# A Parallel-Pipelined Multiprocessor System for the Radiosity Method

L.S. Shen, and E.F. Deprettere

ABSTRACT Ray-tracing and radiosity algorithms can produce very realistic images, but they require a lot of computations which make them impractical for scenes of high complexity. Several attempts have been made to speed up computations through parallel processing. To get orders of magnitude speedup, massive parallelism involving multiple streams will be necessary. In this paper, a parallel-pipelined multiprocessor system is described, which is made of clusters of specialized computing modules, each constructed of an Intersection Computation Unit (ICU) and a number of Cell Traversal Units (CTUs). Both ICU and CTU are of type pipeline and with data-driven execution. A pseudo-dynamic scheduling is used to reconfigure the system at run time so that the workloads distributed over clusters can be more or less balanced. Furthermore, a hierarchical memory structure is proposed to reduce the average loading time of patches. Performance evaluation has been done and 15% more speedup can be obtained as observed by queueing network simulation. A complete system level simulation is under way by using BONeS which is a block oriented network simulator.

## 1.1 Introduction

Recently, several techniques have been developed for rendering high quality images on a video screen. The most noticeable examples are ray tracing and radiosity. Despite the fact of improved realism, they proved to be extremely time-consuming which makes them impractical for rendering complex scenes. An obvious answer to this dilemma lies in parallel processing.

Many parallel architectures have been proposed and developed for computer graphics. These can be classified into three types: (1) Vector Processor, (2) Array Processor, and (3) Multiprocessor. Computation for image creation consists primarily of 3D vector and matrix operations, in which the dimension is ordinarily very low. Hence, computer systems of type 1 like the Cray 1 are not fit for this purpose. Usually, there are two reasons for pursuing computer systems of type 2: (1) it is more cost-effective in any given technology because of the saving in the instruction and decode hardware, and (2) it is conceptually easier to program and debug. However, it is very difficult to keep all processors doing useful work all the time, and so processor utilization might be very low. One noted example is Pixel-planes 4 in which no screen partitioning is built. When rendering polygons, Pixel-planes 4 disables all the pixels outside a polygon, and hence all these pixels' processors remaining idle until the next polygon arrives. Consider a complex scene that is composed mostly of very small polygons, and in which many polygons cover only a handful of pixels. The result would be a rather inefficient use of processors. The same



delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

problem can arise when one attempts to use the Connection Machine for computer graphics. Multiprocessor architectures can range from multiprocessors sharing memory over a common bus (Sequent Balance), through multiprocessors sharing memory on a multistage network (BBN Butterfly), to multiprocessor message- passing systems (Ncube). With this type of computer systems, care must be taken when mapping algorithms from computer graphics onto multiple processors. Inadequate mapping strategies are detrimental to the system performance.

In this paper, we concentrate on multiprocessor architectures since ray-tracing algorithms cannot be efficiently mapped onto computer systems of types 1 and 2 as described above. Published architectures can be classified into three classes: processing without dataflow [5, 6], processing with ray dataflow [1, 2], and processing with object dataflow [3, 7]. The drawback of the first class is that either the entire scene database should be replicated for each processor (limiting the scene complexity for rendering) or the scene database must be in a shared memory (limiting the size of possible configurations). As for the second class, the system performance will be degraded due to the communication overhead of passing ray messages through a number of processors. This can be resolved by assigning rays to processors as the third class does, but it requires an efficient way to access the objects from the database. In [9, 10], we proposed a new space partitioning called the shelling technique that can reduce algorithmic complexity considerably. In this paper, we discuss a pseudo-dynamic scheduling that can map the shelling technique onto a parallel-pipelined architecture classified to the third class. In order to access the objects efficiently, we propose a memory structure which is a hierarchy of main memory, local memory, and cache.

The outline of the paper is as follows. After establishing the background of the shelling technique in Section 1.2, we explain why a parallel-pipelined architecture is chosen and how to map the radiosity method onto it. After that, we give an overview of the system and discuss some crucial issues regarding memory structure, synchronization mechanism, and main primitive functions. Finally, we show the results of some practical scenes and give a conclusion.

## 1.2 Background

The shelling technique is a space partitioning that can reduce the communication overhead of loading patches. An example of a shell-like structure built by the shelling technique is shown in Figure 1.1. The half-space seen by the source patch is first partitioned into shells. Then, a shell is partitioned into a number of subshells based on the ADRC[1] of the local neighborhood of the object space. Once the structure has been constructed, each relevant patch within the subshell currently considered can compute intersection points with a bundle of rays which is determined by the spherical bounding box[2] of this patch. With this arrangement, a relevant patch is necessary to be loaded only once.

On the other hand, the shelling technique is a mapping that can map the partitioned space onto a parallel-pipelined architecture. This is done by a two-step procedure: (1) the object space is uniformly partitioned into subspaces by equally distributed $\Delta\theta$ and $\Delta\phi$ angles, and a low density ray casting is performed recording the number of cell traversals

[1]ADRC stands for Average Degree of Ray Coherence that represents the average number of rays shot from a sample point $O$ over which a hemisphere is placed that hit a patch.

[2]The spherical bounding box of a subshell (or a patch) is a superset of the convex hull on spherical geometry of the subshell (or the patch).

and intersection computations for each subspace, and (2) based on the information of the low density ray casting, the object space is reorganized into sections/sectors by using a Binary Space Partitioning (BSP) subdivision such that the estimated workloads among them are evenly distributed. While mapping this partitioning onto a parallel-pipelined architecture, a section and its corresponding sectors can be mapped onto a cluster which consists of an ICU and a number of CTUs. Both ICU and CTU are of type pipeline.

