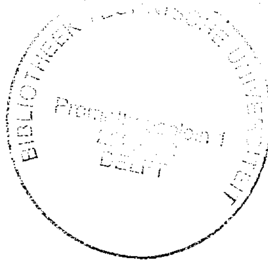
595513
378952
TR diss 2277

TR diss
2277

A Parallel Image Rendering Algorithm and Architecture Based on Ray Tracing and Radiosity Shading

Li-Sheng Shen



Delft University of Technology
September 1993

A Parallel Image Rendering Algorithm and Architecture Based on Ray Tracing and Radiosity Shading

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft, op gezag van de
Rector Magnificus prof. ir. K.F. Wakker,
in het openbaar te verdedigen ten overstaan van een
commissie aangewezen door het College van Dekanen
op maandag 25 oktober 1993 te 10.00 uur

door

Li-Sheng Shen

geboren te Taipei
electrotechnisch ingenieur

Dit proefschrift is goedgekeurd door te promotor
Prof. dr. ir. P. Dewilde

Dr. ir. Ed F. Deprettere heeft als begeleider in
hoge mate bijgedragen aan het totstandkomen
van het proefschrift

To Li-Wen and to my parents

Color Section

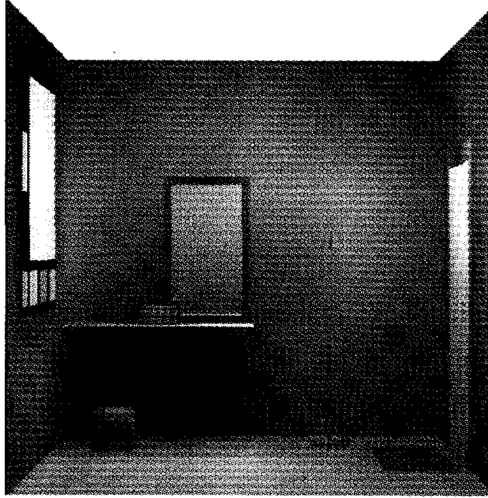


Plate 1: *lobby*

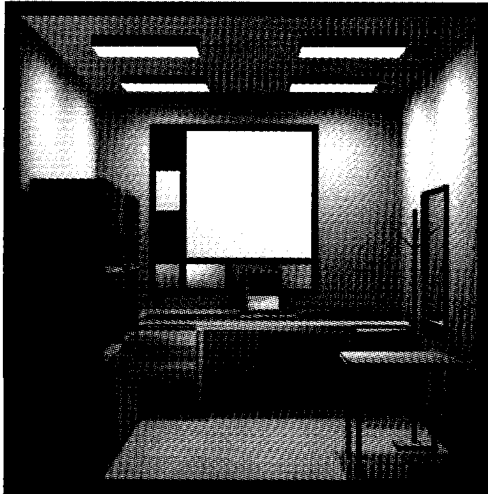


Plate 2: *room 16.27*

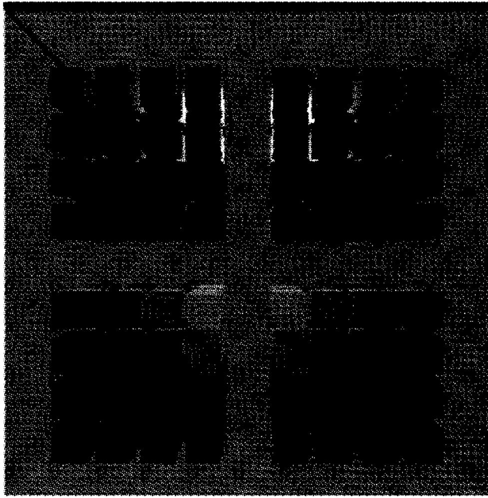


Plate 3: *array of cubes*

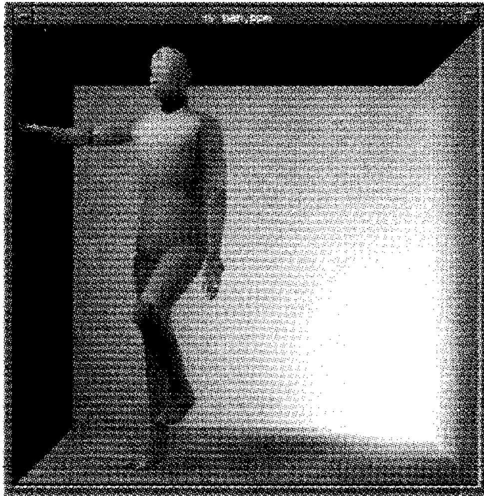


Plate 4: *man*

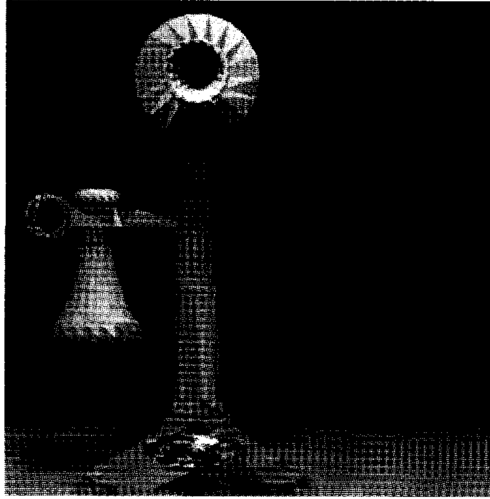


Plate 5: *phone*

Contents

1. Introduction **1**

1.1. Motivation 1

1.2. Overview of the Thesis 6

2. Photo-Realistic Rendering **10**

2.1. Introduction 10

2.2. The Radiosity Method 12

2.2.1. The Form-Factors 13

2.2.2. Radiosity Solutions 22

2.3. Ray Tracing 24

2.4. Two-Pass Approach 25

2.5. A Proximity Enforced Algorithm 26

2.6. Concluding Remarks 28

3. A New Space Partitioning Technique **30**

3.1. Introduction 30

3.2. A Naive Algorithm 31

3.3. The Conventional Technique 34

3.4. The Shelling Technique 38

3.4.1. Issues in the Shelling Technique 43

3.5. A Formal Comparison 75

4. Mapping onto a Pipelined Parallel Architecture **82**

4.1. Introduction 82

4.2. Geometry Based Grain Packing 85
4.3. Runtime Scheduling 90
4.4. Application-Specific Runtime Scheduling 92
4.4.1. Basic Terms 93
4.4.2. Tackling the Scheduling Problem: Some Heuristics 97
4.5. Concluding Remarks 116
5. The Radiosity Engine 119
5.1 Introduction 119
5.2 System Architecture 122
5.2.1. System Configuration 122
5.2.2. Memory Structure 124
5.2.3. Network Design 128
5.2.4. Synchronization Mechanism 135
5.3 Outlining Functionality 140
5.3.1. Introduction 140
5.3.2. Data Structures 142
5.3.3. Radiosity Engine System 146
5.3.4. Cell Traversal Unit 148
5.3.5. Intersection Computation Unit 153
5.3.6. Memory System 160
5.3.7. System Bus Interface 163
5.3.8. Arbiter 165
6. Performance Measurements 167
6.1. Network Performance 167
6.2. Memory Performance 170
6.3. System Performance 181
6.3.1. Overall Computation Time 182
6.3.2. Speedup 182
6.3.3. Pipeline Efficiency 191
7. Concluding Remarks 197

Bibliography **201**
Acknowledgements **209**
About the Author **210**

Chapter 1

Introduction

1.1. Motivation

