fp,h,wb);
              pwritten++;
              if (n%2 == 1)  k = n - 1;
              else  k = n;
              for (jj = 1 ; jj <= (k-4)/2; jj++)
              { for(m=0;m<3;m++)
                { new_verts[0][m] = wb[0][m];
                  new_verts[1][m] = wb[(jj*2)+1][m];
                  new_verts[2][m] = wb[(jj*2)+2][m];
                  new_verts[3][m] = wb[(jj*2)+3][m];
                }
      h[6] = 4;
                write_patch(patch_fp,h,new_verts);
                pwritten++;
              }
              if (n%2 == 1)
              { for(m=0;m<3;m++)
                { new_verts[0][m] = wb[0][m];
                  new_verts[1][m] = wb[(jj*2)+1][m];
                  new_verts[2][m] = wb[(jj*2)+2][m];
                }
      h[6] = 3;
                write_patch(patch_fp,h,new_verts);
                pwritten++;
              }
  }
       }
       clipbox[MINEX(x2)] = clipbox[MAXEX(x2)];
       clipbox[MAXEX(x2)] += PatchSide[x2];
      }
      clipbox[MINEX(x1)] = clipbox[MAXEX(x1)];
      clipbox[MAXEX(x1)] += PatchSide[x1];
    }
  }
  printf("Input polygons = %d\n",polysread);
  printf("Output patches = %d\n",pwritten);

}

/*****************************************************************
                          Main

*****************************************************************/
main(ac, av)
int ac;
```

```
char *av[];
{ int  options;
  FILE* poly_fp; /* Input poly file */
  FILE* patch_fp; /* Output patch file */
  char filename[128];

  if (!(options = option_handler(ac,av))) die("main","bad options",1);

  if (!(poly_fp = fopen(sprintf(filename,"%s.poly",av[options]),"r")))
      die("main","can't open input poly file",1);

  if (!(patch_fp = fopen(sprintf(filename,"%s.patch",av[options]),"w")))
      die("main","can't open output patch file",1);

  patchification(poly_fp,patch_fp); /* Carry out patchification */

  fclose(poly_fp);
  fclose(patch_fp);
}
```

```
/***********************************************************************
            Amitabh Varshney              Howard Good

***********************************************************************

  Patchtomp

  This program takes in a patchified dataset and converts it to
  a binary format file suitable for fast reading in by the MasPar MP-1
  This speeds up the set-up time on the MasPar. It also calculates some
  other parameters like the polygon equations and their areas before
  writing these out to the MasPar binary file.
 **********************************************************************/

#include <stdio.h>
#include <math.h>
#include "host.h"
#include "error.h"

/**********************************************************************
                         bufwrite

   Using block I/O this routine writes out data in the binary format
   into the file fp.
**********************************************************************/
void bufwrite(fp,ptr,size)
FILE*   fp; /* Output file */
char*   ptr; /* Data pointer */
int     size; /* Data size */
{ if (!fwrite(ptr,size,1,fp))
      die("bufwrite","bad write",1);
}

/**********************************************************************
                         cross_prod

  Returns the cross product of two vectors v1 and v2 into r
**********************************************************************/
void cross_prod(r,v1,v2)
float r[4], v1[3], v2[3];
{  r[X] =  v1[Y]*v2[Z] - v1[Z]*v2[Y];
   r[Y] = -v1[X]*v2[Z] + v1[Z]*v2[X];
   r[Z] =  v1[X]*v2[Y] - v1[Y]*v2[X];
}

/**********************************************************************
                          planeEq

  Given a set of vertices and in a plane, this routine finds
   out the equation of the plane.
***********************************************************************/
planeEq(cur_poly,verts,numv)
Poly* cur_poly; /* Polygon pointer */
int numv; /* Number of vertices */
float verts[MAXVERT][3]; /* Vertex array */
{ float eq[4], v1[3], v2[3];
  int i,j,k;
  float mag;

  i = 0;
  do
  { j = (i+1) % 3;
    k = (j+1) % 3;
    /* determine the two vectors in the plane */
    v1[X] = verts[j][X] - verts[i][X];
    v1[Y] = verts[j][Y] - verts[i][Y];
    v1[Z] = verts[j][Z] - verts[i][Z];

    v2[X] = verts[k][X] - verts[j][X];
    v2[Y] = verts[k][Y] - verts[j][Y];
    v2[Z] = verts[k][Z] - verts[j][Z];

    /* take their cross product */
    cross_prod(eq,v1,v2);
  }
  while((++i<numv)&&((fabs(eq[X])+fabs(eq[Y])+fabs(eq[Z])<L1_NORM_MIN)));

  if (i==numv)
  { fprintf(stderr,"planeEq(): bad face:\n");
      for (i = 0; i < numv; i++)
        fprintf(stderr,"%g %g %g\n",verts[i][X],verts[i][Y],verts[i][Z]);
    fflush(stderr);
    eq[X] = eq[Y] = eq[Z] = eq[W] = 0.0;
  }
  else /* normal has been found */
  { mag = sqrt(eq[X]*eq[X] + eq[Y]*eq[Y] + eq[Z]*eq[Z]);
    eq[X] /= mag;
    eq[Y] /= mag;
    eq[Z] /= mag;
    eq[W] = -(eq[X]*verts[0][X]+eq[Y]*verts[0][Y]+eq[Z]*verts[0][Z]);
  }

  /* Return back the equation as elements of the polygon passed in */
  cur_poly->eq[X] = eq[X];
  cur_poly->eq[Y] = eq[Y];
  cur_poly->eq[Z] = eq[Z];
  cur_poly->eq[W] = eq[W];
}

/**********************************************************************
                          area

  Given a polygon, this routine returns the area of the polygon
  This is done by considering the polygon to be made up of a number
  of triangles and finding the area of each triangle by halving the
  magnitude of the cross product of two of its sides.
**********************************************************************/
area(cur_poly)
Poly* cur_poly;
{ int i,j,k;
  float area = 0.0;
  float v1[3], v2[3], v3[4];

  for(i=0; i < cur_poly->numverts - 2; i++)
  { j = i+1;
    k = i+2;

    /* Determine the two vectors as the two sides of the sub-triangle */
    v1[X] = cur_poly->verts[j][X] - cur_poly->verts[i][X];
    v1[Y] = cur_poly->verts[j][Y] - cur_poly->verts[i][Y];
    v1[Z] = cur_poly->verts[j][Z] - cur_poly->verts[i][Z];

    v2[X] = cur_poly->verts[k][X] - cur_poly->verts[i][X];
    v2[Y] = cur_poly->verts[k][Y] - cur_poly->verts[i][Y];
    v2[Z] = cur_poly->verts[k][Z] - cur_poly->verts[i][Z];

    /* Take their cross product */
    cross_prod(v3, v1, v2);

    area += 0.5*sqrt(v3[X]*v3[X] + v3[Y]*v3[Y] + v3[Z]*v3[Z]);
  }
  cur_poly->area =  area;
}

/***********************************************************************
                           io_polys

  Given an ascii file of polygons, this routine reads in
  polygons, finds their plane equations and areas and writes them out
  in the binary format suitable to fast reading in on the MasPar
  It returns the number of polygons read.
***********************************************************************/
int io_polys(patch_fp,mp_fp)
FILE* patch_fp;  /* patch file */
FILE* mp_fp; /* mp file */
{ Poly *cur_poly; /* current polygon */
  Vec3 temp_vert; /*buffer to store vertices */
  int polysread = 0;  /* polygons read so far */
  int  i,j; /* misc counters */
  char line[512]; /* Input file line */
  int front[3],n; /* front face r,g,b's and num verts*/
  int back[3]; /* back face r,g,b's */
  int emitter_id;      /* Id of the emitter polygon*/
  int    txtr_id ; /* Texture stuff */
  int txtr_index;
  int          average_col[3];

  ALLOCN(cur_poly, Poly, 1, "io_polys");

  /* read in the polygons in the Walkthrough format*/
  while ((j = fscanf(patch_fp,"%d %d %d %d %d %d %d %d %d %d",
       &front[RED],&front[GREEN],&front[BLUE],&back[RED],&back[GREEN],
       &back[BLUE], &n,&emitter_id,&txtr_id,&txtr_index)) != EOF)
  { if (j != 10)
    { fprintf(stderr,"Failure for polygon#  %d\n",polysread);
      fprintf(stderr,"Polygon Header is: %d %d %d   %d %d %d   %d %d %d %d\n",
        front[RED],front[GREEN],front[BLUE],back[RED],back[GREEN],
        back[BLUE], n,emitter_id,txtr_id,txtr_index);
      die("read_poly_format","bad header read",1);
    }

    /* Initialize all entries to zeroes */
    for (j=0;j<MAXVERT;j++)
    { cur_poly->verts[j][X] = cur_poly->verts[j][Y] = cur_poly->verts[j][Z]
      = 0.0;
      cur_poly->colors[j][RED]  = cur_poly->colors[j][GREEN] =
      cur_poly->colors[j][BLUE] = cur_poly->colors[j][ALPHA] =(unsigned char)
    }

    average_col[RED] = average_col[GREEN] = average_col[BLUE] = 0;
    /* Assign vertices and colors to polygon to be written out */
    for(i = 0; i < n; i++)
    { if (fscanf(patch_fp,"%f %f %f\n",&temp_vert[X],&temp_vert[Y],
            &temp_vert[Z])!=3)
      { fprintf(stderr,"Failure for polygon#  %d\n",polysread);
        fprintf(stderr,"Polygon Header is: %d %d %d  %d %d %d  %d %d %d %d\n"
        front[RED],front[GREEN],front[BLUE],back[RED],back[GREEN],
        back[BLUE], n,emitter_id,txtr_id,txtr_index);
fflush(stderr);
        die("read_poly_format","bad vert read",1);
      }
      cur_poly->verts[i][X] = temp_vert[X];
      cur_poly->verts[i][Y] = temp_vert[Y];
      cur_poly->verts[i][Z] = temp_vert[Z];
      average_col[RED] += cur_poly->colors[i][RED] = front[RED];
      average_col[GREEN] += cur_poly->colors[i][GREEN] = front[GREEN];
      average_col[BLUE] += cur_poly->colors[i][BLUE] = front[BLUE];
    }
    cur_poly->numverts = n;

    /* find the plane equation of the polygon */
    planeEq(cur_poly,cur_poly->verts, n);

    /* find the area of the polygon */
    area(cur_poly);

    /* initialize the total radiosity values of polygons to zeroes */
    cur_poly->gather[RED] =
    cur_poly->gather[GREEN] =
    cur_poly->gather[BLUE] = 0.0;

    /* initialize the reflectance rho of polys to normalized average of
       vertex colors
    */
    n *= 255;
    cur_poly->rho[RED] = 1.0*average_col[RED] / n;
    cur_poly->rho[GREEN] = 1.0*average_col[GREEN] / n;
    cur_poly->rho[BLUE] = 1.0*average_col[BLUE] / n;


    /* If the polygon is an emitter, initialize its unshot radiosity to
    some desired values else init them to zeroes.           */

    if (emitter_id > 0)
        cur_poly->unshot[RED] =
cur_poly->unshot[GREEN] =
cur_poly->unshot[BLUE] = INIT_EMIT;
    else
        cur_poly->unshot[RED] =
cur_poly->unshot[GREEN] =
cur_poly->unshot[BLUE] = 0.0;

    /* Write out the polygon */
    bufwrite(mp_fp,(char *) cur_poly, sizeof(Poly));

    if(!(++polysread % 500))
    { fprintf(stderr, "Read polygon# %d...\n", polysread);
      fflush(stderr);
    }
  }
  return polysread;
}


/**********************************************************************
                          MAIN

**********************************************************************/
main(ac,av)
int ac;
char*   av[];
{
   FILE* patch_fp; /* Input Ascii file of dataset*/
   FILE* mp_fp; /* Output Binary file */
   char filename[128];
   int NumPolys = 0; /* Number of polygons in the dataset*/

   /* Open up the I/O files */
   if (ac == 1)
   { printf("Usage: %s fname { < fname.patch > fname.mp }\n",av[0]);
     exit(1);
   }

   if (!(patch_fp = fopen(sprintf(filename,"%s.patch",av[1]),"r")))
      die("patchtomp","can't open input",1);
    if (!(mp_fp = fopen(sprintf(filename,"%s.mp",av[1]),"w")))
       die("patchtomp","can't open output",1);

    bufwrite(mp_fp,(char *)&NumPolys, sizeof(int));

    NumPolys = io_polys(patch_fp, mp_fp);

    /* Write the number of polygons in the beginning */
    fseek(mp_fp,0L,0);
    bufwrite(mp_fp,(char *)&NumPolys, sizeof(int));

    fclose(patch_fp);
    fclose(mp_fp);
    printf("Num polys written = %d\n", NumPolys);
}
```

host.h

```
/****************************************************************************
  Copyright 1991: Amitabh Varshney UNC CS Dept. All Rights Reserved

  HOST.H

              Approach: Balanced-Load Ray-Casting


  This file is the main include file and contains the constants, macros
  typedefs used.

****************************************************************************/



/**************************************************************************
                                    CONSTANTS

**************************************************************************/

#define A 0 /* Generally used with plane equation*/
#define B 1
#define C 2
#define D 3

#define X 0 /* Generally used with vertices */
#define Y 1
#define Z 2
#define W 3

#define SKEWX 4 /* Used for the orientation of the polygon*/
#define SKEWY 5
#define SKEWZ 6

#define RED 0 /* Colors associated with the polygon*/
#define GREEN 1
#define BLUE 2
#define ALPHA 3  /* Transparency option with radiosity in
   future? .... right now for data alignment*/

#define L1_NORM_MIN 1e-4 /* Tolerance limit for computation errors */
```

```
#define HUGEF 1e+10 /* Some huge floating pt number */

#define EAST  0 /* Generally used for virtual walls */
#define WEST  1
#define NORTH 2
#define SOUTH 3
#define FLOOR 4
#define CEILING 5

#define U     0
#define V     1

#define NXPROC 64  /* Same as nxproc but facilitates array decls*/
#define NYPROC 64  /* Same as nyproc but facilitates array decls*/

#define NUMRAYS 4050 /* Number of rays per polygon */
#define NUMRADIUS 45 /* Number of radius divisions in ray firing */
#define NUMTHETA 90 /* Number of theta divisions in ray firing */
        /* NUMTHETA = 2*NUMRADIUS and
   NUMTHETA*NUMRADIUS = NUMRAYS
*/
#define MAXPOLYS 40960 /* Maximum no of polygons in all */
#define MAXPEPOLYS       3 /* Maximum no of polygons per PE */
#define MAXROWPOLYS      185 /* Maximum no of polygons per DPU row */
#define MAXRAYS          MAXPEPOLYS*NUMRAYS /* Total maximum no of rays */
#define MAXVERT 4 /* Maximum no of vertices in a polygon */

#define MAXPOLYS_PER_CELL 4096  /* Maximum no of polys in a cell */
/* X dimension of FF matrix per PE */
#define FFX MAXPOLYS_PER_CELL/NXPROC
/* Y dimension of FF matrix per PE */
#define FFY MAXPOLYS_PER_CELL/NYPROC

#define XROOM_DIM        8 /* Number of Virtual Rooms along 3-axes */
#define YROOM_DIM        8
#define ZROOM_DIM        1
#define XCELLS           16
/* No of finer patch subdivisions along axes*/
#define YCELLS           16
#define ZCELLS           2
#define XCELLLENGTH      180.0
/* Cell lengths used for patchification */
#define YCELLLENGTH      120.0
#define ZCELLLENGTH      500.0
```

```
#define INIT_EMIT
1000.0   /* Initial radiosity value for an emitter */
#define TOL             10.0
/* Terminal radiosity value for local iters*/



/************************************************************************
                                   MACROS

************************************************************************/

#define   DOT(a,b)       (a[0]*b[0] + a[1]*b[1] + a[2]*b[2]) /* Dot product */
#define   CROSS(a, b, c)          \
  { a[0] = b[1]*c[2] - c[1]*b[2];  \
    a[1] = c[0]*b[2] - b[0]*c[2]; \
    a[2] = b[0]*c[1] - c[0]*b[1];  \
    }
#define   PLURAL_NORMALIZE(a, b)          \
  { plural float magnitude; \
     magnitude= fp_sqrt(b[0]*b[0] + b[1]*b[1] + b[2]*b[2]);\
    a[0] = b[0]/magnitude;  \
    a[1] = b[1]/magnitude;  \
    a[2] = b[2]/magnitude; \
          }
#define   VEC_SUM(a)    (a[0] + a[1] + a[2])
#define   VEC_ASSIGN(a, b) \
  {a[0] = b[0]; a[1] = b[1]; a[2] = b[2];}
#define   VEC4_ASSIGN(a, b) \
  {a[0] = b[0]; a[1] = b[1]; a[2] = b[2]; a[3] = b[3];}
#define   VEC_ASSIGN_ZERO(a) \
  {a[0] = 0; a[1] = 0; a[2] = 0;}
#define   VEC_ADD(a, b, c) \
  {a[0] = b[0] + c[0]; a[1] = b[1] + c[1]; a[2] = b[2] + c[2];}

#define   MIN(x,y)       (((x)<(y))?(x):(y)) /* Minimum of two nos*/
#define   MAX(x,y)       (((x)>(y))?(x):(y)) /* Maximum of two nos*/
#define   MINX           ((X)<<1) /* Indexing in extent's array*/
#define   MAXX           (((X)<<1)+1)
#define   MINY           ((Y)<<1)
#define   MAXY           (((Y)<<1)+1)
#define   MINZ           ((Z)<<1)
#define   MAXZ           (((Z)<<1)+1)
#define   MINEX(C)       ((C)<<1)
#define   MAXEX(C)       (((C)<<1)+1)
```

```
/* Allocate N items of type TYPE at pointer location PTR on ACU or Front End*/
#define  ALLOCN(PTR,TYPE,N,RTN)     \
        if (!(PTR = (TYPE *) malloc((unsigned) (N)*sizeof(TYPE)))) {    \
   printf("malloc failed\n"); \
   exit(-1); \
 }

/* Allocate N items of type TYPE at pointer location PTR on the PEs */
#define  P_ALLOCN(PTR,TYPE,N,RTN)     \
        if (!(PTR = (plural TYPE *) p_malloc((unsigned) (N)*sizeof(TYPE)))) {
   p_printf("p_malloc failed\n"); \
   exit(-1); \
 }