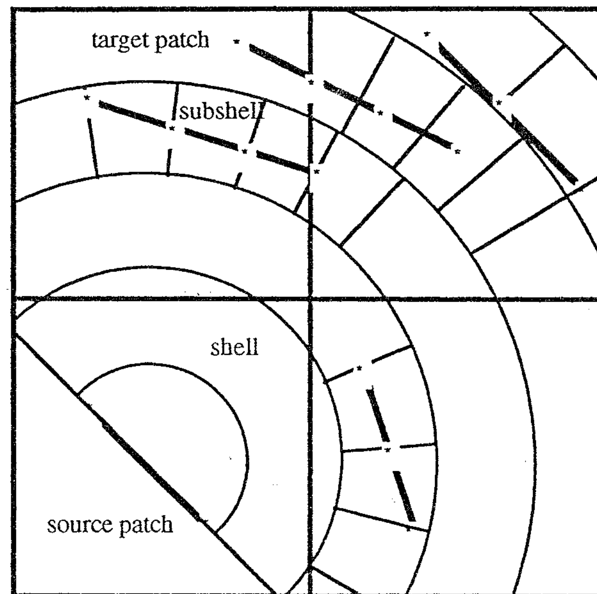


**Fig. 1.1.** A shell-like structure built by the shelling technique.

## 1.3 A Parallel-Pipelined Architecture

At a superficial level, there exists a high degree of inherent parallelism in the ray tracing algorithms applied to compute form-factors for the radiosity method. In a naive ray-tracing algorithm, it is relatively easy to broadcast the objects of a scene and the accompanying ray-tracing program to multiple processors, then have each processor process a ray or group of rays assigned to it. For optimum load-balancing, a given processor's rays should be interleaved with the others. Even with a trivial mapping strategy, speedup can be generally high. However, all processors turn out to be very inefficient because they are doing wasteful work most of the time. Conventional space partition techniques can be used to reduce wasteful work by searching for relevant objects, thereby accelerating ray tracing. However, the searching procedure makes the scheduling problem complicated. On the one hand, the amount of computation and communication, as well as execution dependencies cannot be known a priori. On the other hand, relevant objects are routed to certain processors instead of broadcasting them to all processors, and so heavy communication overhead is involved.

In a multiprocessor ray-tracing algorithm, it is indispensable to moving objects dynamically. Our strategy is to move objects only when necessary and then keep them stationary as much time as possible. This led to the concept of the shelling technique. In the shelling technique, any relevant patch found by a ray is routed to a processor and can continue testing against a bundle of rays. It is difficult to exploit this run-time changing parallelism by assigning a separate processor to each ray stream. Pipeline processing

seems appropriate for this purpose because it can handle changes in the number of ray streams in a natural way. Due to the property of object coherence: the local neighborhoods of space tend to be occupied by the same object, multiple processors can be used for processing distinctly different parts of the space to achieve higher system performance. In conclusion, we argue that a parallel-pipelined architecture is well suited to the radiosity method.

### 1.3.1 Pseudo-Dynamic Scheduling

Scheduling implies determining when (scheduling in time) and where (assignment to a processor) each process is executed. For the purposes of this paper, we may distinguish between static and dynamic scheduling. Within the realm of dynamic scheduling, we further distinguish between fully dynamic and quasi-dynamic scheduling. In the case of static scheduling, information regarding the processes in the system is assumed to be available a priori. Hence, the compiler can determine when and where each process is to be executed before program execution. Static scheduling is attractive since it needs only be performed once. In our case, very little a priori knowledge about the processes in the system is available. So the scheduling can only be performed dynamically as the parallel program executes. For fully dynamic scheduling, a process ready for processing is assigned to an idle processor at run time. It is claimed to be most effective in utilizing resources and to fully exploit the concurrency of an algorithm, regardless of the amount of dependency. However, it requires much hardware/software run-time overhead. Furthermore, it is usually not practical to make globally optimal scheduling decisions at run time. In view of the inapplicability of static scheduling and the high cost of fully dynamic scheduling, we should take a closer look at our problem to determine a suitable scheduling strategy.

The radiosity method generally progresses through a sequence of refinement steps that allow rapid generation of good images. Often, the execution time associated with each step is relatively large, and the workload and communication requirements do not change a lot during a step. It might be advantageous to redistribute data objects and associated workloads by a static assignment at the beginning of each step if the overhead is low. Our approach is to use a low density ray casting to give an estimation of workloads. Based on this estimation, we can partition the space into sections/sectors assuming more or less balanced workloads. They are then assigned to processors and kept unchanged until the next step. In this way, a process is statically assigned to a processor at run time, but when to execute it should be determined by a local run-time scheduler. We call this pseudo-dynamic (or iterative static) scheduling. The drawback in this approach is that dependency relations among processes have been neglected in the assignment procedure. To remedy this, we propose a dynamic workload balancing scheme to adjust workloads at run time. In addition, a fine-grained synchronization mechanism is introduced to exploit the concurrency of an algorithm as much as possible.

### 1.3.2 System Overview

The system configuration is shown in Figure 1.2. The heart of the system is a pool of CTUs and ICUs which can be configured into clusters at run time via the Interconnection Network and under host control. The host is responsible for a low density ray casting from which workloads are estimated and the space is partitioned into sections/sectors that are balanced in terms of computational load. Based on this partitioning, the Interconnection Network is configured to form clusters. More on this subject can be found in [10]. The host controls input/output operations such as the distribution of patches through the Distributor for processing and waiting for intensity messages coming from the Collector. Because we use BSP to form sections/sectors, each patch can be easily assigned with a tag that denotes the address of its destination processor. The Distributor consists of a tree of switches controlled by the tag of a patch. The Collector packs the contribution of ray-patch hits into an intensity message and sends it to the host. Since the intensity of a patch can be totally or partially determined in a cluster, the Collector can be viewed as a collection of units distributed over clusters. In the following, we explain two main modules: CTU and ICU. The memory module will be discussed in Subsection 1.3.3.