This dissertation is about the parallelization of an algorithm that has become known as *the two-pass radiosity method* [Hec90] [KJW93] [SP89] [WCG87], for rendering artificial scenes with photo realism on the screen of a workstation. Here, by parallelization of an algorithm, we mean the design of a parallel architecture for the efficient execution of its parallelizable parts. Parallelization, when so interpreted, is a difficult problem, even when the algorithm has a regular precedence graph which is known at compile time [DHW93]. Roughly stated, parallelization of a program is a decomposition of the program into a number of tasks of a certain size which are then assigned to a parallel system with multiple processors, one for each task. To achieve an efficient parallelization, we are faced with the following two fundamental problems:

1. Latency and Synchronization: Arvind's Two Fundamental Issues [AI87]

   Structurally, a parallel system can be thought of as a collection of processors, some number of memory modules and some means for intercommunication, assembled for the purpose of cooperating on the solution of a given problem. Consider the model that either all the memory modules form one global address space (i.e., *centralized-memory system*) so that they are equally accessible from all processors, or the model that the memories are strictly local to processors (i.e., *distributed-memory system*) so that processors communicate directly with one another via messages. Either model demonstrates that there

will necessarily be limitations on time, or latency, to communicate between tasks by way of sending and receiving messages. On the other hand, a program must be logically decomposed into communicating tasks in order to effect parallel execution, implying the need for some sort of time-coordination, or synchronization to preserve dependency constraints. In order to fully exploit the parallelism in an algorithm, all architects of parallel systems must face those two fundamental issues, namely, *latency* and *synchronization.*

2. Speedup Bottleneck: Amdahl's Law [Amd67]

   Consider a program consisting of parallelizable and non-parallelizable parts. Let f be the fraction of the program that is parallelizable, and let T_s and T_p be the execution times of the program on a single processor system and a parallel system with p identical processors, respectively. Clearly, it holds that $0 \leq f \leq 1$ and the fraction of the program which can be run in parallel cannot exceed f. Because the parallel execution time for the parallelizable part can be no less than $f\, T_s/p$, we have

$$T_p \geq (1 - f)\, T_s + \frac{f\, T_s}{p}.$$

   Then the speedup S achievable on the parallel system is

$$S = \frac{T_s}{T_p} \leq \frac{T_s}{(1 - f)\, T_s + f\, T_s/p} = \frac{1}{(1 - f) + f/p}. \qquad (1.1)$$

   Amdahl's law limits the speedup achievable by a parallel system, in particular when a program contains some percentage of non-parallelizable part. For instance, if 5% of the program must be performed sequentially, then the maximum speedup is 20, no matter how many processors are used.