#define START gettimeofday(&tm,&tz);\
        et = (tm.tv_sec)+ (0.000001* (tm.tv_usec));

#define STOP gettimeofday(&tm,&tz);\
et = (tm.tv_sec)+(0.000001*(tm.tv_usec)) - et;

/* for((v)=(f)->verts[(i)=0];(i)<(f)->n;(v)=(f)->verts[++(i)]) */

/**************************************************************************
                                TYPEDEFS

**************************************************************************/

typedef float Vec3[3];
typedef float Vec4[4];

typedef unsigned char byte;
/* Used to define colors and form-factors*/

/* -POLYGON-  */
typedef struct polygon
{ int numverts;   /* Number of vertices */
  float verts[MAXVERT][3];/* Points of the polygon (quad/triangle) */
  Vec4 eq;   /* Eq of the polygon (quad/triangle) */
  byte          colors[MAXVERT][4];/*Colors at the vertices */
  float unshot[3];     /* Unshot rad value for front & back face */
  float gather[3];   /* Accumulated energy for front & back face */
  float rho[3];      /* Reflectance for front and back face */
  float area;   /* Area of the polygon */
  int id;   /* Polygon id  */
} Poly;

/* -POLYGON GEOMETRY-*/
typedef struct poly_geom
{ int numverts;   /* Number of vertices */
  float verts[MAXVERT][3];/* Points of the polygon (quad/triangle) */
  Vec4 eq;   /* Eq of the polygon (quad/triangle) */
} PolyGeom;


/* -RAY-  */
typedef struct ray
{ float origin[3];    /* Origin of the ray */
  float direction[3];      /* Direction of the ray */
  int id;    /* Id of the polygon it hits */
  float energy[3];    /* Energy with this ray */
  float distance;    /* Parameter 't' in its parametric form */
} Ray;


/*       -Miscellaneous data used per poly during ff calc- */
typedef struct poly_tag
{ float poly_center[3];    /* Center of the polygon */
  int x, y;    /* Axes along which the polygon lies */
  float uv[2][3];    /* Some variables to avoid repeated calc*/
  float beta_denom[2];
  float  poly_mat[3][4];    /* Used in orienting rays to be fired */
} PolyTag;

/*  -ENERGY PACKETS-  */
typedef struct energy
{ int id; /* Id of the patch which shot this packet */
  float unshot[3]; /* Unshot radiosity for the shooting patch */
  float area; /* Area of the shooting patch */
} Packet;


/*================================Extern Defs=============================*/

extern char*             malloc();
```

host.c

```
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include "host.h"
#include "error.h"

extern void polys_to_pe();
Poly *poly_list;      /* list of polygons */


/* Low level C file i/o routine to read in large amounts of data */
bufread(fd,ptr,size,num_items)
int  fd;
char*   ptr;
int size;
int num_items;
{ if (read(fd, ptr, size*num_items) < 0)
      die("bufread","bad read",1);
}



/* This reads in the polygons from the special binary format file */
int read_polys(mp_fp)
int mp_fp;  /* binary file */

{ Poly *cur_poly; /* current polygon in the poly_list */
  int  i;
  int numPolys; /* number of polygons read */

  /* determine the number of polygons to be read */
  bufread(mp_fp,(char *) &numPolys, sizeof(int), 1);

  /* allocate space accordingly */
  ALLOCN(poly_list, Poly, MAXPOLYS, "read_polys");

  /* read in the polygons */
  bufread(mp_fp, (char *) poly_list, sizeof(Poly), numPolys);

  return(numPolys);
}
```

```
/* This function prints out the polygon cur_poly - useful in debugging */
print_poly(cur_poly)
Poly *cur_poly;
{ int  j;

  printf("%d \n",cur_poly->numverts);
  for (j=0;j<MAXVERT;j++)
  { printf("%f %f %f \n",cur_poly->verts[j][X],cur_poly->verts[j][Y],
 cur_poly->verts[j][Z]);
  }
}


int option_handler(ac, av)
int  ac;
char *av[];
{ register int         i, ok = 1;
  register char        *c;

  for (i = 1; i < ac && av[i][0] == '-'; i++)
      for (c = &(av[i][1]); *c; c++) switch (*c) {
          case '?':
               printf("No options are available\n");
               break;
          default:
              fprintf(stderr, "%s: unknown option -%c\n", av[0], *c);
              ok = 0;
              break;
   }
   return (ok ? i : 0);
}

/****************************** Main ***********************************/
main(ac, av)
int ac;
char *av[];
{ int  options;
  int mp_fp; /* Input binary file */
  char filename[128]; /* Input file name */
  int total_polys = 0;

  struct timeval  tm; /* Timing stuff */
  struct timezone  tz;
  double et;
  if (!(options = option_handler(ac,av))) die("io","bad options",1);

  if (!(mp_fp = open(sprintf(filename,"%s.mp",av[options]),O_RDONLY)))
      die("io","can't open input mp file",1);

  /* Read in the polygons*/
  total_polys = read_polys(mp_fp);
  close (mp_fp);

  /* Transfer the polygons to the DPU */
  fprintf(stdout,"Xferring %d polys to PEs\n",total_polys);
  fflush(stdout);

  START

  callRequest(polys_to_pe, 8, poly_list, total_polys);

  STOP

  /* Print out the timing stats */
  fprintf(stdout,"Xfer to PEs over. Time in DPU = %10.2f secs\n", et);
  fflush(stdout);
}
```

error.h

```
/* Include file for error handling */
    extern FILE* ErrFile;
    extern void die(); extern void warning();
```

error.c

```
/* File for handling errors */

#include <stdio.h>
#include "error.h"

FILE* ErrFile = stderr;

/* Print out an error message and exit with an error code */
void die(rtn,msg,code)
char* rtn;
char* msg;
int code;
{ fprintf(ErrFile,"%s: %s\n",rtn,msg); exit(code);  }

/* Print out a warning message */
void warning(rtn,msg,code)
char* rtn;
char* msg;
int code;
{ fprintf(ErrFile,"%s: %s code = %d\n",rtn,msg,code); }
```

dist.m for Balanced-Load Ray-Casting Approach

```
/***************************************************************************

  Copyright 1991: Amitabh Varshney, UNC CS Dept. All Rights Reserved

                                 DIST.M

       Approach: Balanced-Load Ray-Casting

   This part of the code is responsible for distributing the energies.

***************************************************************************/
#include <mpl.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include "host.h"

extern struct timeval          tm; /* timing variables */
extern struct timezone         tz;

extern int maxPEpoly; /* max no of polys per PE */
extern plural Poly*  old_poly_list;  /* Polys to be used */
extern plural int oldPoly; /*No of polys in old_poly_list*/
/* Form factor array */
extern plural float form_factors;

float MinRad = 150.0;


/*************************************************************************
                            dist_energy

  Carry out the distribution of energies from patch to patch within
  the current cell. This assumes that the form-factors have been
  calculated and the polygons are arranged in a load-balanced fashion

 ************************************************************************/
dist_energy()
{ float max_unshot; /* Maximum unshot energy */
  float total_unshot = HUGEF;   /* Total unshot energy */
  float shooting_rad[3]; /* Radiosity  of the shooting patch */
  int i, j, iteration = 0;
```

```
float shooter_area; /* Area of the shooting patch */
int   shooter_id; /* Polygon id of shooting patch */
int total_rays = NUMRADIUS*NUMTHETA;
plural float fraction;
plural float poly_unshot;
plural float temp;
plural Poly* distPoly; /* Polygons used in distribution*/

double et;
double cumulative_time = 0;

distPoly = old_poly_list;
if (distPoly->area < 1.)  distPoly->area  = 1.0; /* clamp areas from below*/

/* Initialize the radiosities */
VEC_ASSIGN(distPoly->gather, distPoly->unshot);

START
do
{ poly_unshot = VEC_SUM(distPoly->unshot);
  max_unshot = reduceMaxf(poly_unshot);
  printf(" %d ", iteration);
  fflush(stdout);
  if (total_unshot >= 2*max_unshot)  /* Print out in a logarithmic fashion */
  { total_unshot = max_unshot;
    STOP
    cumulative_time += et;
    printf("\n Iteration %d total unshot rad = %f max unshot rad = %f time %5
    fflush(stdout);
    START
  }

  /* In determining the shooting patch ensure that only one PE is active
     and then find the id of the polygon on that PE.
  */
  if (poly_unshot == max_unshot)
  { i = selectOne();
    shooter_id = proc[i].distPoly->id;
    shooter_area = proc[i].distPoly->area;
    VEC_ASSIGN(shooting_rad, proc[i].distPoly->unshot);
    VEC_ASSIGN_ZERO(proc[i].distPoly->unshot);
  }

  /* Calculate form factors for this shooting patch */
  calculate_form_factors(i, shooter_id);
```

```
    /* Calculate fraction of the energy received by each patch and then
       update the radiosity values of that patch.
    */
    fraction = (form_factors/total_rays)*(shooter_area/distPoly->area);
    for (j = 0; j < 3; j++)
    { temp = distPoly->rho[j]*fraction*shooting_rad[j];
      distPoly->unshot[j] += temp;
      distPoly->gather[j] += temp;
    }
    iteration++;
  } while (VEC_SUM(shooting_rad) > MinRad);

  STOP
  cumulative_time += et;
  poly_unshot = VEC_SUM(distPoly->unshot);
  max_unshot = reduceMaxf(poly_unshot);
  total_unshot = reduceAddf(poly_unshot);
  printf("\n Iteration %d total unshot rad = %f max unshot rad = %f time %5.2f
  fflush(stdout);
}
```

ff.m for Balanced-Load Ray-Casting Approach

```
/*************************************************************************
 Copyright 1991: Amitabh Varshney, UNC CS Dept. All Rights Reserved

                                FF.M

              Approach: Balanced-Load Ray-Casting

  This version of ff.m calculations spreads out polygons in a balanced
  manner. One ray is broadcast to all the PEs. Each PE computes the
  intersection with its polygon and returns the distance of the
  intersected point from the ray's origin. A global reducemin is done
  to obtain the PE on which the polygon that intersected the ray lies
  and the ff array is then updated by a proc command. The ray-polygon
  intersection code is quite tight for this version.


*************************************************************************/

#include <mpl.h>
#include <math.h>
#include "host.h"

#include <sys/time.h>

extern struct timeval  tm;
extern struct timezone tz;
extern double          et;

extern plural int oldPoly; /* Polygon buffer */
extern plural Poly *old_poly_list; /* List of patches to be used*/
extern int maxPEpoly; /* Max no of polygons perPE */
plural float form_factors; /* Form-factor array */
float canonical_dirs[NUMRADIUS][NUMTHETA][3];
plural PolyTag tag; /* To store precomputed polygon
   data used in form-factor
   calculations
 */

/**********************************************************************
                      initialize_form_factors
  This function intializes the data structures for use in ff
  calculation.

***********************************************************************/
initialize_form_factors()
{
  plural Poly*  tempPoly; /* Polygon pointer */
  float      xyz[3][3];
  register  int i;  /* Misc counters */

START
  /* Initialize the three axes vectors */
  for(i = 0; i < 3; i++)
  { VEC_ASSIGN_ZERO(xyz[i]);
    xyz[i][i] = 1.0;
  }

  /* Assign the id's to polygons on PEs that have >= 1 polygon */
  /* Assumes MAXPOLYS_PER_CELL <= 4k */
  old_poly_list->id = (oldPoly)? iproc: -1;

  tempPoly = old_poly_list;

  /* Compute the tag data for PEs having valid polys*/
  if (tempPoly->id >= 0)
  { register plural int numverts, m;
    plural float         normal[3];
    register plural int l0, l1, l2;

    numverts = tempPoly->numverts;

    if (numverts != 4 & numverts != 3)
    { p_printf("ff bad numverts = %d\n",numverts);
      exit(-1);
    }

    /* Find the center of the polygon */
    VEC_ASSIGN_ZERO(tag.poly_center);
    for (m = 0; m < numverts; m++)
      VEC_ADD(tag.poly_center, tag.poly_center, tempPoly->verts[m]);

    tag.poly_center[X] /= numverts;
    tag.poly_center[Y] /= numverts;
    tag.poly_center[Z] /= numverts;
```

```
/* Find the orientation of the polygon */
VEC_ASSIGN(normal, tempPoly->eq);
normal[X] = (normal[X] < 0)? -normal[X] : normal[X];
normal[Y] = (normal[Y] < 0)? -normal[Y] : normal[Y];
normal[Z] = (normal[Z] < 0)? -normal[Z] : normal[Z];

/* Find the axis that is most nearly perpendicular to the normal */
if ((normal[X] <= normal[Y]) & (normal[X] <= normal[Z]))
  l0 = X;

if ((normal[Y] <= normal[X]) & (normal[Y] <= normal[Z]))
  l0 = Y;

if ((normal[Z] <= normal[X]) & (normal[Z] <= normal[Y]))
  l0 = Z;

/* Find the 2 axes that are most nearly perpendicular to the normal */
if ((normal[X] >= normal[Y]) & (normal[X] >= normal[Z]))
{ l1 = Y;
  l2 = Z;
}
if ((normal[Y] >= normal[X]) & (normal[Y] >= normal[Z]))
{ l1 = X;
  l2 = Z;
}
if ((normal[Z] >= normal[X]) & (normal[Z] >= normal[Y]))
{ l1 = X;
  l2 = Y;
}

/* Compute the polygon matrix */
CROSS(tag.poly_mat[0], xyz[l0], tempPoly->eq);
PLURAL_NORMALIZE(tag.poly_mat[0], tag.poly_mat[0]);

CROSS(tag.poly_mat[1], tempPoly->eq, tag.poly_mat[0]);
VEC_ASSIGN(tag.poly_mat[2], tempPoly->eq);

/* Compute the polygon dependent constants as a precomputation
   step to speed up ray-polygon intersection code.
*/
tag.uv[U][0] = tempPoly->verts[1][l1] - tempPoly->verts[0][l1];
tag.uv[V][0] = tempPoly->verts[1][l2] - tempPoly->verts[0][l2];
tag.uv[U][1] = tempPoly->verts[2][l1] - tempPoly->verts[0][l1];
tag.uv[V][1] = tempPoly->verts[2][l2] - tempPoly->verts[0][l2];
tag.uv[U][2] = tempPoly->verts[3][l1] - tempPoly->verts[0][l1];
```

```
    tag.uv[V][2] = tempPoly->verts[3][12] - tempPoly->verts[0][12];
    tag.beta_denom[0] = tag.uv[V][1]*tag.uv[U][0] - tag.uv[U][1]*tag.uv[V][0];
    tag.beta_denom[1] = tag.uv[V][2]*tag.uv[U][1] - tag.uv[U][2]*tag.uv[V][1];
    tag.x     = 11;
    tag.y     = 12;

  }

  /* Compute and store the directions of ray firings */
  { register float numr_2 = 2*NUMRADIUS;
    register float m_pi_2_by_numr = 2*M_PI/NUMTHETA;
    register int   r, t;
    for (r = 0; r < NUMRADIUS; r++)
    { register float radius = sqrt((2*r + 1.)/(numr_2));
      for (t = 0; t < NUMTHETA; t++)
      { register float theta = (t + 0.5)*m_pi_2_by_numr;
        canonical_dirs[r][t][X] = radius*cos(theta);
        canonical_dirs[r][t][Y] = radius*sin(theta);
        canonical_dirs[r][t][Z] = sqrt(1.0 - radius*radius);
      }
    }
  }

STOP
  printf("Time for initialization %5.2f\n",et);

}

/*********************************************************************
                     calculate_form_factors

  This function calculates the form-factor matrices for the
  polygon with id shooter_id, stored at proc i and stores these
  at each processor. This assumes that the cell is stored over the
  whole 4k grid (one poly per PE)
 *********************************************************************/
calculate_form_factors(i, shooter_id)
int i, shooter_id;
{
  plural Poly*  tempPoly; /* Polygon pointer */
  Ray defaultRay; /* Ray buffer */
  plural Ray testRay;
  register int j, k; /* Counters */
  register int intersected_poly;
  register float        distance;
register int r, t;
register float radius, theta;

tempPoly = old_poly_list;

/* initialize form_factors to zero */
form_factors = 0.0;

/* set ray direction */
VEC_ASSIGN(defaultRay.direction, proc[i].tempPoly->eq);

/* set ray origin to patch center */
VEC_ASSIGN(defaultRay.origin, proc[i].tag.poly_center);

defaultRay.distance = HUGEF;
/* ray has initial id of the shooting poly */
defaultRay.id = shooter_id;

/* Broadcast the ray to all PEs */
/* Compute the direction to shoot the ray in, to ensure equal energy rays
   using John Airey and Ming Young's method of hemisphere subdivision.
   [Airey89].
*/

for (r = 0; r < NUMRADIUS; r++)
{ for (t = 0; t < NUMTHETA; t++)
  {
    testRay.direction[X] = DOT(tag.poly_mat[0], canonical_dirs[r][t]);
    testRay.direction[Y] = DOT(tag.poly_mat[1], canonical_dirs[r][t]);
    testRay.direction[Z] = DOT(tag.poly_mat[2], canonical_dirs[r][t]);

    VEC_ASSIGN(testRay.origin, defaultRay.origin);

    testRay.distance = defaultRay.distance;
    testRay.id = defaultRay.id;

    /* Each PE computes the intersection of this ray with poly(s) it has and
       computes the distance and id of the poly intersected on the PE.
    */
    all if (testRay.id >= 0) intersect_ray(&testRay);
    /* Compute the shortest distance for intersection over all PEs */
    distance = reduceMinf(testRay.distance);

    /* Identify the polygon intersected */
    intersected_poly = -1;
```

```
      if ((distance < HUGEF) & (testRay.distance == distance))
      { k = selectOne();
        intersected_poly = proc[k].testRay.id;
      }

      /* Update the form-factor array */
      proc[intersected_poly].form_factors++;
    }
  }
  /*
  printf("polys hit %d not hit %d\n",hit, unhit);
  */
}


/*************************************************************************
                            intersect_ray