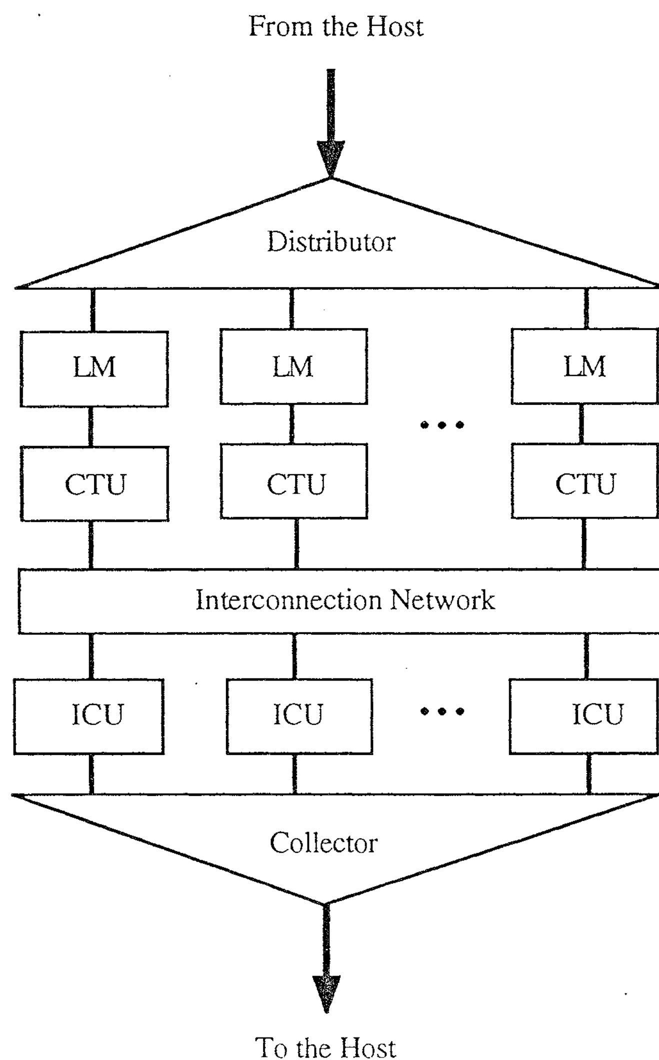


Fig. 1.2. System configuration.

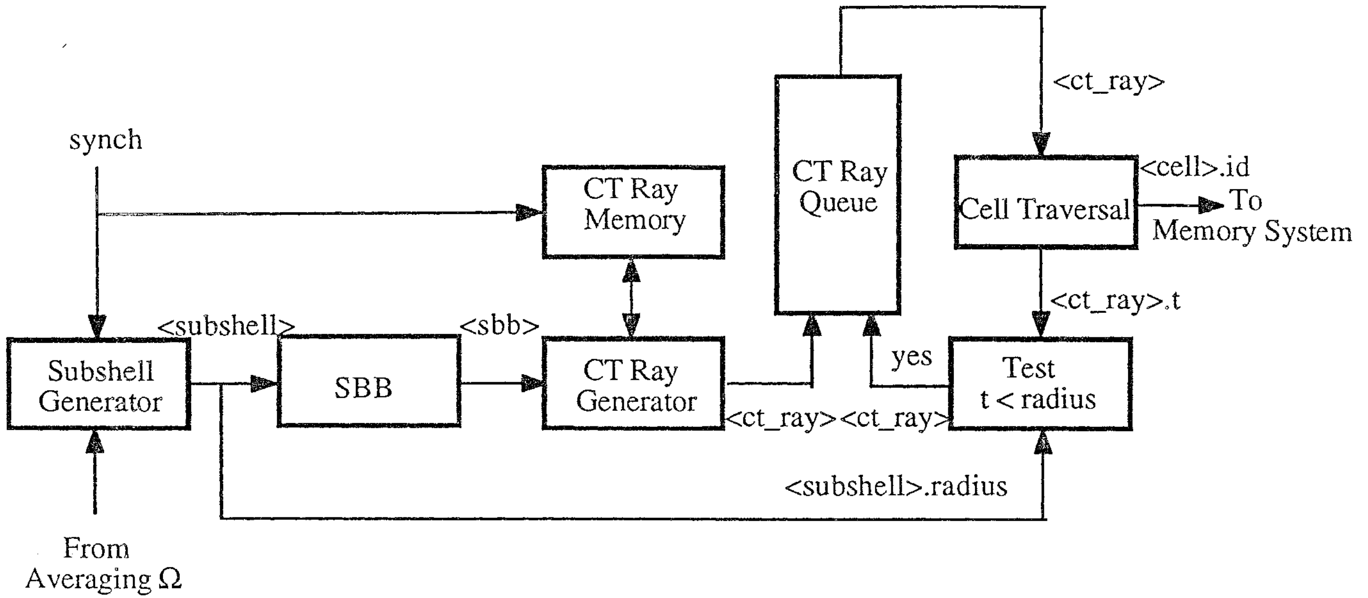


**Fig. 1.3.** Block diagram of CTU.

The block diagram of CTU is depicted in Figure 1.3. To start with a new subshell generated by *Subshell Generator*, a window in *CT Ray Memory* is defined by *CT Ray Generator* based on the spherical bounding box of the subshell. Each ray stored in *CT Ray Memory* has a flag indicating whether it was already shot or not, and only rays not shot yet are output. All the rays in the window are checked and output when necessary. They are enqueued to *CT Ray Queue* in sequence. When a ray is dequeued from *CT Ray Queue*, a single-step cell traversal is done by *Cell Traversal* to determine its next cell address and update its distance to ray origin. It will be enqueued again if its distance is smaller than the current subshell radius. The next cell address will be output to the memory system to access the patches stored in this cell.

The block diagram of ICU, running asynchronously with CTU, is shown in Figure 1.4. A patch retrieved from the memory system first undergoes a coordinate transformation via *GtoR*. Its spherical bounding box is then computed from the transformed vertices of the patch. The spherical bounding box is accumulated by *Averaging* $\Omega$ to set up the next subshell. Based on the spherical bounding box, a window in *IC Ray Memory* can be defined by *IC Ray Generator* that can output a number of rays. Each ray stored in *IC Ray Memory* has a flag indicating whether it already hit or not, and only rays not hit yet are output. They will be enqueued to *IC Ray Queue* in sequence. Together with the accompanying patch, they will flow through a pipelined *Intersection Computation* to calculate distances from the point over which a hemisphere is placed to the intersection points. A ray is declared as hit if the distance to the nearest intersection point is smaller than the current subshell radius and the nearest patch is declared as the intersected patch. The updated ray will be written back to *IC Ray Memory*. We neglect *Deferred Ray Memory* for a moment and leave it for discussion in Subsection 1.3.4. For convenience, we use the notations of $<x>$ and $<x>.y$ to denote a data structure $x$ and a member $y$ in $x$, respectively.