From the above discussion, it is clear that the more finely a program is decomposed into tasks, the greater the opportunity for parallel execution. However, there is a commensurate increase in the overhead of inter-task communication and synchronization requirements. In this sense, if the efficiency of a parallelization is measured in terms of execution time and expressed as a function of task size, then the optimum is a non-trivial compromise between speedup and overhead which are both increasing with decreasing task size. This is shown in Fig. 1.1 [Sar87] in which the execution time of the sequential program is normalized to 1, and task size s is relative to the size of the sequential program. The curve labelled $1/S$ is the inverse of the speedup given in Eq. 1.1 with $p = 1/s$.

The curve labelled Q is the overhead due to inter-task communication and synchronization requirements. At one extreme, the task size s is very small and parallelism can be exploited maximally (f -> 1 and s -> 0; S -> ∞). However, the overhead dominates the useful computation. At the other extreme, the task size $s = 1$ and no parallelism is exploited ($S = 1$) and the overhead is minimal ($Q = 0$). In both cases, the efficiency of parallelization is low. As

can be seen from the figure, the optimum task size s^* leading to a minimal execution time is somewhere in between these two extreme situations.

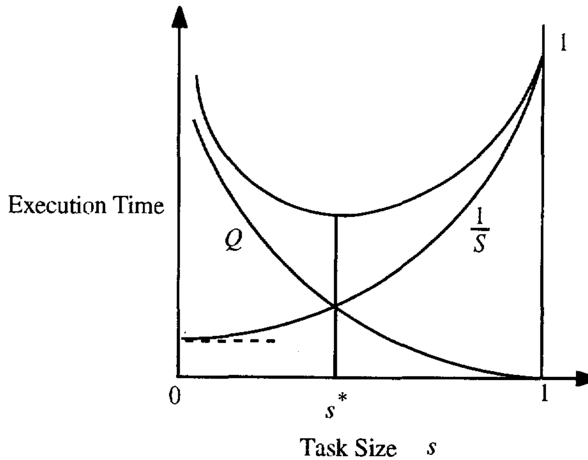


Figure 1.1: The Speedup-Overhead tradeoff.

It must be noticed that Fig. 1.1 is but a rough indication of the speedup versus overhead trade-off, because the optimum depends not only on the topology of the architecture, but also on the structure of the algorithm. Thus, achieving an efficient parallelization is much more involved than simply finding the intersection of Q and $1/S$ curves. As a result, it is almost impossible to come up with an efficient parallel architecture for a wide range of algorithms. Practice has shown that general-purpose architectures cannot meet the requirements of large speedups and low overheads simultaneously. To overcome this bottleneck, one can go two directions of specialization. One is the "systolic-way", that is to require the algorithms to be affine nested-loop algorithms [DHW93], the other is the "special-purpose way", that is focusing on a specific application. Parallel architecture design of affine nested-loop algorithms has achieved great progress in the last decade and its success is due to the fact that a design methodology has emerged which is based on a well-defined model and methods which have sound mathematical support. This research does not fall in this class of specialization, yet experience of designing systolic-like architectures has had a great impact on it. Instead, a specific application for which a systolic solver is not meaningful has been the point of departure. Illustrative problems can be found in the fields of signal processing, image processing and computer graphics. The problem we shall deal with in this dissertation is rendering artificial scenes. Parallel implementation of a non-systolic yet specific algorithm leads to a feasible optimization problem which involves the design of a *good* parallel algorithm *and* a *good* parallel architecture, which form together a

good algorithm-architecture pair in the sense that it approaches an optimum in terms of speedup and overhead trade-off (see Fig. 1.1). Thus, the key to success here is combined algorithm development and architecture design.

Some attempts to combine algorithmic and architectural design systematically have been reported in the literature. One example is the so-called *systolic algorithm,* which is a merging of the notions of *iterative algorithms* and *systolic architectures* (see [MVD92]). Another example is the so-called algorithmic engineering approach [McW92]. Both are dealing with systolic-like static algorithms, and the "architectures" are conceptual and overly simplified: they are merely a graphical representation of a massively parallel algorithm.