  This function would intersect the given ray with all the
  polygons present on the PE to which it belongs and returns the id
  of the polygon it finds has the closest intersection point in the
  id field of the ray. This code for this routine is based on the
  ray-polygon intersection routine in [Glassner90]. It has been
  adapted for SIMD execution and all ray independent terms are
  precomputed and stored in the poly_tag data structure.
*************************************************************************/
intersect_ray(testRay)
plural Ray *testRay; /* Input ray */
{
  plural Poly*          tempPoly;
  plural float p[3];             /* Intersection point */
  plural int j;
  register plural float  t;
  register plural float ndotd; /*Dot prod of normal and ray */
  register plural byte inter;
  register plural float alpha, beta;
  register plural float uv[2];
  register plural int n, m, m_1;
  register plural int numverts_1;

  for (j = 0; j < oldPoly; j++)
  { tempPoly = old_poly_list;

    numverts_1 = tempPoly->numverts - 1;
    /* Compute the dot product of the ray direction with the poly normal */

    ndotd = DOT(tempPoly->eq, testRay->direction);

    if (ndotd == 0.0) continue;

    t = -(tempPoly->eq[D] + DOT(tempPoly->eq, testRay->origin))/ndotd;

    /* Avoid intersecting with originating polygon*/
    /* Polygon is beyond closest intersection found so far*/
    if ((t <= 0.002) | (t >= testRay->distance)) continue;

    /* Calculate the point of intersection */
    p[X] = testRay->origin[X] + testRay->direction[X]*t;
    p[Y] = testRay->origin[Y] + testRay->direction[Y]*t;
    p[Z] = testRay->origin[Z] + testRay->direction[Z]*t;

    /* Verify if the point of intersection is within the polygon */
    uv[U] = p[tag.x] - tempPoly->verts[0][tag.x];
    uv[V] = p[tag.y] - tempPoly->verts[0][tag.y];

    inter = 0;
    m = 1;

    do
    { m_1 = m - 1;

      if (n = (tag.uv[U][m_1] == 0))
        beta = uv[U]/tag.uv[U][m];
      else
        beta =(uv[V]*tag.uv[U][m_1] - uv[U]*tag.uv[V][m_1])/tag.beta_denom[m_1]

      if (beta >= 0.0 & beta <= 1.0)
      { alpha = (uv[n] - beta*tag.uv[n][m])/tag.uv[n][m_1];
        inter = (alpha >= 0.0 & alpha+beta <= 1.0);
      }

    } while (!inter & ++m < numverts_1);

    if (inter)
    { testRay->distance = t;
      testRay->id = tempPoly->id;/* intersection was with a model polygon */
    }
  }
}
```

io.m for Balanced-Load Env-Proj and Ray-Casting Approaches

```
/*********************************************************************
  Copyright 1991: Amitabh Varshney and Howard Good, UNC CS Dept.
   All Rights Reserved

        IO.M
              Approach: Balanced-Load Ray-Casting  and
Balanced-Load Env-Projection

    This program runs on the back-end of MasPar and carries out most
  of the tasks including swapping in of the polygons from the from-end
*********************************************************************/

#include <mpl.h>
#include <stdio.h>
#include <mpl/ppeio.h>
#include <maspar/vmeaccess.h>
#include <math.h>
#include <sys/time.h>
#include "host.h"

/********************************************************************
                               GLOBAL DECLS

********************************************************************/

extern  die();

struct timeval  tm; /* Timing stuff */
struct timezone tz;
double et;

float   MinX, MinY, MinZ; /* Extents of the input dataset*/
float   MaxX, MaxY, MaxZ;
plural Poly *old_poly_list; /* Lists of polygons */
plural Poly *new_poly_list;
plural int oldPoly; /* Number of polygons in old_poly_list*/
plural int newPolyCount; /* Number of polygons in new_poly_list*/
int maxPEpoly; /* Maximum no of polygons/PE */

float  xroomBound[XROOM_DIM];  /* Upper bounds of each virtual room */
float  yroomBound[YROOM_DIM];
float  zroomBound[ZROOM_DIM];
```

```
/* Low level C file i/o routine to read in large amounts of data */
bufread(fd,ptr,size,num_items)
int  fd;
char*   ptr;
int size;
int num_items;
{ if (read(fd, ptr, size*num_items) < 0)
      die("bufread","bad read",1);
}


/***********************************************************************
                              polys_to_pe

  This reads in the polygons from the .mp file and distributes them
  on PEs.

***********************************************************************/

int polys_to_pe(mp_fp)
int mp_fp;
{ int numpoly,maxpoly,minpoly;
  int n;
  int initPEpoly;          /* initial no of polygons per PE */
  int total_polys; /* number of polygons to be read */
  plural int check;

  plural Poly*  tempPoly;

  int cmd;

  cmd = MP_S8;
  p_fcntl(mp_fp,F_SETPIO,cmd);

  /* determine the number of polygons to be read */
  bufread(mp_fp,(char *) &total_polys, sizeof(int), 1);

  initPEpoly = (int)(total_polys/nproc);
  maxPEpoly = initPEpoly*6;

  fprintf(stdout,"polys %d nproc %d polys/pe %d poly_size %d\n", total_polys,
  nproc,initPEpoly, sizeof(Poly));
  fflush(stdout);
  /* Allocate PE memory */
  P_ALLOCN(old_poly_list, Poly, initPEpoly + 1, "polys_to_pe");
  /* P_ALLOCN(new_poly_list, Poly, maxPEpoly, "polys_to_pe"); */

  /* Read in polygons to PEs from the file */
  check = p_read(mp_fp, (plural char *) old_poly_list, sizeof(Poly)*initPEpol
  tempPoly = old_poly_list + initPEpoly;
  check += p_read(mp_fp, (plural char *) tempPoly, sizeof(Poly));
  if (check == -1) die ("polys_to_pe","error reading file",1);

  /* Determine oldPoly for each PE */
  oldPoly = check/sizeof(Poly);

  numpoly = reduceAdd32(oldPoly);
  maxpoly = reduceMax32(oldPoly);
  minpoly = reduceMin32(oldPoly);
  printf("old polys = %d; max = %d, min = %d\n",numpoly,maxpoly,minpoly);

  return(total_polys);

/*
   Start the radiosity iterations
   Carry out the patch - patch energy exchange
  dist_pp();

   Carry out the exchange of energy laden virtual walls
  xchg_walls();
*/
}

/***********************************************************************
                              pe_to_polys

  This writes out the polygons from the PEs to a .0.patch file.

***********************************************************************/
/* for now it writes only the polys that have received any energy */
pe_to_polys(fp)
FILE* fp;
{ int i, j, numverts;
  plural Poly* tempPoly;
  float gather[3];
  float gather_sum;

  tempPoly = old_poly_list;
  for (i = 0; i < nproc ; i++)
  { if (proc[i].tempPoly->id >= 0)
    { numverts =  proc[i].tempPoly->numverts;
      gather_sum = 0;
      for (j = 0; j < 3; j++)
gather_sum += gather[j] = proc[i].tempPoly->gather[j];
      if (gather_sum > 0)
      { fprintf(fp,"%g %g %g %d %d %d %d 0 0 0 0 0\n",gather[RED],gather[GREEN]
        gather[BLUE], (int)proc[i].tempPoly->colors[0][RED],
(int)proc[i].tempPoly->colors[0][GREEN],
(int)proc[i].tempPoly->colors[0][BLUE], numverts);
        for(j = 0; j < numverts; j++)
          fprintf(fp,"%g %g %g\n",proc[i].tempPoly->verts[j][X],
          proc[i].tempPoly->verts[j][Y], proc[i].tempPoly->verts[j][Z]);
      }
    }
  }

}


/*************************************************************************
                          determine_extents

   This routine determines the extents of the polygon dataset.
   It takes into account the coarse cell lengths to determine the
   extents which will be represented on the DPU array once the
   balancing is done.

*************************************************************************/
determine_extents()
{ plural int i, j; /* Miscellaneous counters */
  plural float x, y, z; /* Vertex values */
  plural Poly* plural tempPoly; /* Current polygon */
  plural float  minx, miny, minz; /* Local extents on each PE */
  plural float  maxx, maxy, maxz;


  /* Determine the local extents */
  minx = miny = minz = HUGEF;
  maxx = maxy = maxz = -HUGEF;
  for (i = 0; i<oldPoly; i++)
  { tempPoly = old_poly_list + i;
```

```
    if (tempPoly->numverts < 3 || tempPoly->numverts > MAXVERT) {
      p_printf("ERROR: proc %d numverts = %d\n",iproc,tempPoly->numverts);
      exit(1);
    }

    for (j = 0; j<tempPoly->numverts; j++)
    { x = tempPoly->verts[j][X];
      y = tempPoly->verts[j][Y];
      z = tempPoly->verts[j][Z];
      minx = MIN(minx,x);
      miny = MIN(miny,y);
      minz = MIN(minz,z);
      maxx = MAX(maxx,x);
      maxy = MAX(maxy,y);
      maxz = MAX(maxz,z);
    }
  }


  /* Determine the global extents */

  MinX = reduceMinf(minx);
  MinY = reduceMinf(miny);
  MinZ = reduceMinf(minz);
  MaxX = reduceMaxf(maxx);
  MaxY = reduceMaxf(maxy);
  MaxZ = reduceMaxf(maxz);

  printf("Minx %6.2f MaxX %6.2f MinY %6.2f MaxY %6.2f MinZ %6.2f MaxZ %6.2f\n"
  MinX,MaxX,MinY,MaxY,MinZ,MaxZ);

  /* Take into account the global patchification grid aligned along the axes*/

  MinX = f_floor(MinX/XCELLLENGTH) * XCELLLENGTH;
  MinY = f_floor(MinY/YCELLLENGTH) * YCELLLENGTH;
  MinZ = f_floor(MinZ/ZCELLLENGTH) * ZCELLLENGTH;
  MaxX = f_ceil(MaxX/XCELLLENGTH) * XCELLLENGTH;
  MaxY = f_ceil(MaxY/YCELLLENGTH) * YCELLLENGTH;
  MaxZ = f_ceil(MaxZ/ZCELLLENGTH) * ZCELLLENGTH;

  printf("Minx %6.2f MaxX %6.2f MinY %6.2f MaxY %6.2f MinZ %6.2f MaxZ %6.2f\n"
  MinX,MaxX,MinY,MaxY,MinZ,MaxZ);
}
```

```
/************************************************************************
                                area

   This routine determines the area of a plural polygon defined by
   the vertex array 'vert' and having 'numverts' number of vertices
   For this the routine considers the polygon to be composed of
   triangles and then computes the area of each triangle by taking
   half of the magnitude of the cross product of two of its sides.

************************************************************************/
plural float area(verts, numverts)
plural float verts[MAXVERT][3]; /* Vertex array */
plural int numverts; /* Number of vertices */
{ plural int    i,j,k;
  plural float  area = 0.0;
  plural float  v1[3], v2[3], v3[3];

  for(i=0; i < numverts - 2; i++)
  { j = i+1;
    k = i+2;

    v1[X] = verts[j][X] - verts[i][X];
    v1[Y] = verts[j][Y] - verts[i][Y];
    v1[Z] = verts[j][Z] - verts[i][Z];

    v2[X] = verts[k][X] - verts[i][X];
    v2[Y] = verts[k][Y] - verts[i][Y];
    v2[Z] = verts[k][Z] - verts[i][Z];

    v3[X] =  v1[Y]*v2[Z] - v1[Z]*v2[Y];
    v3[Y] = -v1[X]*v2[Z] + v1[Z]*v2[X];
    v3[Z] =  v1[X]*v2[Y] - v1[Y]*v2[X];

    area += 0.5*fp_sqrt(v3[X]*v3[X] + v3[Y]*v3[Y] + v3[Z]*v3[Z]);
  }
  return area;
}


int option_handler(ac, av)
int  ac;
char *av[];
{ register int        i, ok = 1;
  register char       *c;
  for (i = 1; i < ac & av[i][0] == '-'; i++)
      for (c = &(av[i][1]); *c; c++) switch (*c) {
          case '?':
               printf(" <  fname1.mp  > fname2.0.patch \n");
               break;
          default:
              fprintf(stderr, "%s: unknown option -%c\n", av[0], *c);
              ok = 0;
              break;
   }
   return (ok ? i : 0);
}

/************************************************************************
  Main

************************************************************************/

main(ac, av)
int ac;
char *av[];
{ int  options;
  int mp_fp; /* Input binary file */
  char filename[128]; /* Input file name */
  int total_polys = 0;
  FILE* fp;

  if (!(options = option_handler(ac,av))) die("io","bad options",1);

  /* open the input file in read_only mode */
  if ((mp_fp = open(av[1],0)) <= -1)
      die("io","can't open input mp file",1);

  /* open the output file in write mode */
  if ((fp = fopen(av[2],"w")) == NULL)
      die("io","can't open output .0.patch file",1);

  /* Read in the polygons*/
  START
  total_polys = polys_to_pe(mp_fp);
  STOP
  close (mp_fp);

  /* Print out the timing stats*/
  fprintf(stdout,"Xferred %d polys to PEs in %5.2f secs\n",total_polys,et);
  fflush(stdout);

  /* Determine the extents of the dataset - used in balancing
  determine_extents();
  */

  /* Perform the one-time initialization for the form-factor determinations */
  initialize_form_factors();

  /* Distribute the energy in the environment */
  START
  dist_energy();
  STOP
  fprintf(stdout,"Time for energy distribution %5.2f secs\n",et);
  fflush(stdout);

  /* write out the patches */
  START
  pe_to_polys(fp);
  STOP
  fclose(fp);

  /* Print out the timing stats*/
  fprintf(stdout,"Wrote out patches in %5.2f secs\n",et);
  fflush(stdout);
}
```

host.h for Balanced-Load Env-Proj Approach

```
/*************************************************************************
 This file is the main include file and contains the constants, macros
 typedefs used.

**************************************************************************/



/***********************************************************************
                                 CONSTANTS

***********************************************************************/

#define A 0 /* Generally used with plane equation*/
#define B 1
#define C 2
#define D 3

#define X 0 /* Generally used with vertices */
#define Y 1
#define Z 2
#define W 3

#define SKEWX 4 /* Used for the orientation of the polygon*/
#define SKEWY 5
#define SKEWZ 6

#define RED 0 /* Colors associated with the polygon*/
#define GREEN 1
#define BLUE 2
#define ALPHA 3  /* Transparency option with radiosity in
   future? .... right now for data alignment*/

#define L1_NORM_MIN 1e-4 /* Tolerance limit for computation errors */

#define HUGEF 1e+10 /* Some huge floating pt number */

#define EAST  0 /* Generally used for virtual walls */
#define WEST  1
#define NORTH 2
#define SOUTH 3
#define FLOOR 4
```

```
#define CEILING 5

#define U      0
#define V      1

#define PIXELS 1  /* Hemi-plane pixels per PE */
 /* These are ordered as: 0 1
  2 3
         */
#define PIX_ROW 1  /* No of pixels per PE per row= sqrt(PIXELS) */

#define NPROC 4096  /* Same as nproc but facilitates array decls*/
#define NXPROC 64  /* Same as nxproc but facilitates array decls*/
#define NYPROC 64  /* Same as nyproc but facilitates array decls*/
#define NXPROC_1 63
#define NYPROC_1 63

#define MAX_POLY_PIXELS 1100     /* Max no of item-buffer pixels that a single
   polygon can cover.
*/
#define MAX_OVERLAP 64
/* Maximum no of polygons that get projected on same
   leftmost bounding edge in x on shooter polygon */
#define MAXPOLYS 40960 /* Maximum no of polygons in all */
#define MAXVERT  4 /* Maximum no of vertices in a polygon */

#define MAXPOLYS_PER_CELL 4096  /* Maximum no of polys in a cell */

#define INIT_EMIT
1000.0   /* Initial radiosity value for an emitter */
#define TOL             10.0
/* Terminal radiosity value for local iters*/




/*********************************************************************
                                 MACROS

*********************************************************************/

#define   DOT(a,b)      (a[0]*b[0] + a[1]*b[1] + a[2]*b[2]) /* Dot product */
#define   CROSS(a, b, c)          \
  { a[0] = b[1]*c[2] - c[1]*b[2];  \
    a[1] = c[0]*b[2] - b[0]*c[2]; \
    a[2] = b[0]*c[1] - c[0]*b[1];  \
    }
#define   PLURAL_NORMALIZE(a, b)          \
  { plural float magnitude; \
     magnitude= fp_sqrt(b[0]*b[0] + b[1]*b[1] + b[2]*b[2]);\
    a[0] = b[0]/magnitude;  \
    a[1] = b[1]/magnitude;  \
    a[2] = b[2]/magnitude; \
           }
#define   VEC_SUM(a)    (a[0] + a[1] + a[2])
#define   VEC_ASSIGN(a, b) \
  {a[0] = b[0]; a[1] = b[1]; a[2] = b[2];}
#define   VEC4_ASSIGN(a, b) \
  {a[0] = b[0]; a[1] = b[1]; a[2] = b[2]; a[3] = b[3];}
#define   VEC_ASSIGN_ZERO(a) \
  {a[0] = 0; a[1] = 0; a[2] = 0;}
#define   VEC_ADD(a, b, c) \
  {a[0] = b[0] + c[0]; a[1] = b[1] + c[1]; a[2] = b[2] + c[2];}
#define   VEC_MUL(a, b, c) \
  {a[0] = b[0] * c[0]; a[1] = b[1] * c[1]; a[2] = b[2] * c[2];}

#define   FROM_MP_TO_IEEE_INT(a) \
  (((a << 24) & 0xff000000) | ((a << 8) & 0x00ff0000) \
  |((a >> 8)  & 0x0000ff00) | ((a >> 24) & 0x000000ff))

#define   FROM_MP_TO_IEEE_FLOAT(a) \
  (((a << 8 & 0x0000ff00) | (a >> 8 & 0x000000ff) \
   |(a << 8 & 0xff000000) | (a >> 8 & 0x00ff0000)) - 1)

#define   MIN(x,y)      (((x)<(y))?(x):(y)) /* Minimum of two nos*/
#define   MAX(x,y)      (((x)>(y))?(x):(y)) /* Maximum of two nos*/
#define   MINX          ((X)<<1) /* Indexing in extent's array*/
#define   MAXX          (((X)<<1)+1)
#define   MINY          ((Y)<<1)
#define   MAXY          (((Y)<<1)+1)
#define   MINZ          ((Z)<<1)
#define   MAXZ          (((Z)<<1)+1)
#define   MINEX(C)      ((C)<<1)
#define   MAXEX(C)      (((C)<<1)+1)