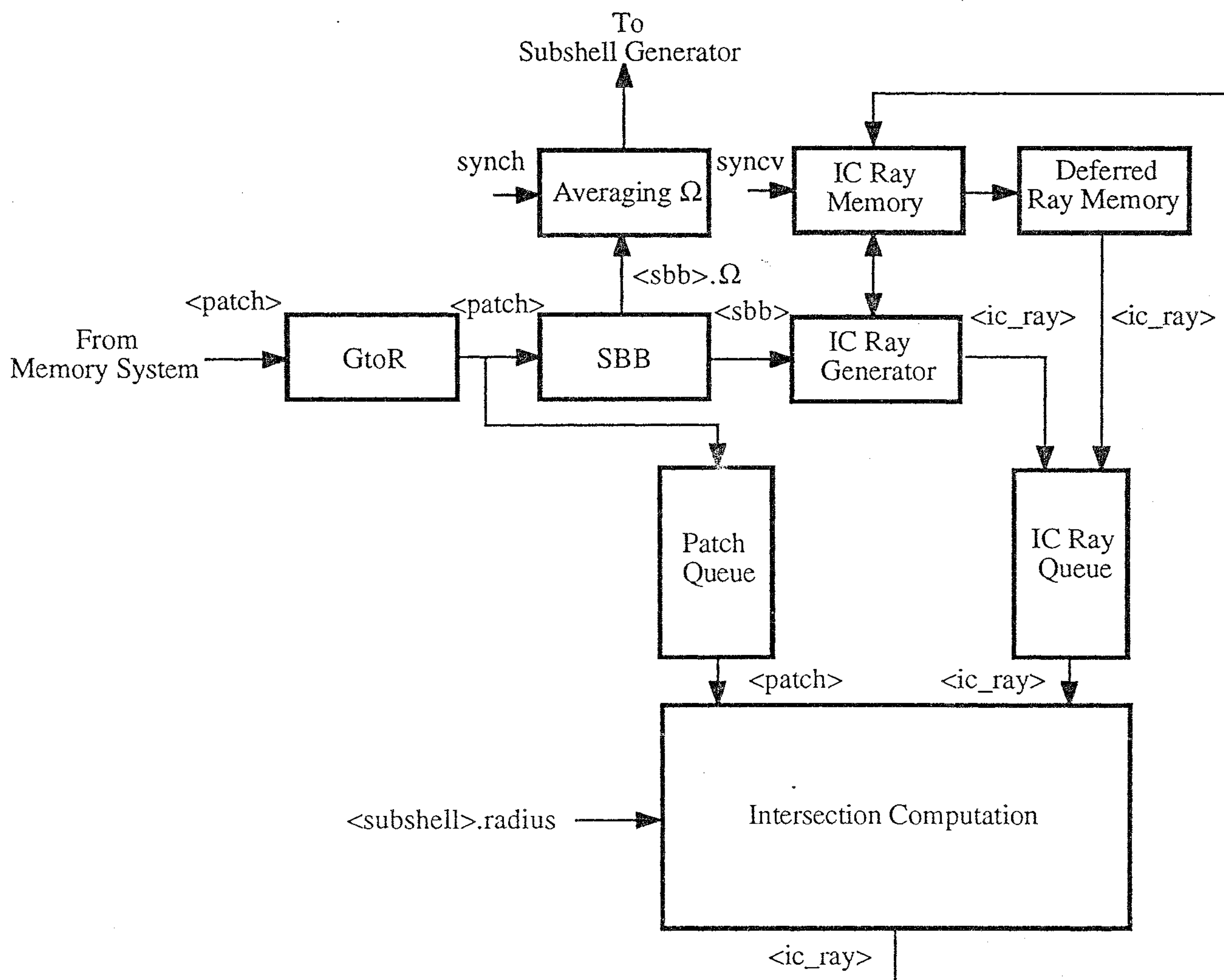


Fig. 1.4. Block diagram of ICU.

### 1.3.3 Memory Structure

In the shelling technique, neighbouring rays will traverse the same cells, and give rise to multiple references to the same patches. Furthermore, in the shelling technique, successive references to cells or patches are made to entries in a local neighborhood of the object space. Due to this so-called locality of reference, cache memories seem appropriate within the proposed memory structure. Nonetheless, it turns out that the derived memory bandwidth still cannot keep up with the required processing throughput. For achieving a better balancing between processing throughput and memory bandwidth, we propose a memory structure which is a hierarchy of resident set, cache, and main memory. A resident set is a memory scheme which can be accessed directly and with less overhead than that associated with a cache memory. For a reference to a patch, the data management algorithm will first check if the resident set contains this patch. If this is the case, then it can be retrieved directly from the resident set. Otherwise, the algorithm will check the cache and finally, if necessary, main memory. In this Subsection, we first describe a two-step procedure: (1) patch identification and (2) patch classification, that can determine patches to be stored in local resident sets (or local memories). After that, different policies in cache design are investigated.

*Patch Identification*

For practical scenes, a relatively small percentage of patches in a database usually account for a large percentage of references to the database. This is because (1) there exist some big patches which will account for many references to the database, and (2) there might be many patches hidden by other patches. Certainly, those big patches which are not hidden by other patches are potential candidates in the resident set. The question is how to search for them with low overhead. In the shelling technique, a low density ray casting is advocated for scheduling purpose. Why not just use those patches found by the low density ray casting as potential candidates in the resident set for the high density ray casting? This is because only big patches can probably be captured by the low density ray casting. We use an example as shown in Figure 1.5, 1.6 to demonstrate that.

Suppose that the frequency function[3] $f_h$ of the high density ray casting is known, and let $U_h$ be the usage function[4] derived from $f_h$. If we are allowed to select the resident set from $U_h$, then an optimum solution can be obtained. Certainly, this will not be the case. Let $f_l$ and $U_l$ be the frequency function and the usage function of the low density ray casting. Instead of selecting the optimum resident set $R_h(k) = \{U_h(1), U_h(2), \ldots, U_h(k)\}$, only $R_l(k) = \{U_l(1), U_l(2), \ldots, U_l(k)\}$ can be selected by the low density ray casting. Then, the *effectiveness* $E(k)$ of $R_l(k)$ is defined as

$$E(k) = \frac{\sum_{i=1}^{k} f_l(U_l(i))}{\sum_{i=1}^{k} f_h(U_h(i))} \text{ for } 1 \leq k \leq N.$$

where $N$ is the number of patches found in the low density ray casting.

The $E(k)$ can be used to indicate the effectiveness of the resident set selected by the low density ray casting. Some results of practical scenes will be shown in the next Section.

---

[3]A frequency function $f$ is a function that returns the number of occurrences of a patch in a reference string.

[4]Patches in a reference string can be ordered in sequence based on the number of occurrences of each patch. A usage function $U$ is a function that returns a patch by using the usage sequence of the patch. That is, $U(1)$ is the most frequently used patch and $U(N)$ is the least frequently used patch.