These approaches can hardly be classified as combined algorithm-architecture design because the emphasis is clearly on massively parallel algorithm design and the architecture is enforced by the algorithm. Moreover, aiming at a massively parallel algorithm is not always meaningful, as the following two points may clarify:

1. The amount of parallelism in algorithms is one factor that has impact on the efficiency of parallelization A common believe is that inherent parallelism should be explored as much as possible. However, this is not always true, in particular when parallelism is hiding wastes of operations. A prominent example is ray tracing. Ray tracing is an algorithm which can produce realistic images. Rays are sent from a viewpoint through every pixel on a display screen and traced as they are reflected and transmitted by objects in the space behind the screen. When a ray hits an object, new rays may be generated, due to reflection, transmission, and/or light sources. These new rays are in turn traced. The most time-consuming computations in the ray-tracing algorithm are ray-object intersections. One may conceive an algorithm which tests all the rays with all the objects in parallel so that a huge amount of parallelism can be realized. However, the operation count in this algorithm is extremely high, at the order of $N \times R$, where N is the total number of objects and R is the total number of rays, and is far beyond what is necessary to be computed, i.e., R intersection points, since one ray can only intersect one object. This amount of parallelism is certainly not meaningful because most parallelism of this form, which may even not be attainable by the underlying architecture, is doomed to be wasteful.

2. Having a large amount of parallelism in an algorithm is no guarantee to having a well-performing implementation. First of all, we must seek such architectures that can mitigate the overhead of latency and synchronization. Once a particular architecture is chosen, the utilization of concurrency depends totally on how the concurrent resources are allocated and managed. This is the resource management problem.

This dissertation is also an attempt to combine algorithm and architecture design. There is, however, quite a difference between this attempt and those mentioned above. As stated previously, the point of departure has been a rendering problem from the field of computer graphics. This problem does neither lead to a static algorithm nor to an architecture that is naturally enforced by a massively parallel algorithm. Communication and synchronization

overheads emerge straight from the beginning and finding a good, if not optimal, algorithm-architecture pair is a true iterative and interactive process.

We aim at developing a parallel image rendering algorithm and architecture based on the so-called two-pass approach. This approach is demanding orders-of-magnitude more processing power for a single processor if we wish to make a state-of-art image in real-time or even interactive time. An obvious answer to this dilemma lies in parallel processing, and all above-mentioned algorithm/architecture issues will turn up, to wit:

1. Combined Algorithmic and Architectural Design is Essential

   We will discuss the two-pass approach in chapter 2. Basically, it is a ray-tracing based algorithm. As stated previously, for ray tracing, it is not worth testing all the rays with all the objects. This is because most parallelism of this form, which may not even be attainable by the underlying architecture, is wasteful. Instead of exploring parallelism in the algorithm blindly, we should search for a certain form of parallelism that is profitable to exploit in an accompanying architecture. This unusual algorithmic characteristic reveals the strong relationship between the algorithm and the architecture.

2. Latency Problem

   As mentioned before, the most time-consuming computations in the ray-tracing algorithm are ray-object intersections. To proceed with one ray-object intersection, first of all, one needs the data representing the ray and the object. Owing to the large amount of data resulting from the running of the algorithm, the latency for memory access becomes serious if inappropriately handled. Imagine the time required to access a large database containing hundreds of thousands or even a million objects (10 Mbytes - 100 Mbytes) in case one wishes to access all pbjects (0.1 M - 1 M objects). Consequently, the parallelism in the algorithm becomes insignificant due to the long latencies.

3. Synchronization Problem

   Synchronization refers to the requirement that one task cannot begin execution until all of its predecessor tasks have completed their execution. To avoid testing blindly with all the objects, a ray may search for objects and test with them only when necessary. Put simply, a ray can start testing with the closest object. If the ray hits the object, then it is not necessary to go further. Otherwise, the ray continues testing with the next closest object and so on until it hits. In other words, a ray-object test must wait for the result of its previous ray-object test.This busy-waiting scheme is actually a synchronization at the *control-level*[1]. As can be seen, the granularity of synchronization at this level is rather fine.

---

[1] Synchronization may take place at control-level or data-level. Fork-Join instructions and data-driven mechanisms are two examples of synchronization at the control-level. Presence_bits associated with registers or memory locations are mechanism for synchronization at the data-level.

The overhead of synchronization will certainly degrade the performance if one wishes to synchronize at that level.

4. Resource Management Problem

   In order to effect parallel execution, a program must be decomposed into a number of tasks in a certain way. The resource management involves: (1) The *grain-size* problem refers to decisions regarding the granularity of tasks and the choices as to which objects should be packed into the same task. (2) The *scheduling* problem refers to the assignment of tasks to available processors.

1.2. Overview of the Thesis

This dissertation is composed of seven chapters. Chapter 1, this one, is introductory.