/* Allocate N items of type TYPE at pointer location PTR on ACU or Front End*/
#define  ALLOCN(PTR,TYPE,N,RTN)     \
         if (!(PTR = (TYPE *) malloc((unsigned) (N)*sizeof(TYPE)))) {   \
   printf("malloc failed\n"); \
   exit(-1); \
 }
```

```
/* Allocate N items of type TYPE at pointer location PTR on the PEs */
#define  P_ALLOCN(PTR,TYPE,N,RTN)     \
         if (!(PTR = (plural TYPE *) p_malloc((unsigned) (N)*sizeof(TYPE)))) {
   p_printf("p_malloc failed\n"); \
   exit(-1); \
 }

#define START gettimeofday(&tm,&tz);\
        et = (tm.tv_sec)+ (0.000001* (tm.tv_usec));

#define STOP gettimeofday(&tm,&tz);\
et = (tm.tv_sec)+(0.000001*(tm.tv_usec)) - et;

/* for((v)=(f)->verts[(i)=0];(i)<(f)->n;(v)=(f)->verts[++(i)]) */

/*********************************************************************
                               TYPEDEFS

*********************************************************************/

typedef float Vec3[3];
typedef float Vec4[4];

#ifndef byte_defined
typedef unsigned char byte;
/* Used to define colors and form-factors*/
#define byte_defined 1
#endif

/* -POLYGON-  */
typedef struct polygon
{ int numverts;   /* Number of vertices */
  float verts[MAXVERT][3];/* Points of the polygon (quad/triangle) */
  Vec4 eq;   /* Eq of the polygon (quad/triangle) */
  byte          colors[MAXVERT][4];/*Colors at the vertices */
  float unshot[3];    /* Unshot rad value for front & back face */
  float gather[3];   /* Accumulated energy for front & back face */
  float rho[3];      /* Reflectance for front and back face */
  float area;   /* Area of the polygon */
  int id;   /* Polygon id  */
} Poly;

/*       -Miscellaneous data used per poly during ff calc- */
typedef struct poly_tag
```

```
{ float poly_center[3];    /* Center of the polygon */
  int x, y;    /* Axes along which the polygon lies */
  float uv[2][3];    /* Some variables to avoid repeated calc*/
  float beta_denom[2];
  float  poly_mat[3][4];    /* Used in orienting rays to be fired */
} PolyTag;

/*    - Tuples used during filling up of item buffers - */
typedef struct ib_tuple
{ unsigned short src_pe;
  unsigned short dest_pe;
  byte  dest_sub_pixel;
  byte  south;
  byte  east;
  float  data;
} IB_Tuple;

typedef union {float f; int i; } float_bits;

/*================================Extern Defs=============================*/

extern char*            malloc();
```

dist.m for Balanced-Load Env-Proj Approach

```
/***************************************************************************

  Copyright 1991: Amitabh Varshney, UNC CS Dept. All Rights Reserved

                                  DIST.M

              Approach: Balanced-Load Environment-Projection


   This part of the code is responsible for distributing the energies


***************************************************************************/
#include <mpl.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include "host.h"

extern struct timeval tm; /* timing variables */
extern struct timezone tz;

extern int maxPEpoly; /* max no of polys per PE */
extern plural Poly*  old_poly_list;  /* Polys to be used */
extern plural int oldPoly; /*No of polys in old_poly_list*/
/* Form factor array */
extern plural float form_factors;
extern float MinRad;


/*************************************************************************
                              dist_energy

  Carry out the distribution of energies from patch to patch within
  the current cell. This assumes that the form-factors have been
  calculated and the polygons are properly arranged with 1 polygon
  per PE.

*************************************************************************/
dist_energy()
{ float max_unshot; /* Maximum unshot energy */
  float last_unshot = HUGEF;
float shooting_rad[3]; /* Shooting patch radiosity */
float shooting_area; /* Area of the shooting-patch */
float current_unshot;
float leakage[3];
float leakage_factor; /* Fraction of energy leaking from
 beneath the single plane */
int i, j, iteration = 0;
plural float fraction;
plural float poly_unshot;
plural float temp;
plural Poly* distPoly; /* Poly used in distribution*/

double et;
double cumulative_time = 0;

distPoly = old_poly_list;
if (distPoly->area < 1.) distPoly->area = 1.0; /* Clamp all areas from below*

/* Initialize the radiosities */
VEC_ASSIGN(distPoly->gather, distPoly->unshot);

VEC_ASSIGN_ZERO(leakage);

START
do
{ poly_unshot = VEC_SUM(distPoly->unshot);
  max_unshot = reduceMaxf(poly_unshot);
  current_unshot = reduceAddf(poly_unshot);
  if (last_unshot >= 2*max_unshot)
  { last_unshot = max_unshot;
    STOP
    cumulative_time += et;
    fprintf(stderr,"Iteration %d total unshot %f max unshot %f time %5.2f sec
    START
  }

  /* In determining the shooting patch ensure that only one PE is active
     and then find the id of the polygon on that PE.
  */
  if (poly_unshot == max_unshot)
  { i = selectOne();
    shooting_area = proc[i].distPoly->area;
    VEC_ASSIGN(shooting_rad, proc[i].distPoly->unshot);
    VEC_ASSIGN_ZERO(proc[i].distPoly->unshot);
  }
```

```
    /* Calculate form factors for this shooting patch */
    calculate_form_factors(i);

    /* Calculate fraction of the energy received by each patch and then
       update the radiosity values of that patch.
    */
    if (distPoly->id >= 0)
    { fraction = form_factors*shooting_area/distPoly->area;

      for (j = 0; j < 3; j++)
      { temp = distPoly->rho[j]*fraction*shooting_rad[j];
        distPoly->unshot[j] += temp;
        distPoly->gather[j] += temp;
      }
    }
    iteration++;
    leakage[RED]   += shooting_rad[RED]*shooting_area;
    leakage[GREEN] += shooting_rad[GREEN]*shooting_area;
    leakage[BLUE]  += shooting_rad[BLUE]*shooting_area;

  }
  /* while (VEC_SUM(shooting_rad) > MinRad); */

  /* Find the energy lost from beneath the single-plane */
  leakage_factor = 1.0 - reduceAddf(form_factors);
  leakage[RED]   *= leakage_factor;
  leakage[GREEN] *= leakage_factor;
  leakage[BLUE]  *= leakage_factor;

  poly_unshot = VEC_SUM(distPoly->unshot);
  max_unshot = reduceMaxf(poly_unshot);
  current_unshot = reduceAddf(poly_unshot);
  STOP
  cumulative_time += et;
  fprintf(stderr,"Iteration %d total unshot %f max unshot %f time %5.2f secs cm
  fflush(stderr);

  /* Add the ambient component to the final solution computed */
  add_ambient(distPoly, leakage);

}

/**********************************************************************
                              add_ambient
  This routine adds an ambient component to the radiosity solution.
  In addition to the unshot energy with the individual polygons, this
  includes in the energy leaking out from under the single plane also
  in calculating the ambient term.

*********************************************************************/

add_ambient(distPoly, leakage)
plural Poly* distPoly;
float leakage[3];
{ float total_area;
  plural float global_formfactor;
  float global_av_rho[3];
  float ambient[3];


  total_area = reduceAddf(distPoly->area);
  global_formfactor = distPoly->area/total_area;

  /* find the global average reflectivity */
  global_av_rho[RED]   = reduceAddf(distPoly->rho[RED]*distPoly->area)
       /total_area;
  global_av_rho[GREEN] = reduceAddf(distPoly->rho[GREEN]*distPoly->area)
       /total_area;
  global_av_rho[BLUE]  = reduceAddf(distPoly->rho[BLUE]*distPoly->area)
       /total_area;

  ambient[RED]   = (leakage[RED]/total_area  +
    reduceAddf(distPoly->unshot[RED]*global_formfactor))
          /(1.0 - global_av_rho[RED]);
  ambient[GREEN] = (leakage[GREEN]/total_area +
   reduceAddf(distPoly->unshot[GREEN]*global_formfactor))
          /(1.0 - global_av_rho[GREEN]);
  ambient[BLUE]  = (leakage[BLUE]/total_area +
   reduceAddf(distPoly->unshot[BLUE]*global_formfactor))
          /(1.0 - global_av_rho[BLUE]);

  fprintf(stderr,"ambient component is %f %f %f\n",ambient[RED],
  ambient[GREEN], ambient[BLUE]);

  distPoly->gather[RED]   += distPoly->rho[RED]*ambient[RED];
  distPoly->gather[GREEN] += distPoly->rho[GREEN]*ambient[GREEN];
  distPoly->gather[BLUE]  += distPoly->rho[BLUE]*ambient[BLUE];
}
```

ff.m for Balanced-Load Env-Proj Approach

```
/**************************************************************************
  Copyright 1991: Amitabh Varshney, UNC CS Dept. All Rights Reserved

                                FF.M

             Approach: Balanced-Load Environment-Projection

  This version of ff.m calculations spreads out polygons in a balanced
  fashion over the whole grid without any regards to the geometry of
  the input dataset. Then a single-plane of a prespecified side and
  height and resolution is used to perform environment-projection.


**************************************************************************/

#include <mpl.h>
#include <stdio.h>
#include <math.h>
#include "host.h"

#include <sys/time.h>

#define GET_RIGHT_NEIGHBOR(from, to) \
to = xnetE[1].from; \
if (ixproc == nxproc - 1) to = xnetS[1].to;

extern struct timeval  tm;
extern struct timezone tz;
extern double          et;

extern plural int oldPoly; /* Polygon buffer */
extern plural Poly *old_poly_list; /* List of patches to be used*/
extern int maxPEpoly; /* Max no of polygons perPE */
/* Form-factor array */
plural float form_factors;
plural PolyTag tag;
plural IB_Tuple final_item[PIXELS]; /* Buffer containing the final
   item-buffers (after
   z-buffering) */
plural unsigned short num_items[PIXELS]; /* Number of items
plural IB_Tuple temp_item[MAX_OVERLAP][PIXELS]; /* Buffers used in the
        intermediate stages of scan-conversion on
```

```
     projection-plane */
plural IB_Tuple         item;
float Plane_Side = 3.0; /*For details refer [Recker90]*/
float Plane_Height = 1.0;
float x_pixel_extent; /* Extent of a single-plane
float y_pixel_extent;    pixel */
plural float canonical_ff[PIXELS]; /* Delta form-factor for each
   pixel */

/**********************************************************************
                       initialize_form_factors

  This function intializes the data structures for use in ff
  calculation.

**********************************************************************/
initialize_form_factors()
{
  plural Poly*  tempPoly; /* Polygon pointer */
  float      xyz[3][3];
  register  int i, j;  /* Misc counters */
  plural float center[PIXELS][3];
  plural float projected_area[PIXELS];
  float sigma_ff = 0;

START

  x_pixel_extent = PIX_ROW*(NXPROC - 1)/(2*Plane_Side);
  y_pixel_extent = PIX_ROW*(NYPROC - 1)/(2*Plane_Side);

  /* Initialize the three axes vectors */
  for(i = 0; i < 3; i++)
  { VEC_ASSIGN_ZERO(xyz[i]);
    xyz[i][i] = 1.0;
  }

  /* Initialize the form factor matrices at each processor */
  form_factors = 0.0;

  /* Initialize the canonical form factor fractions */

  for (i = 0; i < PIXELS; i++)
  { center[i][X] = Plane_Side*(2*(ixproc + (1 + (i % PIX_ROW))/(PIXELS*1.0))/nx
    center[i][Y] = Plane_Side*(2*(iyproc + (1 + (i / PIX_ROW))/(PIXELS*1.0))/ny
    center[i][Z] = Plane_Height;
    /* Area of each hemicube pixel = total area/ total no of pixels
       = (2*Plane_Side)^2 / (PIXELS*nproc) ; as each side of
       hemiplane is from -Plane_Side to Plane_Side
    */
    projected_area[i] = DOT(center[i], center[i]);
    projected_area[i] = 4*Plane_Side*Plane_Side/
(PIXELS*nproc*M_PI*projected_area[i]*projected_area[i]);
  }

  for(j = 0; j < PIXELS; j++)
    canonical_ff[j] = projected_area[j];

  /* Assign the id's to polygons on PEs that have >= 1 polygon */
  /* Assumes MAXPOLYS_PER_CELL <= 4k */
  old_poly_list->id = (oldPoly)? iproc: -1;

  tempPoly = old_poly_list;

  /* Compute the tag data for PEs having valid polys*/
  if (tempPoly->id >= 0)
  { register plural int numverts, m;
    plural float        normal[3];
    register plural int l0;

    numverts = tempPoly->numverts;

    if (numverts != 4 & numverts != 3)
    { p_printf("ff bad numverts = %d\n",numverts);
      exit(-1);
    }

    /* Find the center of the polygon */
    VEC_ASSIGN_ZERO(tag.poly_center);
    for (m = 0; m < numverts; m++)
      VEC_ADD(tag.poly_center, tag.poly_center, tempPoly->verts[m]);

    tag.poly_center[X] /= numverts;
    tag.poly_center[Y] /= numverts;
    tag.poly_center[Z] /= numverts;

    /* Find the orientation of the polygon */
    VEC_ASSIGN(normal, tempPoly->eq);
    normal[X] = (normal[X] < 0)? -normal[X] : normal[X];
    normal[Y] = (normal[Y] < 0)? -normal[Y] : normal[Y];
    normal[Z] = (normal[Z] < 0)? -normal[Z] : normal[Z];
    /* Find the axis that is most nearly perpendicular to the normal */
    if ((normal[X] <= normal[Y]) & (normal[X] <= normal[Z]))
      10 = X;

    if ((normal[Y] <= normal[X]) & (normal[Y] <= normal[Z]))
      10 = Y;

    if ((normal[Z] <= normal[X]) & (normal[Z] <= normal[Y]))
      10 = Z;

    /* Rotate the axis most nearly perpendicular to normal to x-axis */
    CROSS(tag.poly_mat[0], xyz[10], tempPoly->eq);
    PLURAL_NORMALIZE(tag.poly_mat[0], tag.poly_mat[0]);

    /* Rotate the cross product of normal and poly_mat[0] to y-axis */
    CROSS(tag.poly_mat[1], tempPoly->eq, tag.poly_mat[0]);
    /* Rotate the normal to the z-axis */
    VEC_ASSIGN(tag.poly_mat[2], tempPoly->eq);
  }
  else tempPoly->numverts = 0;

STOP
  fprintf(stderr,"Time for initialization %5.2f\n",et);

}

/************************************************************************
                       calculate_form_factors

        This function calculates the form-factor row for the
  shooting patch i. The four vertices of the polygon are projected
  on to the single-plane. Their extents are taken and this area of
  projection is filled-up first along one and then along the other
  dimension.

************************************************************************/
calculate_form_factors(i)
register int i;
{
  plural Poly*  tempPoly; /* Polygon pointer */
  PolyTag shooter_tag;
  register plural float direction[3];
  register plural float unitvec[3];
  register plural float canonical_dir[3];
register plural float extent[4];
register plural int pix[4];
register plural float dist;
register plural float magnitude;
register plural int m, n, invalid, count;
register plural short j, new_j, q, new_q;
register int p, equinum;
register plural byte init_num_items[PIXELS];

tempPoly = old_poly_list;
form_factors = 0.0;

VEC_ASSIGN(shooter_tag.poly_center, proc[i].tag.poly_center);
VEC_ASSIGN(shooter_tag.poly_mat[0], proc[i].tag.poly_mat[0]);
VEC_ASSIGN(shooter_tag.poly_mat[1], proc[i].tag.poly_mat[1]);
VEC_ASSIGN(shooter_tag.poly_mat[2], proc[i].tag.poly_mat[2]);

extent[MINX] = extent[MINY] = Plane_Side;
extent[MAXX] = extent[MAXY] = -Plane_Side;
dist = 0.0;
invalid = 0;

for(j = 0; j < tempPoly->numverts; j++)
{
  VEC_ADD(direction, tempPoly->verts[j], -shooter_tag.poly_center);

  magnitude = DOT(direction, direction);

  if (magnitude > L1_NORM_MIN)
  { dist += magnitude = fp_sqrt(magnitude);

    /* Find the unit direction to each vertex */
    unitvec[0] = direction[0]/magnitude;
    unitvec[1] = direction[1]/magnitude;
    unitvec[2] = direction[2]/magnitude;

    /* Transform this unit vector to the normal space of shooter polygon */
    canonical_dir[X] = DOT(shooter_tag.poly_mat[0], unitvec);
    canonical_dir[Y] = DOT(shooter_tag.poly_mat[1], unitvec);
    canonical_dir[Z] = DOT(shooter_tag.poly_mat[2], unitvec);

    /* Find the extents of poly on the hemi-plane at z = Plane_Height */
    if (canonical_dir[Z] > L1_NORM_MIN)  /* Consider only +ve direction*/
    { canonical_dir[X] = canonical_dir[X]*Plane_Height/canonical_dir[Z];
      canonical_dir[Y] = canonical_dir[Y]*Plane_Height/canonical_dir[Z];
```

```
extent[MINX] = MIN(extent[MINX], canonical_dir[X]);
        extent[MINY] = MIN(extent[MINY], canonical_dir[Y]);
        extent[MAXX] = MAX(extent[MAXX], canonical_dir[X]);
        extent[MAXY] = MAX(extent[MAXY], canonical_dir[Y]);
      }
      else invalid = 1;
    }
  }

  if (tempPoly->numverts > 0)
    dist /= tempPoly->numverts;

  if ((extent[MINX] < -Plane_Side) | (extent[MAXX] > Plane_Side) | (extent[MINY
      (extent[MAXY] > Plane_Side) | (extent[MINX] > extent[MAXX]) | (extent[MIN
    invalid = 1;

  /* Find the bounding item buffer pixels to which data will need to be sent */
  pix[MINX] = (plural int)((extent[MINX] + Plane_Side) * x_pixel_extent);
  pix[MINY] = (plural int)((extent[MINY] + Plane_Side) * y_pixel_extent);
  pix[MAXX] = (plural int)((extent[MAXX] + Plane_Side) * x_pixel_extent);
  pix[MAXY] = (plural int)((extent[MAXY] + Plane_Side) * y_pixel_extent);