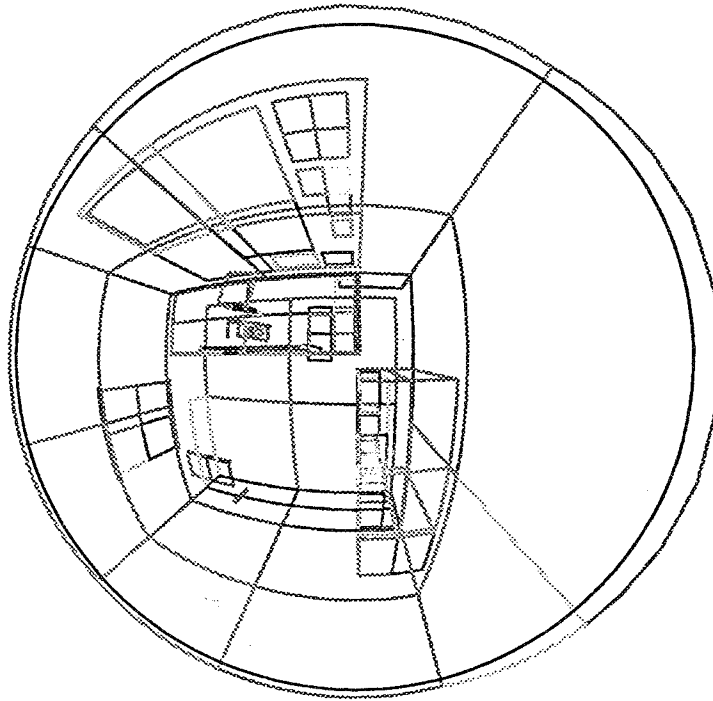

**Fig. 1.5.** An example showing the patches captured by the low density ray casting.

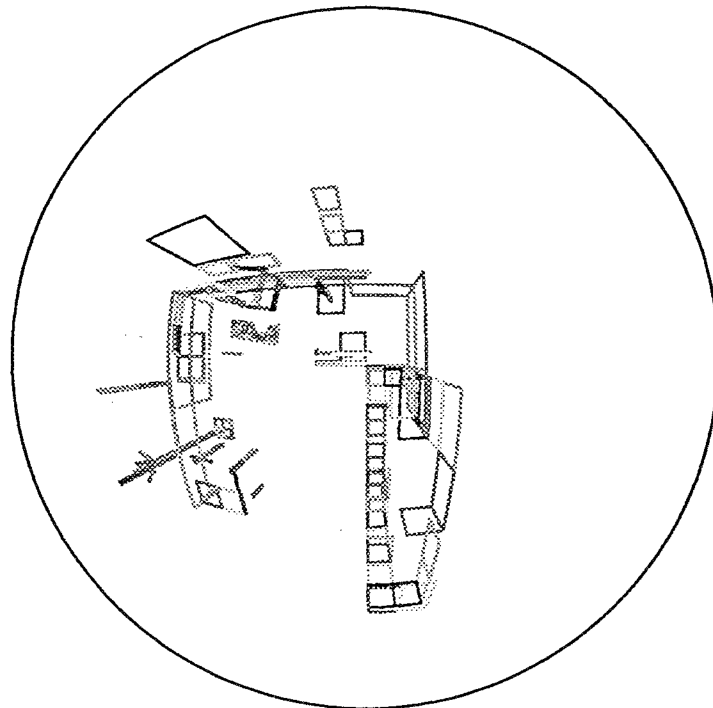

Fig. 1.6. The patches captured by the high density ray casting but not shown in Figure 1.5.

*Patch Classification*

Since each processor (ICU/CTU) is responsible for a section/sector which is a part of the entire space, only those patches intersecting with this section/sector are potential candidates for its resident set. As described above, workload balancing is achieved by recursively applying a BSP technique to form sections/sectors. For BSP level $l$, the entire space which is initially subdivided in uniform subspaces, is subdivided into $2^l$ parts by using $l$ median planes. The $l^{th}$ bit of a patch's tag that denotes the address of its destination processor is determined by classifying to which part it belongs. It is sufficient to classify subspaces against median planes. Only when a median plane is inside a subspace, we classify patches residing in the subspace against the median plane. We now pursue patch classification in this sense. We need the following theorem to derive the spherical bounding box of a polygonal patch, which is essential to the patch-classification step.

**Theorem 1:** Let $H$ be a unit hemisphere with center $O$ and normal vector directed at $Z$ direction, let $P$ be a polygonal patch with vertices $V_i, i = 1, 2, \ldots, v$, and let **CH** be the convex hull on spherical geometry of $P$ on $H$. Furthermore, let $\theta_{min}$, $\theta_{max}$, $\phi_{min}$, and $\phi_{min}$ be the minimal and maximal $\theta$ and $\phi$ angles of the vertices of $P$. Then, S, the set of directions bounded by

$$\begin{cases} \theta_{min} \leq \theta \leq \theta_{max}, \\ \text{Max}(0, \phi_{min} - \varepsilon_{max}) \leq \phi \leq \text{Min}(\frac{\pi}{2}, \phi_{max}), \end{cases}$$

is a superset of that bounded by **CH**, where

$$\varepsilon_{max} = cos^{-1}(\frac{\sqrt{1 - 2sin^2(\frac{\Delta\theta}{4})}}{1 - sin^2(\frac{\Delta\theta}{4})}),$$

and $\Delta\theta = \theta_{max} - \theta_{min}$.

**Proof:** (refer to [8]).

In case of a Bezier patch, 16 control points are used instead to derive the spherical bounding box of the patch. By definition, a subspace is a region bounded by two constant-$\theta$ and two constant-$\phi$ planes. The spherical bounding box of a patch is also defined by two constant-$\theta$ and two constant-$\phi$ angles. So the patch classification can be done by just comparing those angles.

*Cache Design*

There are four placement policies in cache design: direct, fully associative, set-associative, and sector mappings (see [4]). We have implemented the first two policies in our cache design. Direct mapping is the simplest one in the sense that a simple rule: address $i$ in main memory maps to the frame $i$ *modulo* $S$ of a cache with size $S$, is applied for both placement and replacement policies. Furthermore, it does not rely on a special hardware for an associative search of address tags. On the contrary, fully associative mapping is the most flexible one: an address in main memory can map to any frame of a cache and almost any replacement policy can be implemented. However, its performance relies on a fast associative search of address tags. The results of using those two policies will be shown in the next Section.