In chapter 2, we first provide the background for the radiosity method and ray tracing. We then presents a ray-tracing based two-pass approach that serves as basis for the target algorithm. From a hardware perspective, we suggest a proximity enforced algorithm to take advantage of the *data-coherence* property[2] at algorithmic level.

In chapter 3, we present the shelling technique. By and large, it is an algorithm that strives to build the visibility orderings for the sets of input patches and rays. In an attempt to explain the underlying ideas, we start with a "naive algorithm" and discuss its inefficiencies. We then move to the conventional space partition technique, and we show that the conventional technique gives too much constraints to the architectures, which are difficult to be realized by hardware.

In chapter 4, we turn to the resource management problem. We first point out some dynamic behaviours like latencies in memory accesses, communications and synchronizations which make the scheduling problem very difficult to manage and analyze at compile time. This leads to a so-called runtime resource management problem which defers the actual scheduling to some point during program execution. In contrast to compile-time scheduling, runtime scheduling allows to manage resources for highly data-dependent programs and dynamic system environments. However, a major shortcoming of runtime scheduling is that a non-negligible runtime overhead will be introduced. In order to eventually gain in overall performance,

---

[2] The data-coherence property is analogous to the property of locality of reference in virtual memory systems. Locality of reference has two components: *temporal locality* and *spatial locality*. In temporal locality, there is a tendency for a process to reference in the near future those elements of the reference string referenced in the recent past. In spatial locality, there is a tendency for a process to make references to entries in the neighbourhood of the previous reference.

seemingly time-consuming or time-indeterminate algorithms are not feasible even though they are sophisticated and may lead to a better solution in case of compile-time scheduling. This limits feasible scheduling algorithms to those with low complexity and highly efficient implementation. For this reason, we propose a technique called *application-specific runtime scheduling* (*ASRS*) by tailoring the characteristics of this specific application.

In chapter 5, we present the system configuration of the radiosity engine, and discuss some essential issues such as the memory structure, network design and synchronization mechanism and give an outline of the functionality of the system. In order to take advantage of the data-coherence property at all levels, a hierarchical memory which is divided into working register, local memory, cache and main memory is proposed. As for the network design, we compare different network topologies including ring, Illiac, torus and hypercube. It turns out that a mesh-connected network is a good choice. To support fine grained synchronization, we borrow the concept of I-structure memory from dataflow machine theory.

In chapter 6, we quantify the performance of the radiosity engine by using Monte Carlo simulation. Finally, in chapter 7, we give some conclusions.

Chapter 2

Photo-Realistic Rendering

2.1. Introduction

The production of realistic images requires to simulate the propagation of light within an environment. Light leaving a surface may originate from the surface by direct emission, as from a light source, or by the reflection or transmission of incident light. The incident light on the surface can in turn arrive directly from a light source or indirectly by intermediate reflections and transmissions from other surfaces within the environment. Thus realism can only be achieved by taking global illumination into account, which includes the effects of all objects in the environment.

In general, the transfer of light from one surface to another can be thought of as occurring by way of four mechanisms, namely, diffuse to diffuse reflection, specular to diffuse reflection, diffuse to specular reflection and specular to specular reflection (refer to Fig. 2.1). Two commonly used methods of accounting for global illumination are radiosity and ray tracing. Ray tracing, first described by Appel [App68] , only considers specular to specular and diffuse to specular transfer. Although some of the most realistic images have been generated by ray tracing, it ignores the illumination of a diffuse surface by light reflected specularly from another surface (specular to diffuse) and the interreflection of light between diffusely reflecting surfaces (diffuse to diffuse). Some extended ray-tracing algorithms, such as the methods of Kajiya [Kaj86] and Ward et al. [WRC88], have been presented to account for these transfer mechanisms. However, many incoming directions at each sample point on a diffuse surface

which is visible to the eyepoint may have to be sampled, since the significant sources of illumination may be difficult to find. This pixel by pixel determination of intensity imposed by ray tracing from the eyepoint may introduce more work than necessary since the illumination of a diffuse surface as perceived by the eyepoint typically changes relatively slowly from one pixel to the next. In 1984, Goral et al. [GTGB84] introduced a radiosity method based on principles from the field of thermal engineering to model the interreflection of light between diffusely reflecting surfaces. In the radiosity method, the set of sample points for which the intensities are calculated depends on the discretization of the environment surfaces rather than the eyepoint and image resolution. As compared with the ray tracing, the number of sample points required can be greatly reduced. Unfortunately, the standard radiosity method treats only diffuse surfaces, the specular reflection (specular to specular and diffuse to specular) is not taken into account. Immel [ICG86] extended the radiosity method to incorporate specular effects by discretizing a *global cube* placed over a specular surface into a finite number of directions. In this approach, a relationship between a given incoming direction for a specular surface and all outgoing directions for all other surfaces gives the directional intensity for this certain direction. For highly specular surfaces, the discretization of the cube should be very fine (may be down to pixel levels) as the directional intensity changes very quickly from one direction to another. The computation and storage required precludes discretization to the required level. As a result, artifacts appear on specular surfaces.