  /* Set up the transmission tuple */
  item.dest_pe = pix[MINX]/PIX_ROW + pix[MINY]*nxproc/PIX_ROW;
  item.src_pe  = iproc;
  item.south   = pix[MAXY] - pix[MINY];
  item.east    = pix[MAXX] - pix[MINX];
  item.data    = dist;
  item.dest_sub_pixel = pix[MINX] % PIX_ROW + (pix[MINY] % PIX_ROW)*PIX_ROW;

  all for (i = 0; i < PIXELS; i++)
  { temp_item[0][i].data = final_item[i].data = HUGEF;
    temp_item[0][i].east = temp_item[0][i].south = 0;
  }

  m = 0;
  if (!invalid)
  { sp_rsend(item.dest_pe,(plural char*)&item,
     (plural char* plural)&(temp_item[0][item.dest_sub_pixel]),
     sizeof(IB_Tuple));
    router[item.dest_pe].m += 1;
  }

  /* Find the number of balanced tuples on each PE */
  equinum = reduceMax32(m);
  if (equinum >  1) fprintf(stderr,"max rout m %d ",equinum);

  /* Initialize num_items */
  for(i = 0; i < PIXELS; i++) init_num_items[i] = num_items[i] = (temp_item[0][

  /* Distribute the item-buffers south */
  all for(i = 0; i < PIXELS; i++)  /* Do for each virtual pixel */
  { q = new_q = i;
    j = new_j = init_num_items[i] - 1; /* (m > 0) ? 0 : -1;  */

    count = (j < 0)? j : temp_item[j][q].south;
       /* Start moving this ib_tuple south if this is valid */
    all while (count > 0)
    { count--;
      if (new_j > MAX_OVERLAP)
p_printf("iproc %d, overlap %d\n",iproc,new_j);
      q = new_q;
      if (temp_item[j][q].data < final_item[q].data)/* Do z-buffering */
      { final_item[q].data   = temp_item[j][q].data;
        final_item[q].src_pe = temp_item[j][q].src_pe;
      }
      new_q = (q + PIX_ROW) % PIXELS;
/* Locate the virtual south neighbor */
      if (new_q > q)  /* We don't need to xsend to south PE yet */
      { new_j = num_items[new_q]++;
        temp_item[new_j][new_q].src_pe = temp_item[j][q].src_pe;
        temp_item[new_j][new_q].east   = temp_item[j][q].east;
        temp_item[new_j][new_q].data   = temp_item[j][q].data;
        j = new_j;
      }
      else /* xsend to the south PE */
      { if (iyproc < NYPROC_1) /* avoid wraparound */
{ xnetS[1].count = count;
  xnetS[1].new_q = new_q;
  new_j = xnetS[1].num_items[new_q];
  xnetS[1].num_items[new_q] = new_j + 1;
  new_q = xnetS[1].new_q; /* regenerate new_q for next step only */
          pp_xsend(-1, 0, (plural char* plural)&temp_item[j][q],
  (plural char* plural)&temp_item[new_j][new_q], sizeof(IB_Tuple));
  xnetS[1].new_q = new_q; /* restore new_q now*/
  xnetS[1].j = new_j;
        }
else /* disable this ib_tuple */
  count = 0;
      }
    }
  }


  /* Distribute the item-buffers east */
  all
  for(i = 0; i < PIXELS; i++) /* Do for each virtual pixel */
  { q = i;
    for(j = 0; j < num_items[i]; j++) /* Do for each ib_tuple */
    { count = temp_item[j][i].east;
      item.data = temp_item[j][i].data;
      item.src_pe = temp_item[j][i].src_pe;
      all while (count > 0)
      { count--;
        if (item.data < final_item[q].data)/* Do z-buffering */
        { final_item[q].data   = item.data;
          final_item[q].src_pe = item.src_pe;
        }
        new_q = (q + 1) % PIX_ROW;
if (new_q > q) /* We do not need to xsend to east PE */
         q = new_q;
else  /* We need to xsend to east PE */
{ ss_xsend(0, 1, (plural char*)&item, (plural char*)&item, sizeof(IB_Tuple));
  xnetE[1].count = count;
  xnetE[1].q = new_q;
        }
      }
    }
  }

  /* Update the form factors for the finally selected PEs */
  form_factors = 0;
  for(i = 0; i < PIXELS; i++)
  { if (final_item[i].data < HUGE)
      dist = canonical_ff[i];
    else
    { dist = 0;
      final_item[i].src_pe = iproc;
    }
    form_factors += sendwithAddf(dist, final_item[i].src_pe);
  }
}
```

dist.m for Object-Space Ray-Casting Approach

```
/***********************************************************************
               Amitabh Varshney          Howard Good

                                  DIST.M

                     Approach: Object-Space Ray-Casting

   This part of the code is responsible for distributing the energies
  from local and global iterations.

***********************************************************************/
#include <mpl.h>
#include <stdio.h>
#include <math.h>
#include "host.h"

extern int maxPEpoly; /* max no of polys per PE */
extern plural Poly*  old_poly_list;  /* Polys to be used */
extern plural int oldPoly; /*No of polys in old_poly_list*/
/* Form factor array */
extern plural byte form_factors[MAXPEPOLYS][MAXROWPOLYS];
/* Rays stored in walls */
extern plural Ray wall_rays[NUMWALLS][MAXRAYS];
/* No of rays in each wall */
extern plural int wall_count[NUMWALLS];


/**********************************************************************
                           dist_pp

  Carry out the distribution of energies from patch to patch within
  each virtual room.

**********************************************************************/
dist_pp()
{ plural Poly* plural  tempPoly;
  plural Ray tempRay;
  plural int i,j;
  int n,r;
  plural float  procRadSum = 0.0;
  plural float shootingEnergy;
  plural float tempEnergy;
  plural Packet shootingPatch;
  plural int shootingPatchValid = 0;
  plural int shooter;
  plural float deltaRad,tempRad;
  float totalUnshotRad;

  /* Initialize procRadSum */
  for(i = 0; i < oldPoly; i++)
  { tempPoly = old_poly_list + i;
    for (n = 0; n <3; n++)
      procRadSum += tempPoly->unshotRad[n];
  }
  totalUnshotRad = reduceAddf(procRadSum);
  printf("Total unshotRad radiosity = %f\n",totalUnshotRad);
  fflush(stdout);

  /* Start energy xfer */
  while (totalUnshotRad > TOL)
  {
    shooter = -1;
    shootingEnergy = 0;
    shootingPatchValid = 0;

    /* Choose the shooting patch on each PE as the brightest patch */
    for (i = 0; i < oldPoly; i++)
    { tempPoly = old_poly_list + i;
      tempEnergy = tempPoly->unshotRad[RED]+
   tempPoly->unshotRad[GREEN]+
   tempPoly->unshotRad[BLUE];
      if (tempEnergy > shootingEnergy)
      { shootingEnergy = tempEnergy;
shooter = i;
      }
    }

    /* If a shooting patch exists, compose an energy packet */
    if (shooter >= 0)
    { tempPoly = old_poly_list + shooter;
      shootingPatch.unshot[RED] = tempPoly->unshotRad[RED];
      shootingPatch.unshot[GREEN] = tempPoly->unshotRad[GREEN];
      shootingPatch.unshot[BLUE] = tempPoly->unshotRad[BLUE];
      shootingPatch.area = tempPoly->area;
      shootingPatch.id = tempPoly->id;
      shootingPatchValid = 1;
    }
    if (shootingEnergy > 0)
  p_printf("Shooter %d on [%d,%d], energy = %f\n",shootingPatch.id,
ixproc, iyproc, shootingEnergy);


    /* Transmit the energy packet formed above from processor to processor
       distributing its energy to all polygons on those processors as
       determined by the form-factors.
    */
    for (r = 0; r<nxproc; r++ )
    { if(shootingPatchValid == 1)
      { shootingPatchValid = 0;
        for(i=0; i<oldPoly; i++)
        { tempPoly = old_poly_list + i;
  tempRad = (plural float) form_factors[i][shootingPatch.id]*
     shootingPatch.area/(255.0*255.0*tempPoly->area);
          for(n=0; n<3; n++)
  { deltaRad = tempRad*(plural float)(tempPoly->colors[0][n])*
                         shootingPatch.unshot[n];
    tempPoly->unshotRad[n] += deltaRad;
    tempPoly->totalRad[n] += deltaRad;
          }
        }
        ss_xsend(0,1,&shootingPatch, &shootingPatch, sizeof(Packet));
        xnetE[1].shootingPatchValid = 1;
      }
    }

    /* Set the unshot radiosity of the shooting patch to zero */
    if (shooter >= 0)
    { tempPoly = old_poly_list + shooter;
      tempPoly->unshotRad[RED]=
      tempPoly->unshotRad[GREEN]=
      tempPoly->unshotRad[BLUE]=0.0;
    }

    /* Compute the total remaining energy in the system */
    procRadSum = 0.0;
    for(i = 0; i < oldPoly; i++)
    { tempPoly = old_poly_list + i;
      for (n = 0; n <3; n++)
        procRadSum += tempPoly->unshotRad[n];
    }
    totalUnshotRad = reduceAddf(procRadSum);
    printf("Total unshotRad radiosity = %f\n",totalUnshotRad);
    fflush(stdout);
  }
}


/***********************************************************************
                          dist_pw

  Carry out the distribution of energies from patch to walls within
  each virtual room. This is done by checking out for each polygon
  all the rays in the wall arrays which have originated from this
  polygon and then incrementing their energy values by the energy
  values of the polygon. This has not been tested.
***********************************************************************/
dist_pw()
{ plural int i,j;
  int n;
  plural Poly* plural  tempPoly;

  for(i = 0; i < oldPoly; i++)
  { tempPoly = old_poly_list + i;
    if (tempPoly->unshotRad[RED] +
tempPoly->unshotRad[GREEN] +
tempPoly->unshotRad[RED] > 0.0)
    { for(n = 0; n < NUMWALLS; n++)
      { for(j = 0; j < wall_count[n]; j++)
        { if (tempPoly->id == wall_rays[n][j].id)
          { wall_rays[n][j].energy[RED] +=
tempPoly->unshotRad[RED]*tempPoly->area/NUMRAYS;
            wall_rays[n][j].energy[GREEN] +=
tempPoly->unshotRad[GREEN]*tempPoly->area/NUMRAYS;
            wall_rays[n][j].energy[BLUE] +=
tempPoly->unshotRad[BLUE]*tempPoly->area/NUMRAYS;
          }
        }
      }
    }
  }
}

/***********************************************************************
                          xchg_walls

  This routine exchanges the energy filled walls from one room to the
  other room. This has not been tested.

************************************************************************/
xchg_walls()
{ plural Ray* plural    tempRay;

  /* Xchg N wall ie move the wall North */
  if (iyproc > XROOM_DIM)
  { ss_xsend(XROOM_DIM, 0, &wall_rays[NORTH][0],&wall_rays[NORTH][0],
     MAXRAYS*sizeof(Poly));
    xnetN[XROOM_DIM].wall_count[NORTH] = wall_count[NORTH];
  }

  /* Xchg S wall */
  if (iyproc < nyproc - XROOM_DIM)
             /* Send it XROOM_DIM rows down */
  { ss_xsend(-XROOM_DIM, 0, &wall_rays[SOUTH][0], &wall_rays[SOUTH][0],
     MAXRAYS*sizeof(Poly));
    xnetS[XROOM_DIM].wall_count[SOUTH] = wall_count[SOUTH];
  }

  /* Xchg E wall */
  if ((iyproc+1)%XROOM_DIM > 0)
           /* Send it 1 down */
  { ss_xsend(1, 0,&wall_rays[EAST][0],&wall_rays[EAST][0],MAXRAYS*sizeof(Poly))
    xnetS[1].wall_count[EAST] = wall_count[EAST];
  }

  /* Xchg W wall */
  if (iyproc % XROOM_DIM > 0)
  /* Send it 1 up */
  { ss_xsend(-1,0,&wall_rays[WEST][0],&wall_rays[WEST][0],MAXRAYS*sizeof(Poly))
    xnetN[1].wall_count[WEST] = wall_count[WEST];
  }
}
```

ff.m for Object-Space Ray-Casting Approach

```
/*************************************************************************
              Amitabh Varshney            Howard Good

                                  FF.M

        Approach: Object-Space Ray-Casting

   This part of the code is responsible for doing much of one time
  processing required to get the radiosity iterations going. This
  involves setting up of the virtual walls and calculating the form
  factors.

*************************************************************************/

#include <mpl.h>
#include <math.h>
#include "host.h"


extern float MinX, MinY, MinZ; /* Dataset extents */
extern float MaxX, MaxY, MaxZ;
extern plural int oldPoly; /* Polygon buffer */
extern plural Poly *old_poly_list; /* List of patches to be used*/
extern int maxPEpoly; /* Max no of polygons perPE */
extern float xroomBound[XROOM_DIM];  /* Room bounds in X, Y and Z */
extern float yroomBound[YROOM_DIM];
extern float zroomBound[ZROOM_DIM];


/* Form factor array at each processor*/
plural byte form_factors[MAXPEPOLYS][MAXROWPOLYS];
/*Rays hitting walls at each processor*/
plural Ray wall_rays[NUMWALLS][MAXPEPOLYS*NUMRAYS];
/* No of rays stored for each  wall */
plural int wall_count[NUMWALLS];


/************************************************************************
                        calculate_form_factors

  This function calculates the form-factor matrices of the
  polygons and stores these at each processor. It also computes the
  rays which hit the virtual room walls and stores these in ray
  arrays which are then later used for energy distribution in
  radiosity iterations
*************************************************************************/
calculate_form_factors()
{
  int n; /* Misc counter */
  plural Poly* plural tempPoly; /* Polygon pointer */
  plural Ray testRay; /* Ray buffer */
  plural float normal[3]; /* Normal to the polygon */
  plural int patch_id; /* Id of the patch */
  plural byte wall_id; /* Id of the wall - N,S,E,W,F,C */
  plural int i,j,k,r; /* Misc counters */
  plural float ff;
  plural int is,it;
  plural float s,t;
  plural int ff_count;


  /* Initialize the form factor matrices at each processor */
  for (i=0; i < MAXPEPOLYS; i++)
  { for(j=0; j<MAXROWPOLYS; j++)
      form_factors[i][j] = (plural byte) 0.0;
  }


  /* Set up the walls */
  init_walls();

  /* Intialize the wall - ray arrays */
  ff_count = 0;
  for (i = 0; i < NUMWALLS; i++)
    wall_count[i] = 0;


  /* Assign the patch id's */
  patch_id = 0;
  for (n = 0; n < nxproc; n++)
  { if (ixproc == n)
    { for (i = 0; i < oldPoly; i++)
      { tempPoly = old_poly_list + i;
        tempPoly->id = patch_id++;
      }
      xnetE[1].patch_id = patch_id;
    }
    }


    for (i = 0; i < oldPoly; i++)
    { tempPoly = old_poly_list + i;

      normal[X] = tempPoly->eq[A];
      normal[Y] = tempPoly->eq[B];
      normal[Z] = tempPoly->eq[C];

      all testRay.id = -1; /* to indicate invalid rays */

      for (r = 0; r < NUMRAYS; r++)
      { testRay.origin[X] = testRay.origin[Y] = testRay.origin[Z] = 0.0;
        testRay.distance = HUGEF;
        testRay.id = tempPoly->id;

        /* set ray origin to patch center */
        for (j = 0; j < tempPoly->numverts; j++)
        { testRay.origin[X] += tempPoly->verts[j][X];
          testRay.origin[Y] += tempPoly->verts[j][Y];
          testRay.origin[Z] += tempPoly->verts[j][Z];
        }
        testRay.origin[X] /= tempPoly->numverts;
        testRay.origin[Y] /= tempPoly->numverts;
        testRay.origin[Z] /= tempPoly->numverts;


        /* set ray direction */
        testRay.direction[X] = normal[X];
        testRay.direction[Y] = normal[Y];
        testRay.direction[Z] = normal[Z];


        for (k = 0; k < nxproc; k++)
        {
          /* intersect ray with all polygons and virtual walls */
all if (testRay.id >= 0) intersect_ray(&testRay);

          ss_xsend(0,1,&testRay,&testRay,sizeof(Ray));
        }

        if (testRay.id >= nxproc*maxPEpoly + NUMWALLS)
          p_printf("bad id = %d\n",testRay.id);
        else if (testRay.id >= nxproc*maxPEpoly) /* Intersection is with a wall*/
      { wall_id = testRay.id - nxproc*maxPEpoly;
/* Store the ray into the buffer for appropriate wall */
if (wall_id < NUMWALLS)
        { j = wall_count[wall_id];
          wall_rays[wall_id][j].id = tempPoly->id;
          wall_rays[wall_id][j].distance = testRay.distance;
          for (k = 0; k<3; k++)
          { wall_rays[wall_id][j].origin[k] = testRay.origin[k];
            wall_rays[wall_id][j].energy[k] = 0.0;
            wall_rays[wall_id][j].direction[k] = testRay.direction[k];
          }
        }
        wall_count[wall_id]++;

  p_printf("shooter %d hit wall %d\n",tempPoly->id,wall_id);
      }
      else  /* Intersection is with a model poly*/
      if (testRay.id >= 0 && testRay.distance < HUGEF)
      { ff_count++;
        form_factors[i][testRay.id]++;
  p_printf("shooter %d hit poly %d\n",tempPoly->id,testRay.id);
      }

      if (testRay.distance >= HUGEF)
      { p_printf("x,y = %d,%d; origin = %g,%g,%g; direction = %g,%g,%g\n",
  ixproc,iyproc,testRay.origin[X], testRay.origin[Y], testRay.origin[Z],
  testRay.direction[X], testRay.direction[Y], testRay.direction[Z]);
      }
    }
  }


  if (oldPoly*NUMRAYS > 255)
  { p_printf("proc %d %d:  too many rays\n",ixproc,iyproc,oldPoly*NUMRAYS);
    exit(-1);
  }

  /* scale form factors */
  if (oldPoly > 0)
  { for (i = 0; i < oldPoly; i++)
    { for(j=0; j<MAXROWPOLYS; j++)
      { ff = (plural float) form_factors[i][j];
        ff /= NUMRAYS;
        ff *= 255.0;
        form_factors[i][j] = (plural byte) ff;
      }
    }
  }

  /* Verify that the rays are conserved */
  for (i = 0; i < NUMWALLS; i++)
    ff_count += wall_count[i];

  if (ff_count != oldPoly*NUMRAYS)
    p_printf("proc %d %d:  bad ray count = %d oldPoly = %d;\n",ixproc,iyproc,
    ff_count,oldPoly);
}