### 1.3.4 Synchronization Mechanism

As pointed out earlier, we have neglected dependency relations among processes in the assignment procedure, and let a local run-time scheduler determine when a process is to be executed. As a result, our approach can tolerate more data dependency if there exists a synchronization mechanism that can support fine-grained parallelism.

We borrow a concept called I-structure memory from dataflow concept for this purpose. Each I-structure memory location has presence bits indicating whether it is full or empty. Each location is permitted to be written only once and any read of an empty location is deferred until the corresponding write occurs. It is this concept that allows us to initiate many subshells in parallel. This can be explained by using a Task Precedence Graph among subshells as shown in Figure 1.7. When the processing of the current Subshell (0,0) is completed , we can start with both Subshell (0,1) and Subshell (1,0). A patch found in Subshell (0,1) can only start testing against the rays leaving Subshell (0,0) those leaving Subshell (1,0) being available only through the use of I-structure memory as shown in Figure 1.8. The presence bit of a ray in Subshell (1,0) remains empty until Subshell (1,0) is completed. The ray will be stored in *Deferred Ray Memory* (see Figure 1.4) and tested against the accompanying patch only when the presence bit has become full. In this approach, the processing of subshells is reminiscent of wavefronts swept over entire space. It is current challenge to determine subshells and give them an ordering so that processing wavefront can propagate with high throughput. We shall not discuss this issue in this paper.

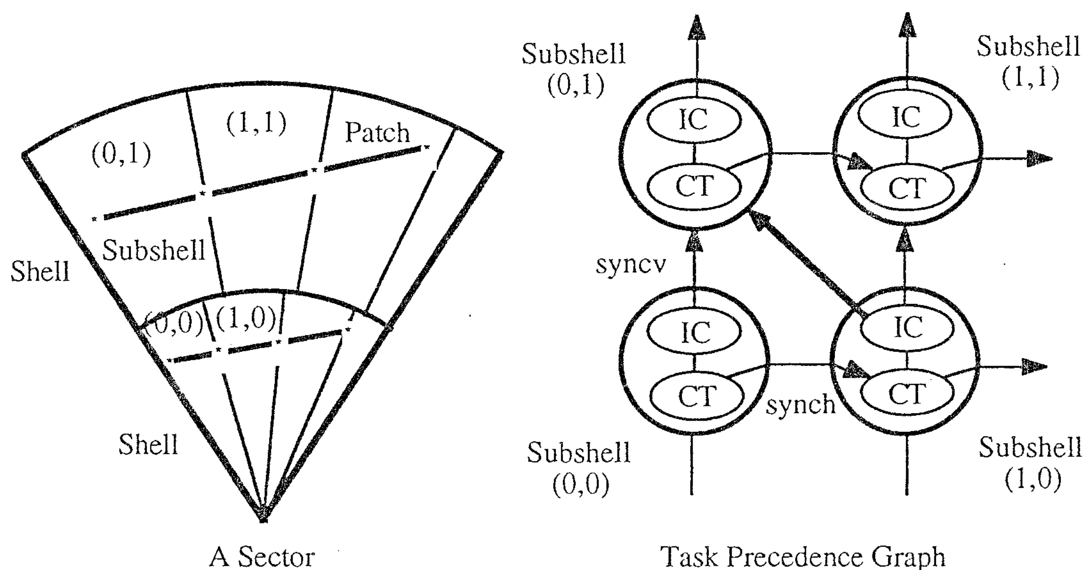


**Fig. 1.7.** Task precedence graph among subshells.

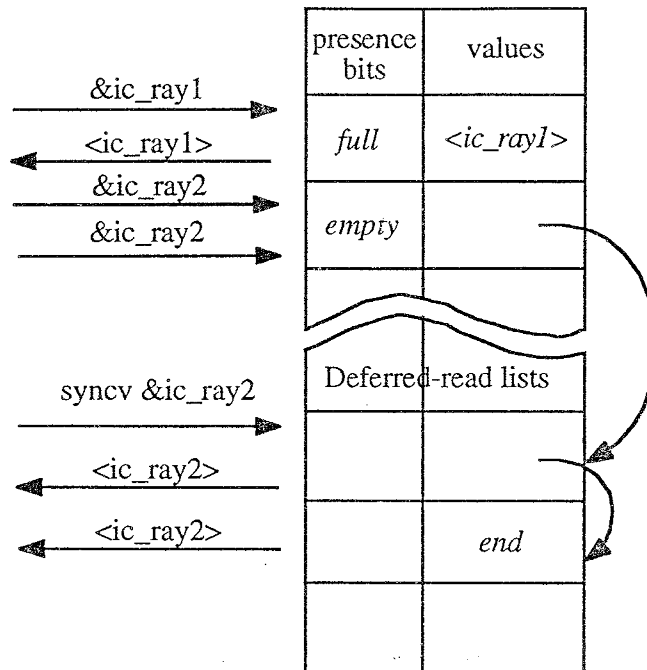


**Fig. 1.8.** I-structure memory.

### 1.3.5 Primitive Functions

In this Subsection, we use pseudo-code with some C notation to explain the functions of three main primitives: Spherical Bounding Box Computation, Intersection Computation, and Cell Traversal. The pseudo-codes for them are given as follows. As can be seen they are tailored to hardware implementation.

SphericalBoundingBox(*patch*)
  if *patch* is a Polygon
    $\Delta\theta$ = ComputeSpanningAngleTheta(*patch*, $\&\phi_{min}$, $\&\phi_{max}$);
    ComputeAnglePhi(*patch*, $\&\phi_{min}$, $\&\phi_{max}$);
    $\varepsilon$ = ComputeErrorTerm($\phi_{min}, \Delta\theta$); /* Can use Table Look-up */
    $\phi_{min}$ = Min$(0, \phi_{min} - \varepsilon)$;
  if *patch* is a Bezier
    $CH_{xy}$ = ConvexHullxy(*patch*);
    $\Delta\theta$ = ComputeSpanningAngleTheta($CH_{xy}$, $\&\phi_{min}$, $\&\phi_{max}$);
    ComputeAnglePhi(*patch*, $\&\phi_{min}$, $\&\phi_{max}$);
    $\varepsilon$ = ComputeErrorTerm($\phi_{min}, \Delta\theta$); /* Can use Table Look-up */
    $\phi_{min}$ = Min$(0, \phi_{min} - \varepsilon)$;

```
point O; /* a sample point */
Intersection(O, ray, patch);
        triangle = Triangulize(patch);
        for each triangle with vertices A, B, C
                n_AB = ComputeNormal(O, A, B);
                n_BC = ComputeNormal(O, B, C);
                n_CA = ComputeNormal(O, C, A);
                d_AB = DistanceFromPlane(n_AB, ray);
                d_BC = DistanceFromPlane(n_BC, ray);
                d_CA = DistanceFromPlane(n_CA, ray);
                if d_AB >= 0 and d_BC >= 0 and d_CA >= 0
                        Subdivide(d_AB, d_BC, d_CA, &u, &v);
```

```
int A_x, A_y, A_z; /* x, y, z components of cell address */
float d_x, d_y, d_z; /* x, y, z components of the direction vector of a ray */
float t_x, t_y, t_z; /* x, y, z components of the distance parameter of a ray */
CellTraversal(ray)
        t = MinimalDistance(ray);
        if t_x == t
                A_x = A_x + Sign(d_x);
        if t_y == t
                A_y = A_y + Sign(d_y);
        if t_z == t
                A_z = A_z + Sign(d_z);
```