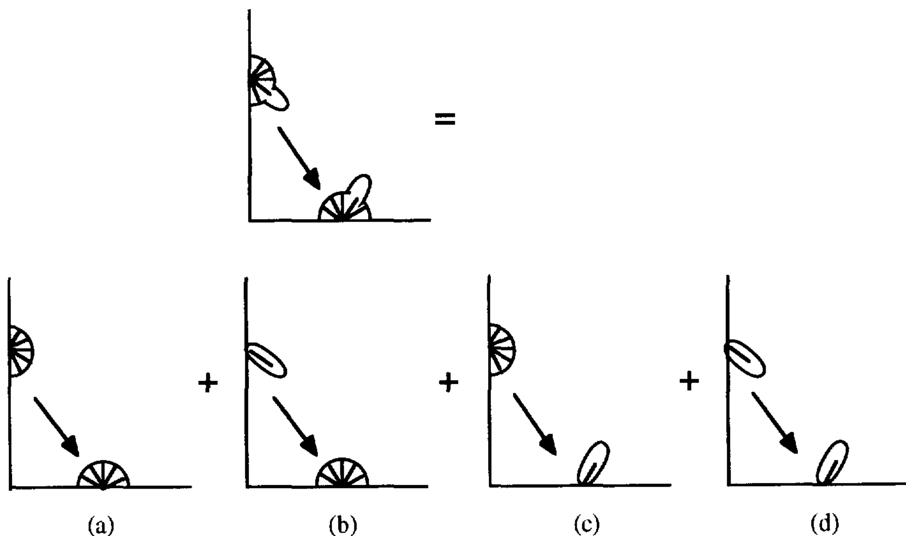


Figure 2.1: The four mechanisms of light transport: (a) diffuse to diffuse, (b) specular to diffuse, (c) diffuse to specular and (d) specular to specular.

From the above discussion, we can conclude that ray tracing is most suited for the display of specular environments, while the radiosity method is most suited for the display of diffuse environments. In order to render a complex environment, several attempts have been made to integrate radiosity and ray tracing. We shall discuss this after introducing the radiosity method and ray tracing.

2.2. The Radiosity Method

The radiosity method, based on principles from the field of thermal engineering, models the interaction of light between diffusely reflecting surfaces and accurately computes the global illumination effects.

To start with, we say that the environment is defined by an enclosure consisting of a number of *surfaces*. All surfaces of the enclosure are assumed to be ideal diffuse reflectors, ideal diffuse light emitters, or a combination of the two. To perform radiosity analysis, the surfaces are discretized into *patches*, for which a constant radiosity is assumed. The radiosity method computes global illumination effects by solving the following radiosity equation that describes an equilibrium energy balance within the enclosure.

$$B_i A_i = E_i A_i + \rho_i \sum_{j=1}^{N} B_j F_{ji} A_j, \tag{2.1}$$

where

Radiosity B:	The total rate of energy leaving a patch.
Area A:	The area of a patch.
Emission E:	The rate of energy (light) emitted from a patch.
Reflectivity ρ:	The fraction of incident energy which is reflected back into the environment.
Form-factor F_{ji}:	The fraction of energy leaving patch j that lands on patch i.
N:	The total number of patches in the environment.

The radiosity equation states that the amount of energy (or light) leaving a surface is equal to the sum of self-emitted light and the reflected light. The reflected light is equal to the light leaving every other surface multiplied by the fraction of that light which lands on the receiving surface and the reflectivity of the receiving surface.

Using the reciprocity relationship for form-factors, that is, $F_{ij} A_i = F_{ji} A_j$, the radiosity equation becomes:

$$B_i = E_i + \rho_i \sum_{j=1}^{N} B_j F_{ij}. \tag{2.2}$$

or in matrix form:

$$\begin{bmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1N} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \cdots & -\rho_2 F_{2N} \\ . & . & \cdots & . \\ -\rho_N F_{N1} & -\rho_N F_{N2} & \cdots & 1-\rho_N F_{NN} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ . \\ B_N \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ . \\ E_N \end{bmatrix} \tag{2.3}$$

Given the reflectivity of each diffuse reflector and the emission of each light emitter, the radiosity of each patch can be obtained by solving the set of equations if the form-factor between each pair of patches is known. These involve: (1) computing the form-factors and (2) solving the radiosity equation. In the following, we shall discuss those two issues.