/**********************************************************************
                        intersect_ray

  This function would intersect the given ray with all the
  polygons present on the row to which it belongs and returns the id
  of the polygon it finds has the closest intersection point in the
  id field of the ray
**********************************************************************/
intersect_ray(testRay)
plural Ray *testRay; /* Input ray */
{
  plural Poly* plural tempPoly; /* Polygon buffer */
  plural int i,j,k,l; /* Misc counters */
  plural float normal[3]; /* Normal */
  plural float ndotd; /*Dot prod of normal and ray direction*/
  plural float p[3]; /*Temporaries */
  plural float d,t;
  plural byte l0,l1,l2;
  plural byte inter;
  plural float alpha,beta;
  plural float u0,u1,u2;
  plural float v0,v1,v2;


  for (j = 0; j < oldPoly+NUMWALLS; j++)
  { tempPoly = old_poly_list + j;

    if (tempPoly->numverts != 4 && tempPoly->numverts != 3)
    { p_printf("ff bad numverts = %d\n",tempPoly->numverts);
      exit(-1);
    }
    normal[X] = tempPoly->eq[A];
    normal[Y] = tempPoly->eq[B];
    normal[Z] = tempPoly->eq[C];
    d = tempPoly->eq[D];

    /* Compute the dot product of the ray direction with the poly normal */
    ndotd = normal[X]*testRay->direction[X] +
    normal[Y]*testRay->direction[Y] +
            normal[Z]*testRay->direction[Z];

    if (ndotd == 0.0) continue;

    t = -(d + normal[X]*testRay->origin[X] + normal[Y]*testRay->origin[Y] +
      normal[Z]*testRay->origin[Z])/ndotd;

    if (t <= 0.002) continue;
/* Avoid intersecting with originating polygon*/
/* Polygon is beyond closest intersection found so far*/
    if (t >= testRay->distance) continue;

    /* Calculate the point of intersection */
    p[X] = testRay->origin[X] + testRay->direction[X]*t;
    p[Y] = testRay->origin[Y] + testRay->direction[Y]*t;
    p[Z] = testRay->origin[Z] + testRay->direction[Z]*t;

    /* Find the orientation of the polygon */
    normal[X] = fp_fabs(normal[X]);
    normal[Y] = fp_fabs(normal[Y]);
    normal[Z] = fp_fabs(normal[Z]);

    if (normal[X] >= normal[Y] && normal[X] >= normal[Z])
    { l1 = Y;
      l2 = Z;
    }
    if (normal[Y] >= normal[X] && normal[Y] >= normal[Z])
    { l1 = X;
      l2 = Z;
    }
    if (normal[Z] >= normal[X] && normal[Z] >= normal[Y])
    { l1 = X;
      l2 = Y;
    }

    /* Verify if the point of intersection is within the polygon */
    u0 = p[l1] - tempPoly->verts[0][l1];
    v0 = p[l2] - tempPoly->verts[0][l2];

    inter = 0;
    l = 2;

    do {
      u1 = tempPoly->verts[l-1][l1] - tempPoly->verts[0][l1];
      v1 = tempPoly->verts[l-1][l2] - tempPoly->verts[0][l2];
      u2 = tempPoly->verts[l  ][l1] - tempPoly->verts[0][l1];
      v2 = tempPoly->verts[l  ][l2] - tempPoly->verts[0][l2];

      if (u1 == 0)
      { beta = u0/u2;
        if (beta >= 0.0 && beta <= 1.0)
        { alpha = (v0 - beta*v2)/v1;
          inter = (alpha >= 0.0 && alpha+beta <= 1.0);
        }
      }
      else
      { beta = (v0*u1 - u0*v1)/(v2*u1 - u2*v1);
        if (beta >= 0.0 && beta <= 1.0)
        { alpha = (u0 - beta*u2)/u1;
  inter = (alpha >= 0.0 && alpha+beta <= 1.0);
        }
      }
    } while (!inter && ++l < tempPoly->numverts) ;

    if (inter)
    { testRay->distance = t;
      if (j < oldPoly)
        testRay->id = tempPoly->id;/* intersection was with a model polygon */
      else      /* intersection was with a virtual wall */
testRay->id = nxproc*maxPEpoly + (j-oldPoly);
    }
  }
}


/**********************************************************************
                        init_walls

  Sets up  the virtual wall polygons at all the processors
**********************************************************************/
init_walls()
```

```
{
  plural Poly* plural tempPoly;
  plural byte xroom,yroom;

  xroom = iyproc/YROOM_DIM;
  yroom = iyproc%YROOM_DIM;

  /* North wall */
  tempPoly = old_poly_list + oldPoly + NORTH;
  tempPoly->verts[0][X] = MinX + (xroom > 0 ? xroomBound[xroom-1] : 0);
  tempPoly->verts[1][X] = MinX + xroomBound[xroom];
  tempPoly->verts[2][X] = MinX + xroomBound[xroom];
  tempPoly->verts[3][X] = MinX + (xroom > 0 ? xroomBound[xroom-1] : 0);
  tempPoly->verts[0][Y] = MinY + yroomBound[yroom];
  tempPoly->verts[1][Y] = MinY + yroomBound[yroom];
  tempPoly->verts[2][Y] = MinY + yroomBound[yroom];
  tempPoly->verts[3][Y] = MinY + yroomBound[yroom];
  tempPoly->verts[0][Z] = MinZ;
  tempPoly->verts[1][Z] = MinZ;
  tempPoly->verts[2][Z] = MaxZ;
  tempPoly->verts[3][Z] = MaxZ;
  tempPoly->eq[A] = 0.0;
  tempPoly->eq[B] = -1.0;
  tempPoly->eq[C] = 0.0;
  tempPoly->eq[D] = MinY + yroomBound[yroom];
  tempPoly->numverts = 4;
  tempPoly->area = 0.0;

  /* South wall */
  tempPoly = old_poly_list + oldPoly + SOUTH;
  tempPoly->verts[0][X] = MinX + xroomBound[xroom];
  tempPoly->verts[1][X] = MinX + (xroom > 0 ? xroomBound[xroom-1] : 0);
  tempPoly->verts[2][X] = MinX + (xroom > 0 ? xroomBound[xroom-1] : 0);
  tempPoly->verts[3][X] = MinX + xroomBound[xroom];
  tempPoly->verts[0][Y] = MinY + (yroom > 0 ? yroomBound[yroom-1] : 0);
  tempPoly->verts[1][Y] = MinY + (yroom > 0 ? yroomBound[yroom-1] : 0);
  tempPoly->verts[2][Y] = MinY + (yroom > 0 ? yroomBound[yroom-1] : 0);
  tempPoly->verts[3][Y] = MinY + (yroom > 0 ? yroomBound[yroom-1] : 0);
  tempPoly->verts[0][Z] = MinZ;
  tempPoly->verts[1][Z] = MinZ;
  tempPoly->verts[2][Z] = MaxZ;
  tempPoly->verts[3][Z] = MaxZ;
  tempPoly->eq[A] = 0.0;
  tempPoly->eq[B] = 1.0;
  tempPoly->eq[C] = 0.0;
```

```
tempPoly->eq[D] = -(MinY + (yroom > 0 ? yroomBound[yroom-1] : 0));
tempPoly->numverts = 4;
tempPoly->area = 0.0;

/* West wall */
tempPoly = old_poly_list + oldPoly + WEST;
tempPoly->verts[0][X] = MinX + (xroom > 0 ? xroomBound[xroom-1] : 0);
tempPoly->verts[1][X] = MinX + (xroom > 0 ? xroomBound[xroom-1] : 0);
tempPoly->verts[2][X] = MinX + (xroom > 0 ? xroomBound[xroom-1] : 0);
tempPoly->verts[3][X] = MinX + (xroom > 0 ? xroomBound[xroom-1] : 0);
tempPoly->verts[0][Y] = MinY + (yroom > 0 ? yroomBound[yroom-1] : 0);
tempPoly->verts[1][Y] = MinY + yroomBound[yroom];
tempPoly->verts[2][Y] = MinY + yroomBound[yroom];
tempPoly->verts[3][Y] = MinY + (yroom > 0 ? yroomBound[yroom-1] : 0);
tempPoly->verts[0][Z] = MinZ;
tempPoly->verts[1][Z] = MinZ;
tempPoly->verts[2][Z] = MaxZ;
tempPoly->verts[3][Z] = MaxZ;
tempPoly->eq[A] = 1.0;
tempPoly->eq[B] = 0.0;
tempPoly->eq[C] = 0.0;
tempPoly->eq[D] = -(MinX + (xroom > 0 ? xroomBound[xroom-1] : 0));
tempPoly->numverts = 4;
tempPoly->area = 0.0;

/* East wall */
tempPoly = old_poly_list + oldPoly + EAST;
tempPoly->verts[0][X] = MinX + xroomBound[xroom];
tempPoly->verts[1][X] = MinX + xroomBound[xroom];
tempPoly->verts[2][X] = MinX + xroomBound[xroom];
tempPoly->verts[3][X] = MinX + xroomBound[xroom];
tempPoly->verts[0][Y] = MinY + yroomBound[yroom];
tempPoly->verts[1][Y] = MinY + (yroom > 0 ? yroomBound[yroom-1] : 0);
tempPoly->verts[2][Y] = MinY + (yroom > 0 ? yroomBound[yroom-1] : 0);
tempPoly->verts[3][Y] = MinY + yroomBound[yroom];
tempPoly->verts[0][Z] = MinZ;
tempPoly->verts[1][Z] = MinZ;
tempPoly->verts[2][Z] = MaxZ;
tempPoly->verts[3][Z] = MaxZ;
tempPoly->eq[A] = -1.0;
tempPoly->eq[B] =  0.0;
tempPoly->eq[C] =  0.0;
tempPoly->eq[D] =  MinX + xroomBound[xroom];
tempPoly->numverts = 4;
tempPoly->area = 0.0;
```

```
  /* Floor */
  tempPoly = old_poly_list + oldPoly + FLOOR;
  tempPoly->verts[0][X] = MinX + (xroom > 0 ? xroomBound[xroom-1] : 0);
  tempPoly->verts[1][X] = MinX + xroomBound[xroom];
  tempPoly->verts[2][X] = MinX + xroomBound[xroom];
  tempPoly->verts[3][X] = MinX + (xroom > 0 ? xroomBound[xroom-1] : 0);
  tempPoly->verts[0][Y] = MinY + (yroom > 0 ? yroomBound[yroom-1] : 0);
  tempPoly->verts[1][Y] = MinY + (yroom > 0 ? yroomBound[yroom-1] : 0);
  tempPoly->verts[2][Y] = MinY + yroomBound[yroom];
  tempPoly->verts[3][Y] = MinY + yroomBound[yroom];
  tempPoly->verts[0][Z] = MinZ;
  tempPoly->verts[1][Z] = MinZ;
  tempPoly->verts[2][Z] = MinZ;
  tempPoly->verts[3][Z] = MinZ;
  tempPoly->eq[A] = 0.0;
  tempPoly->eq[B] = 0.0;
  tempPoly->eq[C] = 1.0;
  tempPoly->eq[D] = -MinZ;
  tempPoly->numverts = 4;
  tempPoly->area = 0.0;

  /* Ceiling */
  tempPoly = old_poly_list + oldPoly + CEILING;
  tempPoly->verts[0][X] = MinX + xroomBound[xroom];
  tempPoly->verts[1][X] = MinX + (xroom > 0 ? xroomBound[xroom-1] : 0);
  tempPoly->verts[2][X] = MinX + (xroom > 0 ? xroomBound[xroom-1] : 0);
  tempPoly->verts[3][X] = MinX + xroomBound[xroom];
  tempPoly->verts[0][Y] = MinY + (yroom > 0 ? yroomBound[yroom-1] : 0);
  tempPoly->verts[1][Y] = MinY + (yroom > 0 ? yroomBound[yroom-1] : 0);
  tempPoly->verts[2][Y] = MinY + yroomBound[yroom];
  tempPoly->verts[3][Y] = MinY + yroomBound[yroom];
  tempPoly->verts[0][Z] = MaxZ;
  tempPoly->verts[1][Z] = MaxZ;
  tempPoly->verts[2][Z] = MaxZ;
  tempPoly->verts[3][Z] = MaxZ;
  tempPoly->eq[A] =  0.0;
  tempPoly->eq[B] =  0.0;
  tempPoly->eq[C] = -1.0;
  tempPoly->eq[D] =  MaxZ;
  tempPoly->numverts = 4;
  tempPoly->area = 0.0;
}
```

io.m for Object-Space Ray-Casting Approach

```
/***************************************************************************
                Amitabh Varshney           Howard Good

     IO.M

             Approach: Object-Space Ray-Casting

    This program runs on the back-end of MasPar and carries out most
  of the tasks including swapping in of the polygons from the from-end
***************************************************************************/

#include <mpl.h>
#include <stdio.h>
#include <math.h>
#include "host.h"

visible extern die();

/*************************************************************************
  GLOBAL DECLS

*************************************************************************/

float   MinX, MinY, MinZ; /* Extents of the input dataset*/
float   MaxX, MaxY, MaxZ;
plural Poly *old_poly_list; /* Lists of polygons */
plural Poly *new_poly_list;
plural int oldPoly; /* Number of polygons in old_poly_list*/
plural int newPolyCount; /* Number of polygons in new_poly_list*/
int maxPEpoly; /* Maximum no of polygons/PE */

float  xroomBound[XROOM_DIM];  /* Upper bounds of each virtual room */
float  yroomBound[YROOM_DIM];
float  zroomBound[ZROOM_DIM];


/*************************************************************************
                              polys_to_pe

   This is equivalent of the main program on the back end. It blocks
  in the polygon dataset and distributes it on the PE's. Then it
  calls the appropriate routines to balance the polygons and start
  radiosity iterations.

*************************************************************************/

visible polys_to_pe(poly_list, total_polys)
Poly *poly_list; /* List of polygons on the front-end*/
int total_polys; /* Total no of polys to be swapped-in*/
{ int numpoly,maxpoly,minpoly;
  int n;
  int initPEpoly;          /* initial no of polygons per PE */

  initPEpoly = (int)(total_polys/nproc + 1);
  maxPEpoly = initPEpoly*6 ;

  fprintf(stdout,"polys %d nproc %d polys/pe %d poly_size %d\n", total_polys,
  nproc,initPEpoly, sizeof(Poly));
  fflush(stdout);

  oldPoly = 0;

  /* Allocate PE memory */
  P_ALLOCN(old_poly_list, Poly, maxPEpoly + NUMWALLS, "polys_to_pe");
  P_ALLOCN(new_poly_list, Poly, maxPEpoly + NUMWALLS, "polys_to_pe");

  /* Read in polygons to PEs from the front end side */
  n = blockIn(poly_list, old_poly_list,
    0, 0, nxproc, nyproc, (initPEpoly-1)*sizeof(Poly));
  n += blockIn(poly_list+(initPEpoly-1)*nproc, old_poly_list+initPEpoly-1,
    0, 0, nxproc, nyproc, sizeof(Poly));
  printf("Num bytes Xferred = %d; num polys = %d\n",n,n/sizeof(Poly));

  /* Determine oldPoly for each PE */
  oldPoly =
    (iproc < total_polys-(initPEpoly-1)*nproc ? initPEpoly : initPEpoly-1);

  numpoly = reduceAdd32(oldPoly);
  maxpoly = reduceMax32(oldPoly);
  minpoly = reduceMin32(oldPoly);
  printf("old polys = %d; max = %d, min = %d\n",numpoly,maxpoly,minpoly);

  /* determine the extents of the dataset - used in balancing */
  determine_extents();

  /* Coarsely balance the dataset */
  balance();
  /* Refine the balance on the DPU by further subdividing if necessary*/
  refine_balance();

  /* Perform the one-time calculations for the form-factor determinations*/
  calculate_form_factors();

  /* Start the radiosity iterations */
  /* Carry out the patch - patch energy exchange*/
  dist_pp();

  /* Carry out the exchange of energy laden virtual walls */
  xchg_walls();
}


/************************************************************************
                           determine_extents

   This routine determines the extents of the polygon dataset.
   It takes into account the coarse cell lengths to determine the
   extents which will be represented on the DPU array once the
   balancing is done.

************************************************************************/
determine_extents()
{ plural int i, j; /* Miscellaneous counters */
  plural float x, y, z; /* Vertex values */
  plural Poly* plural tempPoly; /* Current polygon */
  plural float  minx, miny, minz; /* Local extents on each PE */
  plural float  maxx, maxy, maxz;


  /* Determine the local extents */
  minx = miny = minz = HUGEF;
  maxx = maxy = maxz = -HUGEF;
  for (i = 0; i<oldPoly; i++)
  { tempPoly = old_poly_list + i;

    if (tempPoly->numverts < 3 || tempPoly->numverts > MAXVERT) {
      p_printf("ERROR: proc %d numverts = %d\n",iproc,tempPoly->numverts);
      exit(1);
    }

    for (j = 0; j<tempPoly->numverts; j++)
    { x = tempPoly->verts[j][X];
      y = tempPoly->verts[j][Y];
      z = tempPoly->verts[j][Z];
      minx = MIN(minx,x);
      miny = MIN(miny,y);
      minz = MIN(minz,z);
      maxx = MAX(maxx,x);
      maxy = MAX(maxy,y);
      maxz = MAX(maxz,z);
    }
  }


  /* Determine the global extents */

  MinX = reduceMinf(minx);
  MinY = reduceMinf(miny);
  MinZ = reduceMinf(minz);
  MaxX = reduceMaxf(maxx);
  MaxY = reduceMaxf(maxy);
  MaxZ = reduceMaxf(maxz);

  printf("Minx %6.2f MaxX %6.2f MinY %6.2f MaxY %6.2f MinZ %6.2f MaxZ %6.2f\n",
  MinX,MaxX,MinY,MaxY,MinZ,MaxZ);

  /* Take into account the global patchification grid aligned along the axes*/

  MinX = f_floor(MinX/XCELLLENGTH) * XCELLLENGTH;
  MinY = f_floor(MinY/YCELLLENGTH) * YCELLLENGTH;
  MinZ = f_floor(MinZ/ZCELLLENGTH) * ZCELLLENGTH;
  MaxX = f_ceil(MaxX/XCELLLENGTH) * XCELLLENGTH;
  MaxY = f_ceil(MaxY/YCELLLENGTH) * YCELLLENGTH;
  MaxZ = f_ceil(MaxZ/ZCELLLENGTH) * ZCELLLENGTH;

  printf("Minx %6.2f MaxX %6.2f MinY %6.2f MaxY %6.2f MinZ %6.2f MaxZ %6.2f\n",
  MinX,MaxX,MinY,MaxY,MinZ,MaxZ);
}



/**********************************************************************
                              area

   This routine determines the area of a plural polygon defined by
   the vertex array 'vert' and having 'numverts' number of vertices
   For this the routine considers the polygon to be composed of
   triangles and then computes the area of each triangle by taking
   half of the magnitude of the cross product of two of its sides.