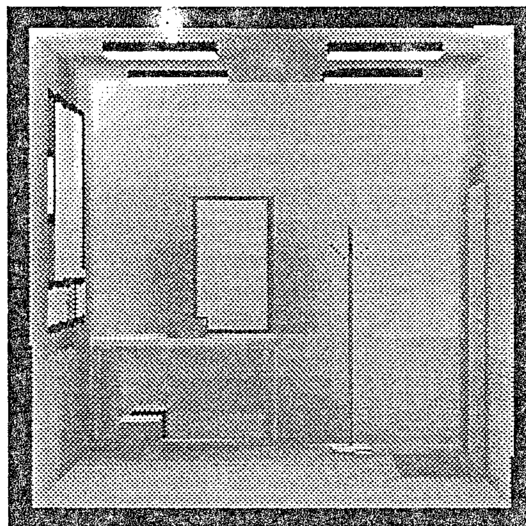

Fig. 1.9. Test scene *scene 1* contains 244 patches.

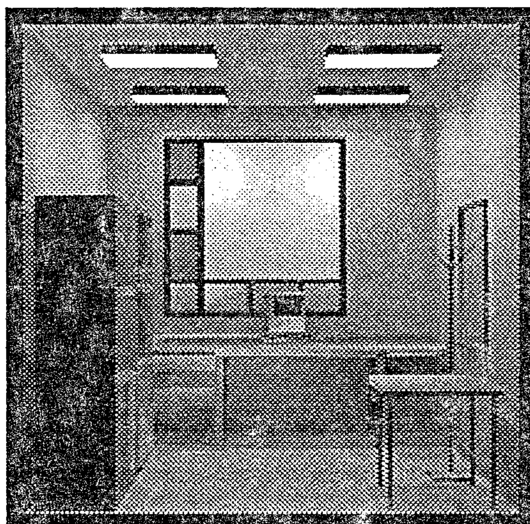

**Fig. 1.10.** Test scene *scene 2* contains 1473 patches.

## 1.4 The Results

In this Section, the results of two test scenes as depicted in Figure 1.9 and Figure 1.10 are shown.

- The Effectiveness of Low Density Ray Casting (refer to Figure 1.11, 1.12)

  The effectiveness of low density ray casting is about 0.7–0.8 when Hi_Low_Ratio[5] is chosen as twice the ADRC of a scene. A highly effective resident set can be selected by low density ray casting with a small overhead relative to high density ray casting when a scene consists of many large patches.

- The Miss Ratio of Two Replacement Policies (refer to Figure 1.13, 1.14)

  Two placements policies, i.e., direct and fully associative mappings, have been implemented in our cache design. In fully associative mapping, we choose LFU (Least Frequently Used) as the replacement policy. From the figures, we see that fully associative mapping is better than direct mapping as expected and a reasonable miss ratio can be achieved with small cache size.

- Speedup (refer to Figure 1.15)

  For *test scene 2*, 15% more speedup can be obtained when using the proposed memory structure. The result only shows one progressive refinement step of the radiosity method. A complete system level simulation is under way by using BONeS which is a block oriented network simulator.

---

[5] Hi_Low_Ratio represents the proportionality of the number of rays shot in high and low resolution ray castings.