2.2.1. The Form-Factors

The form-factor, by definition, is the fraction of energy leaving one patch which lands on another. For a non-occluded environment, the form-factor from the differential area dA_i to the differential area dA_j, as shown in Fig. 2.2, is given by

$$F_{dA_i dA_j} = \frac{dA_j \cos\varphi_i \cos\varphi_j}{\pi r^2}. \tag{2.4}$$

By integrating over area A_j, the form-factor from the differential area dA_i to the finite area A_j is given by:

$$F_{dA_i A_j} = \int_{A_j} \frac{\cos\varphi_i \cos\varphi_j}{\pi r^2} dA_j. \tag{2.5}$$

The form-factor from the finite area A_i to the finite area A_j is defined as the area average, and is given by:

$$F_{A_i A_j} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\varphi_i \cos\varphi_j}{\pi r^2} dA_j\, dA_i. \tag{2.6}$$

This expression for the form-factor does not account for the possibility of occluding patches hiding all or part of one patch from another. If hidden patches are to be accounted for, an additional term *HID* must be included in the integrand, as follows:

$$F_{A_i A_j} = \frac{1}{A_i} \int_{A_i}\int_{A_j} \frac{\cos\varphi_i \cos\varphi_j}{\pi r^2} \, HID \, dA_j \, dA_i, \tag{2.7}$$

where *HID* takes the value *1* if the differential area dA_i can *see* the differential area dA_j, otherwise *HID* takes the value *0*.

For simple environments which do not contain occluded patches, the double area integral can be transformed into a double contour integral by using Stoke's theorem, and can be solved analytically [GH71] [SC78]. However, this approach is not feasible to compute form-factors for general complex environments. In the following, we start with a geometric analogue to the analytic derivation, and then discuss some efficient but approximate methods to compute form-factors.

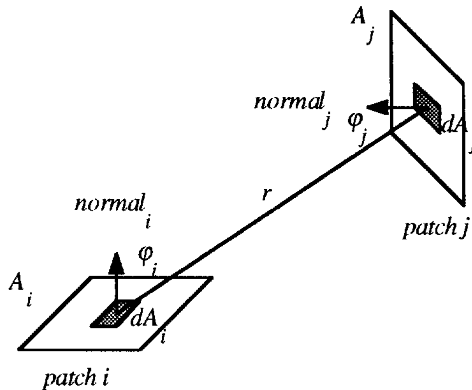


Figure 2.2: Geometry for form-factor derivation.

2.2.1.1. The Nusselt Analogue

A geometric analogue for the form-factor integral was developed by Nusselt [SRHJ78], as illustrated in Fig. 2.3. For a finite area, the form-factor is equivalent to the fraction of the base of the hemisphere covered by projecting the area onto the hemisphere and then orthographically down to the base. The derivation of the form-factor is as follows:

1. Place a unit hemisphere oriented around the normal of a differential area dA_i, and project another differential area dA_j, with a distance r from dA_i, onto the hemisphere. The projected area is given by:

$$dA_j' = \frac{dA_j \cos \varphi_j}{r^2}. \tag{2.8}$$

2. Orthographically project dA_j' onto the base of the hemisphere to get an area:

$$dA_j'' = dA_j' \cos \varphi_i = \frac{dA_j \cos \varphi_i \cos \varphi_j}{r^2}. \tag{2.9}$$

3. The ratio of dA_j'' to the base of the hemisphere is given by:

$$\frac{dA_j''}{\pi} = \frac{\frac{dA_j \cos \varphi_i \cos \varphi_j}{r^2}}{\pi} = \frac{dA_j \cos \varphi_i \cos \varphi_j}{\pi r^2}, \tag{2.10}$$

which is just the form-factor from the differential area dA_i to the differential area dA_j as given in Eq. 2.4.

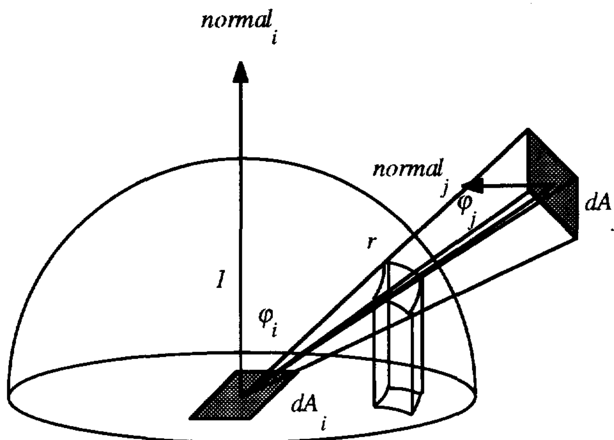


Figure 2.3: Nusselt Analogue.