*************************************************************************/
plural float area(verts, numverts)
plural float verts[MAXVERT][3]; /* Vertex array */
plural int numverts; /* Number of vertices */
{ plural int    i,j,k;
  plural float  area = 0.0;
  plural float  v1[3], v2[3], v3[3];

  for(i=0; i < numverts - 2; i++)
  { j = i+1;
    k = i+2;

    v1[X] = verts[j][X] - verts[i][X];
    v1[Y] = verts[j][Y] - verts[i][Y];
    v1[Z] = verts[j][Z] - verts[i][Z];

    v2[X] = verts[k][X] - verts[i][X];
    v2[Y] = verts[k][Y] - verts[i][Y];
    v2[Z] = verts[k][Z] - verts[i][Z];

    v3[X] =  v1[Y]*v2[Z] - v1[Z]*v2[Y];
    v3[Y] = -v1[X]*v2[Z] + v1[Z]*v2[X];
    v3[Z] =  v1[X]*v2[Y] - v1[Y]*v2[X];

    area += 0.5*fp_sqrt(v3[X]*v3[X] + v3[Y]*v3[Y] + v3[Z]*v3[Z]);
  }
  return area;
}


/*************************************************************************
                               balance

   This routine evaluates the orthogonal subdivisions which would
  permit an approximate load balancing and then routes the polgyons
  their destination processors based on above subdivisions of model
  into rooms and mapping of these rooms on the DPU array.
 ************************************************************************/
balance()
{ plural int i, j, k, m; /* Misc counters */
  plural Poly* plural tempPoly; /* Polygon pointers */
plural Poly* plural newPoly;
plural int*  plural oldPolyBuf;
plural int*  plural newPolyBuf;
plural int newPolyTotal; /* Optimal no of polys per PE */
plural int newPEpoly;          /* Near optimal no of polys/PE*/
plural int roomCount[XROOM_DIM][YROOM_DIM][ZROOM_DIM];
plural Vec3 patchCenter; /* Center of a polygon */
plural byte xpatchRoom[XCELLS]; /* Describe location of cell*/
plural byte ypatchRoom[YCELLS];      /*    in the grid of virtual*/
plural byte zpatchRoom[ZCELLS];      /*    rooms.     */
plural int xpatchCount[XCELLS];    /* No of polys in each cell */
plural int ypatchCount[YCELLS];
plural int zpatchCount[ZCELLS];
plural byte curr_xproc; /* Temporaries */
plural byte curr_yproc;
plural byte xpatch,ypatch,zpatch;    /* Physical cell coords */
plural byte xroom, yroom, zroom; /* Virtual room grid coords*/
plural byte room_row;
int roomsum[XROOM_DIM][YROOM_DIM][ZROOM_DIM];
int room_sum; /* No of patches in room */
byte curr_room; /* Current room */
int polycount; /* Actual no of polygons */
int polyquota; /* Polys to be assigned to PE*/
int xpatchSum[XCELLS],ypatchSum[YCELLS],zpatchSum[ZCELLS];
int numpoly,maxpoly,minpoly;
int n,r,q; /* Misc singular counters */
int minRoomCount;             /*Min polygons in a room*/
int maxRoomCount;             /*Max polygons in a room*/

/* Initialize the variables */

for(n = 0; n < XROOM_DIM; n++)
  for(r = 0; r < YROOM_DIM; r++)
    for(q = 0; q < ZROOM_DIM; q++)
      roomCount[n][r][q] = 0;

for(n = 0; n < XCELLS; n++)
  xpatchCount[n] = xpatchSum[n] = 0;
for(n = 0; n < YCELLS; n++)
  ypatchCount[n] = ypatchSum[n] = 0;
for(n = 0; n < ZCELLS; n++)
  zpatchCount[n] = zpatchSum[n] = 0;


/* Find the number of polygons per cell */
```

```
for(i=0; i<oldPoly; i++)
{ tempPoly = old_poly_list + i;

  patchCenter[X] = patchCenter[Y] = patchCenter[Z] = 0.0;
  for(j = 0; j<tempPoly->numverts; j++)
  { patchCenter[X] += tempPoly->verts[j][X];
    patchCenter[Y] += tempPoly->verts[j][Y];
    patchCenter[Z] += tempPoly->verts[j][Z];
  }
  patchCenter[X] /= tempPoly->numverts;
  patchCenter[Y] /= tempPoly->numverts;
  patchCenter[Z] /= tempPoly->numverts;
  xpatch = (plural byte) ((patchCenter[X] - MinX)/XCELLLENGTH);
  ypatch = (plural byte) ((patchCenter[Y] - MinY)/YCELLLENGTH);
  zpatch = (plural byte) ((patchCenter[Z] - MinZ)/ZCELLLENGTH);

  if (xpatch >= XCELLS || ypatch >= YCELLS || zpatch >= ZCELLS) {
    p_printf("bad patch:  %d,%d,%d\n",xpatch,ypatch,zpatch);
    exit(-1);
  }

  xpatchCount[xpatch]++;
  ypatchCount[ypatch]++;
  zpatchCount[zpatch]++;
}

/* Verify that total number of polygons distributed along the x, y and z
   cells is equal */
numpoly = 0;
for(n = 0; n < XCELLS; n++)
{ xpatchSum[n] = reduceAdd32(xpatchCount[n]);
  numpoly += xpatchSum[n];
}
printf("numpoly = %d\n",numpoly);

numpoly = 0;
for(n = 0; n < YCELLS; n++)
{ ypatchSum[n] = reduceAdd32(ypatchCount[n]);
  numpoly += ypatchSum[n];
}
printf("numpoly = %d\n",numpoly);

numpoly = 0;
for(n = 0; n < ZCELLS; n++)
{ zpatchSum[n] = reduceAdd32(zpatchCount[n]);
```

```
    numpoly += zpatchSum[n];
  }
  printf("numpoly = %d\n",numpoly);


  /* Distribute the polygons so that they are approximately equal along the
     X - axis divisions.
  */
  room_sum = 0;
  curr_room = 0;
  polycount = numpoly;
  for(n = 0; n < XCELLS; n++)
  { polyquota = polycount/(XROOM_DIM-curr_room);

    if (_ABS(room_sum-polyquota) > _ABS(room_sum+xpatchSum[n]-polyquota)) {
      xpatchRoom[n] = curr_room;
    }
    else {
      xroomBound[curr_room] = n*XCELLLENGTH;
      printf("x curr_room = %d; bound = %g\n",curr_room,xroomBound[curr_room]);

      if (curr_room < XROOM_DIM-1)
curr_room++;
      xpatchRoom[n] = curr_room;
      polycount -= room_sum;
      room_sum = 0;
    }
    room_sum += xpatchSum[n];

    if (curr_room >= XROOM_DIM) {
      printf("bad curr_room = %d\n",curr_room);
      exit(-1);
    }
  }
  xroomBound[curr_room] = n*XCELLLENGTH;
  printf("x curr_room = %d; bound = %g\n",curr_room,xroomBound[curr_room]);

  /* Distribute the polygons so that they are approximately equal along the
     Y - axis divisions.
  */
  room_sum = 0;
  curr_room = 0;
  polycount = numpoly;
  for(n = 0; n < YCELLS; n++)
  { polyquota = polycount/(YROOM_DIM-curr_room);
```

```
      if (_ABS(room_sum-polyquota) > _ABS(room_sum+ypatchSum[n]-polyquota)) {
        ypatchRoom[n] = curr_room;
      }
      else {
        yroomBound[curr_room] = n*YCELLLENGTH;
        printf("y curr_room = %d; bound = %g\n",curr_room,yroomBound[curr_room]);

        if (curr_room < YROOM_DIM-1)
curr_room++;
        ypatchRoom[n] = curr_room;
        polycount -= room_sum;
        room_sum = 0;
      }
      room_sum += ypatchSum[n];

      if (curr_room >= YROOM_DIM) {
        printf("bad ycurr_room = %d\n",curr_room);
        exit(-1);
      }
    }
    yroomBound[curr_room] = n*YCELLLENGTH;
    printf("y curr_room = %d; bound = %g\n",curr_room,yroomBound[curr_room]);

    /* Distribute the polygons so that they are approximately equal along the
       Z - axis divisions.
    */
    room_sum = 0;
    curr_room = 0;
    polycount = numpoly;
    for(n = 0; n < ZCELLS; n++)
    { polyquota = polycount/(ZROOM_DIM-curr_room);

      if (_ABS(room_sum-polyquota) > _ABS(room_sum+zpatchSum[n]-polyquota)) {
        zpatchRoom[n] = curr_room;
      }
      else {
        zroomBound[curr_room] = n*ZCELLLENGTH;
        printf("z curr_room = %d; bound = %g\n",curr_room,zroomBound[curr_room]);

        if (curr_room < ZROOM_DIM-1)
curr_room++;
        zpatchRoom[n] = curr_room;
        polycount -= room_sum;
        room_sum = 0;
    }
    room_sum += zpatchSum[n];

    if (curr_room >= ZROOM_DIM) {
      printf("bad curr_room = %d\n",curr_room);
      exit(-1);
    }
  }
  zroomBound[curr_room] = n*ZCELLLENGTH;
  printf("z curr_room = %d; bound = %g\n",curr_room,zroomBound[curr_room]);

  /* Find the number of polygons within each room */
  for(i=0; i<oldPoly; i++)
  { tempPoly = old_poly_list + i;

    patchCenter[X] = patchCenter[Y] = patchCenter[Z] = 0.0;
    for(j = 0; j<tempPoly->numverts; j++)
    { patchCenter[X] += tempPoly->verts[j][X];
      patchCenter[Y] += tempPoly->verts[j][Y];
      patchCenter[Z] += tempPoly->verts[j][Z];
    }
    patchCenter[X] /= tempPoly->numverts;
    patchCenter[Y] /= tempPoly->numverts;
    patchCenter[Z] /= tempPoly->numverts;
    xpatch = (plural byte) ((patchCenter[X] - MinX)/XCELLLENGTH);
    ypatch = (plural byte) ((patchCenter[Y] - MinY)/YCELLLENGTH);
    zpatch = (plural byte) ((patchCenter[Z] - MinZ)/ZCELLLENGTH);

    if (xpatch >= XCELLS || ypatch >= YCELLS || zpatch >= ZCELLS) {
      p_printf("bad patch:  %d,%d,%d\n",xpatch,ypatch,zpatch);
      exit(-1);
    }

    if (xpatchRoom[xpatch] >= XROOM_DIM ||
      ypatchRoom[ypatch] >= YROOM_DIM ||
      zpatchRoom[zpatch] >= ZROOM_DIM) {
      p_printf("bad patch room:  %d,%d,%d\n",
xpatchRoom[xpatch],ypatchRoom[ypatch],zpatchRoom[zpatch]);
      exit(-1);
    }

    roomCount[xpatchRoom[xpatch]][ypatchRoom[ypatch]][zpatchRoom[zpatch]]++;
  }

  /* Print out the minimum and maximum number of polygons per room */
```

```
minRoomCount = 10000;
maxRoomCount = -HUGE;
numpoly = 0;
for(n = 0; n < XROOM_DIM; n++) {
  for(r = 0; r < YROOM_DIM; r++) {
    for(q = 0; q < ZROOM_DIM; q++) {
      roomsum[n][r][q] = reduceAdd32(roomCount[n][r][q]);
      minRoomCount = MIN(minRoomCount, roomsum[n][r][q]);
      maxRoomCount = MAX(maxRoomCount, roomsum[n][r][q]);
      numpoly += roomsum[n][r][q];
      printf("room %d,%d,%d count = %d\n",n,r,q,roomsum[n][r][q]);
    }
  }
}
printf("numpoly = %d minRoomCount = %d maxRoomCount = %d\n",
numpoly,minRoomCount,maxRoomCount);
fflush(stdout);

/* Transfer the polgyons to the appropriate PE taking into account the way
   rooms are mapped onto the MasPar grid.
*/

all newPolyCount = 0;

for(i=0; i<oldPoly; i++)
{ tempPoly = old_poly_list + i;

  /* Find the virtual room to which this polygon belongs */
  patchCenter[X] = patchCenter[Y] = patchCenter[Z] = 0.0;
  for(j = 0; j<tempPoly->numverts; j++)
  { patchCenter[X] += tempPoly->verts[j][X];
    patchCenter[Y] += tempPoly->verts[j][Y];
    patchCenter[Z] += tempPoly->verts[j][Z];
  }
  patchCenter[X] /= tempPoly->numverts;
  patchCenter[Y] /= tempPoly->numverts;
  patchCenter[Z] /= tempPoly->numverts;
  xpatch = (plural byte) ((patchCenter[X] - MinX)/XCELLLENGTH);
  ypatch = (plural byte) ((patchCenter[Y] - MinY)/YCELLLENGTH);
  zpatch = (plural byte) ((patchCenter[Z] - MinZ)/ZCELLLENGTH);

  if (xpatch >= XCELLS || ypatch >= YCELLS || zpatch >= ZCELLS) {
    p_printf("bad patch:  %d,%d,%d\n",xpatch,ypatch,zpatch);
    exit(-1);
  }
```

```
    xroom = xpatchRoom[xpatch];
    yroom = ypatchRoom[ypatch];
    zroom = zpatchRoom[zpatch];

    if (xroom >= XROOM_DIM || yroom >= YROOM_DIM || zroom >= ZROOM_DIM) {
      p_printf("bad patch room:  %d,%d,%d\n", xroom,yroom,zroom);
      exit(-1);
    }

    /* Find the processors to which this room is assigned and get the processor
       to which to send the current polygon
    */
    room_row = xroom*YROOM_DIM + yroom;

    if (room_row > nxproc) {
      p_printf("bad room_row %d\n",room_row);
      exit(-1);
    }

    curr_yproc = room_row;
    curr_xproc = ixproc;

    if (curr_yproc < 0 || curr_yproc >= nyproc) {
      p_printf("bad yproc:  %d\n",curr_yproc);
      exit(-1);
    }


    all  m = 0;

    /* Transfer the polygon to the destination processor using router */
    while ((plural) 1) {
      if (connected(curr_yproc*nxproc + curr_xproc)) {

        all
        { newPoly = new_poly_list + newPolyCount;
  newPolyBuf = (plural int* plural)newPoly;
}

oldPolyBuf = (plural int* plural)tempPoly;

        m = router[curr_yproc*nxproc + curr_xproc].newPolyCount;
        m++;
all if (m >= maxPEpoly) {
  p_printf("bad m = %d\n",m);
  exit(-1);
}

          newPEpoly = roomsum[xroom][yroom][zroom]/nxproc+1;
          newPolyTotal =
            (curr_xproc < roomsum[xroom][yroom][zroom]-(newPEpoly-1)*nxproc ?
              newPEpoly : newPEpoly-1);

          if (m > newPolyTotal) {
            curr_xproc++;
            curr_xproc %= nxproc;
            continue;
          }
          else {
            for (n = 0; n < sizeof(Poly)/sizeof(int); n++)
              router[curr_yproc*nxproc+curr_xproc].newPolyBuf[n] = oldPolyBuf[n];

            router[curr_yproc*nxproc + curr_xproc].newPolyCount = m;
            break;
          }
        }
      }

      /* Print the number of minimum and maximum number of polygons to a processo
         after the above routing and grouping of polygons to rooms.
      */
      all
      { numpoly = reduceAdd32(newPolyCount);
        maxpoly = reduceMax32(newPolyCount);
        minpoly = reduceMin32(newPolyCount);
        printf("new polys = %d; max = %d, min = %d\n",numpoly,maxpoly,minpoly);
      }
    }
}

/*********************************************************************
                         clip_to_ortho_plane

   This routine clips the polygon according to one of the orthogonal
  planes specified (X, Y, or Z) and returns the number of clipped
  vertices.