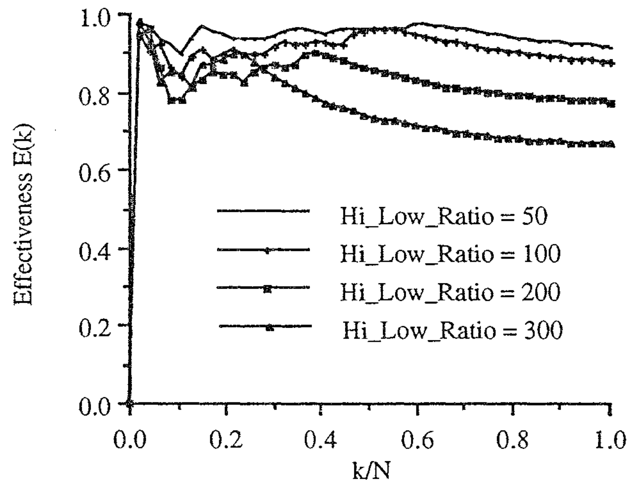


**Fig. 1.11.** The effectiveness of low density ray casting for *scene 1* with ADRC = 124.01.

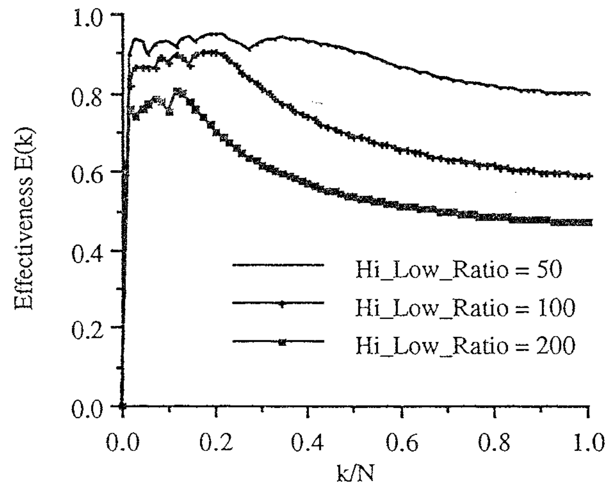


**Fig. 1.12.** The effectiveness of low density ray casting for *scene 2* with ADRC = 76.01.

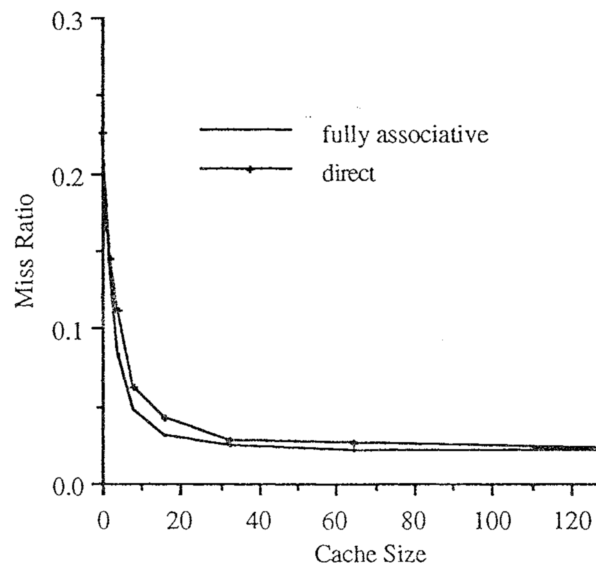


**Fig. 1.13.** Two placement policies for *scene 1* with ADRC = 124.01.

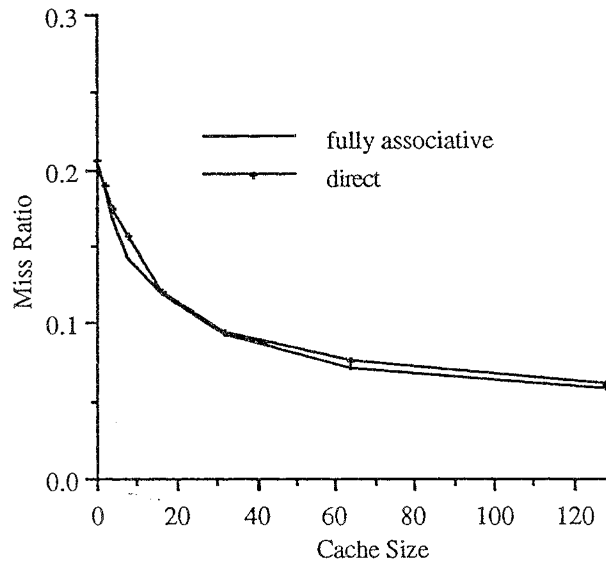


**Fig. 1.14.** Two placement policies for *scene 2* with ADRC = 76.01.

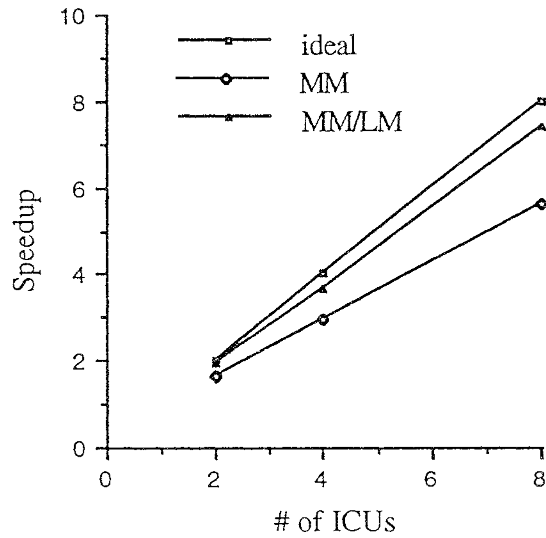


**Fig. 1.15.** Speedup vs the # of ICUs.

## 1.5 Conclusion

In this paper we proposed a parallel-pipelined architecture that is well-suited for the radiosity method. At the beginning of each progressive refinement step, a pseudo-dynamic (or iterative static) scheduling is invoked to redistribute data objects and associated workloads to processors. Together with a proposed I-structure memory, it offers an economical way to make the most of concurrency of an algorithm. Further improvement in performance is accomplished by using a hierarchical memory structure. Block diagrams have been defined that allow a system level simulation to be run under a block oriented network simulator called BONeS.

### Acknowledgements

This research has been supported in part by the commission of the EEC under the ESPRIT program, project BRA 3280 (NANA) and by the Dutch Technology Foundation under contract DEL 99.1982. The author would like to thank Prof. F.W. Jansen and Dr. A.J.F. Kok from the Informatics Department for many valuable comments. Thanks are also due to G.J. Hekstra and M.T. Verelst for many fruitful discussions.

## 1.6 References

[1] J. Cleary, B. Wyvill, G. Birtwistle, and R. Vatti. Multiprocessor ray tracing. *Computer Graphics Forum*, pages 3–12, 1986.

[2] M. Dippe and J. Swensen. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. *SIGGRAPH'84*, pages 149–157, 1984.

[3] S. Green, D. Paddon, and E. Lewis. A parallel algorithm and tree-based computer architecture for ray traced computergraphics. *Parallel Processing for Computer Vision and Display*, 1988.

[4] K. Hwang and F.A. Briggs. Computer architecture and parallel processing. *McGraw-Hill Inc.*, pages 98–118, 1984.

[5] K. Murakami, K. Hirota, and M. Ishii. Fast ray tracing. *FUJITSU Sci. Technical Journal*, pages 150–159, 1988.

[6] T. Naruse, M. Yoshida, T. Takahashi, and S. Naito. Sight : A dedicated computer graphics machine. *Computer Graphics Forum*, pages 327–334, 1987.

[7] T. Priol and K. Bouatouch. Static load balancing for a parallel ray tracingon a mimd hypercube. *The VisualComputer*, pages 109–119, 1989.

[8] L.S. Shen and Ed. F. Deprettere. A hierarchical memory structure for the 3d shelling technique. *Technical Report, Delft University of Technology*, 1991.

[9] L.S. Shen, Ed. F. Deprettere, and P. Dewilde. A new space partitioning for mapping computations of the radiosity ontoa highly pipelined parallel architecture (i). *Fifth Eurographics Workshop on Graphics Hardware*, 1990.

[10] L.S. Shen, F.A.J. Laarakker, and E. Deprettere. A new space partitioning for mapping computations of the radiosity ontoa highly pipelined parallel architecture (ii). *Sixth Eurographics Workshop on Graphics Hardware*, 1991.