*********************************************************************/
```

```
plural int clip_to_ortho_plane(n,src,si,sv,bad_side,dest)
plural int n; /* Number of vertices*/
plural float src[2*MAXVERT][3]; /* Input vertices */
plural int si;   /* Orthogonal Plane */
plural float sv; /* Equation of ortho plane*/
plural int  bad_side;
plural float dest[2*MAXVERT][3]; /* Clipped verts */
{
   plural int clipped_n = 0;
   plural int i,k;
   plural int side,next_side;
   plural float t;

   /* this is really part of the loop initialization */
   if      ((src[0][si] -  sv) < -L1_NORM_MIN) side = -1;
   else if ((src[0][si] -  sv) >  L1_NORM_MIN) side =  1;
   else side =  0;

   for (i = 0; i < n; i++){
      /* add the point if side is the correct side */
      if (side != bad_side){
 if (clipped_n >= 2*MAXVERT)
    die("clip_to_ortho_plane","too many clipped verts",1);
 dest[clipped_n][X] = src[i][X];
 dest[clipped_n][Y] = src[i][Y];
 dest[clipped_n++][Z] = src[i][Z];
      }

      /* now check to see if the edge has an intersection point */
      k = ((i+1) == n ? 0 : i+1);
      if      ((src[k][si] -  sv) < -L1_NORM_MIN) next_side = -1;
      else if ((src[k][si] -  sv) >  L1_NORM_MIN) next_side =  1;
      else    next_side =  0;

      if (side != 0 && next_side != 0 && side != next_side){
 /* solve sv = t*(v2 - v1) + v1 */
 /* and use the solution to compute the intersection */
 t = (sv - src[i][si]) / (src[k][si] - src[i][si]);
 if (clipped_n == 2*MAXVERT)
    die("clip_to_ortho_plane","too many clipped verts",1);
 dest[clipped_n][X]   = src[i][X] + t*(src[k][X] - src[i][X]);
 dest[clipped_n][Y]   = src[i][Y] + t*(src[k][Y] - src[i][Y]);
 dest[clipped_n++][Z] = src[i][Z] + t*(src[k][Z] - src[i][Z]);
      }
      side = next_side;
   }
   return clipped_n;
}


/*********************************************************************
                          clip_face_to_box

   This routine clips the polygon to the six extents specified in its
  argument list and returns the clipped polygon vertices

*********************************************************************/
plural int clip_face_to_box(xmin,xmax,ymin,ymax,zmin,zmax,f,dest,orientation,ex
plural float xmin,xmax,ymin,ymax,zmin,zmax;/* Clipping extents*/
plural Poly* plural f;       /* Input Poly */
plural float dest[2*MAXVERT][3];        /* Output verts */
plural int orientation;                /* Polygon orientation*/
plural float ex[6];        /* Input Poly extents*/
{
   plural int n;
   plural float b[2*MAXVERT][3];
   plural float v1[3],v2[3],norm[3];
   plural int i,j,k;

   if ((n = f->numverts) > MAXVERT)
      die("clip_face_to_box","too many verts in poly",1);

   for (i = 0; i < n; i++){
      dest[i][X] = f->verts[i][X];
      dest[i][Y] = f->verts[i][Y];
      dest[i][Z] = f->verts[i][Z];
   }

   switch (orientation){
      case Z:  /* have to clip to xmin,xmax, ymin,ymax */
  if (ex[MINZ] > zmax || ex[MINZ] < zmin) return 0;
  if (!(n = clip_to_ortho_plane(n,dest,(plural int)X,
    xmin,(plural int)(-1),b)))
     return (plural int)0;
  if (!(n = clip_to_ortho_plane(n,b,(plural int)X,
     xmax,(plural int)1,dest)))
     return (plural int)0;
  if (!(n = clip_to_ortho_plane(n,dest,(plural int)Y,
     ymin,(plural int)(-1),b)))
     return (plural int)0;
```

```
if (!(n = clip_to_ortho_plane(n,b,(plural int)Y,
   ymax,(plural int)1,dest)))
   return (plural int)0;
break;
     case Y:  /* have to clip to xmin,xmax, zmin, zmax */
if (ex[MINY] > ymax || ex[MINY] < ymin) return 0;
if (!(n = clip_to_ortho_plane(n,dest,(plural int)X,
  xmin,(plural int)(-1),b)))
   return (plural int)0;
if (!(n = clip_to_ortho_plane(n,b,(plural int)X,
  xmax,(plural int)1,dest)))
   return (plural int)0;
if (!(n = clip_to_ortho_plane(n,dest,(plural int)Z,
  zmin,(plural int)(-1),b)))
   return (plural int)0;
if (!(n = clip_to_ortho_plane(n,b,(plural int)Z,
  zmax,(plural int)1,dest)))
   return (plural int)0;
break;
     case X:  /* have to clip to ymin,ymax, zmin, zmax */
if (ex[MINX] > xmax || ex[MINX] < xmin) return 0;
if (!(n = clip_to_ortho_plane(n,dest,(plural int)Y,
  ymin,(plural int)(-1),b)))
   return (plural int)0;
if (!(n = clip_to_ortho_plane(n,b,(plural int)Y,
  ymax,(plural int)1,dest)))
   return (plural int)0;
if (!(n = clip_to_ortho_plane(n,dest,(plural int)Z,
  zmin,(plural int)(-1),b)))
   return (plural int)0;
if (!(n = clip_to_ortho_plane(n,b,(plural int)Z,
  zmax,(plural int)1,dest)))
   return (plural int)0;
break;
     case SKEWX:
     case SKEWY:
     case SKEWZ:
if (!(n = clip_to_ortho_plane(n,dest,(plural int)X,
  xmin,(plural int)(-1),b)))
   return (plural int)0;
if (!(n = clip_to_ortho_plane(n,b,(plural int)X,
  xmax,(plural int)1,dest)))
   return (plural int)0;
if (!(n = clip_to_ortho_plane(n,dest,(plural int)Y,
  ymin,(plural int)(-1),b)))
```

```
     return (plural int)0;
  if (!(n = clip_to_ortho_plane(n,b,(plural int)Y,
    ymax,(plural int)1,dest)))
     return (plural int)0;
  if (!(n = clip_to_ortho_plane(n,dest,(plural int)Z,
    zmin,(plural int)(-1),b)))
     return (plural int)0;
  if (!(n = clip_to_ortho_plane(n,b,(plural int)Z,
    zmax,(plural int)1,dest)))
     return (plural int)0;
  break;
    }

    /* need to check that the clipped poly is not degenerate */
    if (n > 2){
       for (i = 0; i < n; i++){
  j = (i+1) % n;
  k = (j+1) % n;

  v1[X] = dest[j][X] - dest[i][X];
  v1[Y] = dest[j][Y] - dest[i][Y];
  v1[Z] = dest[j][Z] - dest[i][Z];

  v2[X] = dest[k][X] - dest[j][X];
  v2[Y] = dest[k][Y] - dest[j][Y];
  v2[Z] = dest[k][Z] - dest[j][Z];

       norm[X] =  v1[Y]*v2[Z] - v1[Z]*v2[Y];
       norm[Y] = -v1[X]*v2[Z] + v1[Z]*v2[X];
       norm[Z] =  v1[X]*v2[Y] - v1[Y]*v2[X];

  if (fp_fabs(norm[X])+fp_fabs(norm[Y])+fp_fabs(norm[Z])< L1_NORM_MIN)
      return (plural int)0;
       }
       return n;
    }
    else return (plural int)0;
}


/***********************************************************************
                           refine_balance

  This function improves on the load balancing achieved by the
  routine balance(). It does this by patchifying at a finer level the
  polygons on each processor. After this, the newly formed patches
  are further rebalanced across all the processors representing a
  room (a DPU row right now)

************************************************************************/
refine_balance()
{ float rowGrid[NYPROC];
  int row_n;
  int rowPolyCount[NYPROC];
  plural float   gridSide[3]; /* Finer patchification grid */
  plural int i, j, jj, k, l, m; /* Misc counters */
  plural Poly* plural   tempPoly;
  int n, r, q; /* Misc counters */
  plural float ex[3*2]; /* Extents of the polygon */
  plural float clipbox[3*2]; /* Clipping Box for patches*/
  plural float normal[3]; /* Normals of the polygon */
  plural int orientation;/* Axis to which the poly is mainly normal*/
  plural int x0, x1, x2; /* Temporaries */
  plural int x1i, x2i, x1dim, x2dim;
  plural float wb[2*MAXVERT][3]; /* Clipped vertices */
  plural float new_verts[2*MAXVERT][3];
  plural Poly polyBuf; /* Temporary buffer for a poly*/
  plural char polyBufvalid = 0;
  int         maxRowpoly; /* Max no of rows per poly */

  /* Compute the max number of polys in a row */
  r = 0;
  for(n = 0; n < nyproc; n++)
  { if (iyproc == n)
      r = MAX(r,reduceAdd32(newPolyCount));
  }

  /* Compute the new patchification grid for each row */
  for(n = 0; n < nyproc; n++)
  { if (iyproc == n)
    { rowGrid[n] = f_sqrt(f_floor(r*1.0/(MAX(1,reduceAdd32(newPolyCount)))));
      gridSide[X] = XCELLLENGTH/rowGrid[n];
      gridSide[Y] = YCELLLENGTH/rowGrid[n];
      gridSide[Z] = ZCELLLENGTH/rowGrid[n];
      /* Clamp at 24 * 24 inch patches */
      gridSide[X] = MAX(24.0,gridSide[X]);
      gridSide[Y] = MAX(24.0,gridSide[Y]);
      gridSide[Z] = MAX(24.0,gridSide[Z]);
    }
  }
```

```
oldPoly = 0;
for(i = 0; i < newPolyCount; i++)
{ tempPoly = new_poly_list + i;
  ex[MINEX(X)] = ex[MINEX(Y)] = ex[MINEX(Z)] = HUGE;
  ex[MAXEX(X)] = ex[MAXEX(Y)] = ex[MAXEX(Z)] = -HUGE;

  /* Find the polgyon extents */
  for(j = 0; j<tempPoly->numverts; j++)
  { ex[MINEX(X)] = MIN(ex[MINEX(X)],tempPoly->verts[j][X]);
    ex[MINEX(Y)] = MIN(ex[MINEX(Y)],tempPoly->verts[j][Y]);
    ex[MINEX(Z)] = MIN(ex[MINEX(Z)],tempPoly->verts[j][Z]);
    ex[MAXEX(X)] = MAX(ex[MAXEX(X)],tempPoly->verts[j][X]);
    ex[MAXEX(Y)] = MAX(ex[MAXEX(Y)],tempPoly->verts[j][Y]);
    ex[MAXEX(Z)] = MAX(ex[MAXEX(Z)],tempPoly->verts[j][Z]);
  }

  /* Find the orientation - ie the principal axis which is most nearly
     normal to the polygon
  */
  normal[X] = fp_fabs(tempPoly->eq[A]);
  normal[Y] = fp_fabs(tempPoly->eq[B]);
  normal[Z] = fp_fabs(tempPoly->eq[C]);

  if (normal[X] > normal[Y])
  { if (normal[X] > normal[Z]) orientation = SKEWX;
    else orientation = SKEWZ;
  }
  else
  { if (normal[Y] > normal[Z]) orientation = SKEWY;
   else orientation = SKEWZ;
  }
  if ((normal[Y] == 0)&&(normal[Z] == 0)) orientation = X;
  if ((normal[X] == 0)&&(normal[Z] == 0)) orientation = Y;
  if ((normal[Y] == 0)&&(normal[X] == 0)) orientation = Z;

  switch (orientation)
  { case X: case SKEWX: x1 = Y; x2 = Z; x0 = X; break;
    case Y: case SKEWY: x1 = Z; x2 = X; x0 = Y; break;
    case Z: case SKEWZ: x1 = X; x2 = Y; x0 = Z; break;
  }

  /* Determine the clipping boxes required for patchifying the polygon */
  x1i =  (plural int)(fp_floor(ex[MINEX(x1)]/gridSide[x1]));
  x2i =  (plural int)(fp_floor(ex[MINEX(x2)]/gridSide[x2]));
  x1dim = (plural int)(fp_ceil(ex[MAXEX(x1)]/gridSide[x1])) - x1i;
  x2dim = (plural int)(fp_ceil(ex[MAXEX(x2)]/gridSide[x2])) - x2i;

  clipbox[MINEX(x0)] = ex[MINEX(x0)];
  clipbox[MAXEX(x0)] = ex[MAXEX(x0)];
  clipbox[MINEX(x1)]= x1i*gridSide[x1];
  clipbox[MAXEX(x1)]= clipbox[MINEX(x1)] + gridSide[x1];

  /* Clip and write out the new finer patches from the polgyons */
  for (l = 0; l < x1dim; l++)
  { clipbox[MINEX(x2)] = x2i*gridSide[x2];
    clipbox[MAXEX(x2)] = clipbox[MINEX(x2)] + gridSide[x2];
    for (j = 0; j < x2dim; j++)
    { if ((m = clip_face_to_box(clipbox[MINX],clipbox[MAXX],
clipbox[MINY],clipbox[MAXY],
clipbox[MINZ],clipbox[MAXZ],
tempPoly,wb,orientation,ex)) > 2)
      { if (m <= 4)
          write_patch(tempPoly,wb,m);
else
        { write_patch(tempPoly,wb,(plural int)4);
          if (m%2 == 1)  k = m - 1;
          else  k = m;
          for (jj = 1 ; jj <= (k-4)/2; jj++)
          { for(r=0;r<3;r++)
            { new_verts[0][r] = wb[0][r];
              new_verts[1][r] = wb[(jj*2)+1][r];
              new_verts[2][r] = wb[(jj*2)+2][r];
              new_verts[3][r] = wb[(jj*2)+3][r];
            }
            write_patch(tempPoly,new_verts,(plural int)4);
          }
          if (m%2 == 1)
          { for(r=0;r<3;r++)
            { new_verts[0][r] = wb[0][r];
              new_verts[1][r] = wb[(jj*2)+1][r];
              new_verts[2][r] = wb[(jj*2)+2][r];
            }
            write_patch(tempPoly,new_verts,(plural int)3);
          }
}
      }
      clipbox[MINEX(x2)] = clipbox[MAXEX(x2)];
      clipbox[MAXEX(x2)] += gridSide[x2];
    }
    clipbox[MINEX(x1)] = clipbox[MAXEX(x1)];
    clipbox[MAXEX(x1)] += gridSide[x1];
  }
}


if (oldPoly > maxPEpoly)
{ printf("Excessive  Poly Counts after finer patchification\n");
  p_printf("[%d,%d]=%d ",ixproc,iyproc,oldPoly);
}

/* Now try and balance the number of patches across each row */

for(n = 0; n<nyproc; n++)
{  if (iyproc == n) rowPolyCount[n] = reduceAdd32(oldPoly);
   rowPolyCount[n] /= nxproc; /* Find the desired no of patches per proc */
}

/* Start distributing patches to achieve the above no of patches per proc */
/* within rowPolyCount[n]+1 is ok */
while (oldPoly > rowPolyCount[iyproc] + 1)
{ tempPoly = old_poly_list + (--oldPoly);
  for(n = 0; n<MAXVERT; n++)
    for(r=0; r<3; r++)
      polyBuf.verts[n][r] = tempPoly->verts[n][r];
  for(r = 0; r<4; r++)
  { polyBuf.eq[r] = tempPoly->eq[r];
    for(n = 0; n<MAXVERT; n++)
      polyBuf.colors[n][r] = tempPoly->colors[n][r];
  }
  polyBuf.numverts = tempPoly->numverts;
  polyBuf.area = tempPoly->area;
  polyBufvalid = 1;

  all for(q=0; q<nxproc ; q++)
  { if (polyBufvalid)
    { polyBufvalid = 0;
      if (oldPoly <= rowPolyCount[iyproc])
      { tempPoly = old_poly_list + oldPoly++;
for(n = 0; n<MAXVERT; n++)
          for(r=0; r<3; r++)
            tempPoly->verts[n][r] = polyBuf.verts[n][r];
     for(r = 0; r<4; r++)
     { tempPoly->eq[r] = polyBuf.eq[r];
         for(n = 0; n<MAXVERT; n++)
```

```
                tempPoly->colors[n][r] = polyBuf.colors[n][r];
          }
          tempPoly->numverts = polyBuf.numverts;
          tempPoly->area = polyBuf.area;
           }
           else
           { ss_xsend(0,1,&polyBuf,&polyBuf,sizeof(Poly));
             xnetE[1].polyBufvalid = (plural char)1;
           }
        }
      }
  }

  /* Find the total and the maximum number of patches in a row */
  for (n=0; n<nyproc; n++)
    if (iyproc == n) {
      row_n = reduceAdd32(oldPoly);
      printf("Total patches on row %d = %d\n",n,row_n);
      maxRowpoly = MAX(maxRowpoly,row_n);
    }

  printf("max row patches = %d\n",maxRowpoly);

  maxPEpoly = reduceMax32(oldPoly);
  printf("max PE patches = %d\n",maxPEpoly);

}


/**********************************************************************
                          write_patch

  This patch is written onto the current processor if it has
  enough memory space available. Else it is written to the nearest
  processor that has space for it.
**********************************************************************/
write_patch(poly, verts, numverts)
plural Poly* plural poly; /* Parent polygon */
plural float verts[2*MAXVERT][3]; /* Patch vertices */
plural int numverts; /* No of patch vertices*/
{ plural Poly* plural   tempPoly;
  plural Poly polyBuf;
  plural char polyBufvalid = 0;
  plural int i,j;
  int n,r,q;
  plural float temp_area;

  tempPoly = old_poly_list + oldPoly;
  temp_area = area(verts, numverts);

  if (oldPoly < maxPEpoly)
/* Write the patch on this proc if possible*/
  { for(n = 0; n<MAXVERT; n++)
      for(r=0; r<3; r++)
        tempPoly->verts[n][r] = verts[n][r];
    for(r = 0; r<4; r++)
    { tempPoly->eq[r] = poly->eq[r];
      for(n = 0; n<MAXVERT; n++)
        tempPoly->colors[n][r] = poly->colors[n][r];
    }
    tempPoly->numverts = numverts;
    tempPoly->area = temp_area;
    if(temp_area > 1.0) oldPoly++;
  }
  else  /* Not enough memory on this processor */
  if (temp_area > 1.0)
  { for(n = 0; n<MAXVERT; n++)
      for(r=0; r<3; r++)
        polyBuf.verts[n][r] = verts[n][r];
    for(r = 0; r<4; r++)
    { polyBuf.eq[r] = poly->eq[r];
      for(n = 0; n<MAXVERT; n++)
        polyBuf.colors[n][r] = poly->colors[n][r];
    }
    polyBuf.numverts = numverts;
    polyBuf.area = temp_area;

    /* Spread to the nearest proc on the right that is still not fully filled*/
    j = ixproc;
    i = j + 1;
    while ((i != j) & (polyBufvalid))
    { if (polyBufvalid)
      { polyBufvalid = 0;
        if(oldPoly < maxPEpoly)
        { tempPoly = old_poly_list + oldPoly++;
  for(n = 0; n<MAXVERT; n++)
            for(r=0; r<3; r++)
              tempPoly->verts[n][r] = polyBuf.verts[n][r];
       for(r = 0; r<4; r++)
       { tempPoly->eq[r] = polyBuf.eq[r];
```

```
            for(n = 0; n<MAXVERT; n++)
               tempPoly->colors[n][r] = polyBuf.colors[n][r];
         }
         tempPoly->numverts = polyBuf.numverts;
         tempPoly->area = polyBuf.area;
          }
          else
          { ss_xsend(0,1,&polyBuf,&polyBuf,sizeof(Poly));
            xnetE[1].polyBufvalid = (plural char)1;
          }
       }
       i = (i+1)%nxproc;
     }
   }
}


/************************************************************************
                            print_poly

  Print out the polygon tempPoly at processor(x,y) on the DPU.
************************************************************************/
print_poly(x,y,tempPoly)
plural int  x;
plural int  y;
plural Poly* plural tempPoly;
{ if ((ixproc == x)&&(iyproc == y))
    p_printf("Poly at [%d,%d]: \n X0 %f Y0 %f Z0 %f\n X1 %f Y1 %f Z1 %f\n X2 %f

}
```