2.2.1.2. The Hemisphere Method

The Nusselt's analogue gives us a first hint to work on the hemisphere. To compute the form-factor between a pair of patches, the surface of the hemisphere is discretized into small delta areas, each representing a delta form-factor as defined in Eq. 2.10. The form-factor is equal to the sum of the delta form-factors of the delta areas covered when projecting another patch onto the hemisphere placed at the center of one patch. Instead of evaluating the complex double area integral, the form-factor can be computed by the use of projection and summations. For an occluded environment, if two patches project onto the same delta area on the hemisphere, a depth comparison is made to determine which patch is seen through that particular direction by comparing distances to each patch and selecting the closer one. The major difficulties of this approach are:

1. How to derive the projected area when projecting a patch on the hemisphere?

   The difficulty lies in the fact that the projected area is bounded by a complex curved surface. This derivation is rather expensive and not amenable to hardware implementation (and so slow).

2. How to discretize the surface of the hemisphere?

   The main considerations in determining the discretization of the surface of the hemisphere are: (1) The delta form-factors should be easily determined. In general, delta form-factors can either be pre-stored in a memory or just derived on-the-fly. To reduce the storage requirement or simplify the derivation, we could discretize the base of the hemisphere into equal areas, then orthographically project each area onto the surface of the hemisphere. Note that the discrete areas which are not covered by the base of the hemisphere are discarded. With this arrangement, a particular discretization of the surface of the hemisphere is obtained such that each delta area carries the same delta form-factor. So the form-factor can be easily determined by accumulating the constant delta form-factor representing each delta area covered. (2) A good discretization should take surface radiative properties and geometric relationship between sources and receivers into account. Otherwise, aliasing or wasting may happen due to under or over estimations of surface projections, respectively.

2.2.1.2. The Hemi-Cube Method

Another efficient approach to approximate form-factors is the hemi-cube method. In this approach, a hemi-cube instead of a hemisphere is used according to Nusselt's analogue. The hemisphere described above is replace by the upper half of the surface of a unit cube constructed around the center of the source patch. The surface of the hemi-cube (i.e., five planar surfaces) is discretized into square *pixels* (or delta areas) at a given resolution, each representing a delta form-factor (see [CG85]). The other patch is then projected onto the five planar surfaces, and the form-factor is equal to the sum of the delta form-factors of the pixels

(or delta areas) covered by the projected area. The item buffer can be used to determine which patch is seen through a particular direction if two patches project onto the same pixel on the hemi-cube. The advantage of the hemi-cube method is that it can take advantage of current hardware Z-buffer supports which are available on high-end workstations. However, it has the following disadvantages:

1. The regular arrangement of the pixels of the hemi-cube may cause aliasing due to the limited resolution used. A typical example is: Suppose the original projection of a patch covers 6×6 pixels; If the patch is slightly offset, it covers 7×7 pixels instead. Another quite obvious artifact is due to small patches, in particular light emitters. Image a door knob is lit with a small emitter. The projection of the emitter may not cover a pixel, therefore, the door knob may look black even it is viewed from a close distance.
2. The patches are restricted to polygons as it is difficult to scan-convert curved surfaces such as Bezier. A large memory is required to store the database.
3. As compared with the hemisphere method, the hemi-cube discretization yields less accuracy at higher memory costs (see [WJC88]).

2.2.1.1. A Ray-Casting Based Approach

By virtue of the Nusselt's analogue, it seems more natural to place a hemisphere instead of a hemi-cube around the center of a source patch. To overcome the problems of the hemisphere method, a ray casting based approach has been proposed [WEH89] [SP89]. As before, a unit hemisphere, say H, is placed at the center of the front face of a source patch, say O, such that the base of the hemisphere is perpendicular to the normal of the patch, say in Z-direction. The surface of H is discretized into $2 \times R_\theta$ circles parallel to the XY-plane by a constant angle $\Delta\theta$ and each circle is in turn discretized into $2 \times R_\varphi$ grids by a constant angle $\Delta\varphi$ (see Fig. 2.4a). Instead of projecting patches onto the hemisphere, rays are cast from O through the center of their corresponding delta areas (the shaded rectangle in Fig. 2.4b), and denoted as r_{ij}, $i = 0, 1, .., R_\varphi - 1, j = 0, 1, .., R_\theta - 1$. There are thus $R_\varphi \times R_\theta$ rays uniformly distributed over (φ, θ)-plane as shown in Fig. 2.4b. The delta form-factor of each delta area is assigned to the corresponding ray. A delta area is said to be covered by the projection of a patch if the corresponding ray intersects the patch. The computation of the form-factor then proceeds as usual in the hemisphere method. Henceforth, we shall call the above procedure a *ray-casting procedure*. Notice that the core computations involved in the ray-casting procedure will be the ray-patch intersections.

The advantages of the ray-casting based approach are:

1. It is capable of intersecting rays with curved surfaces such as Bezier. This allows users to use the original geometric representation of surfaces without polygonization.
2. It provides a means for more flexible sampling to avoid aliasing due to uniform sampling.

The resolution and distribution of rays can be adapted to surface radiative properties and geometric relationship between sources and receivers.

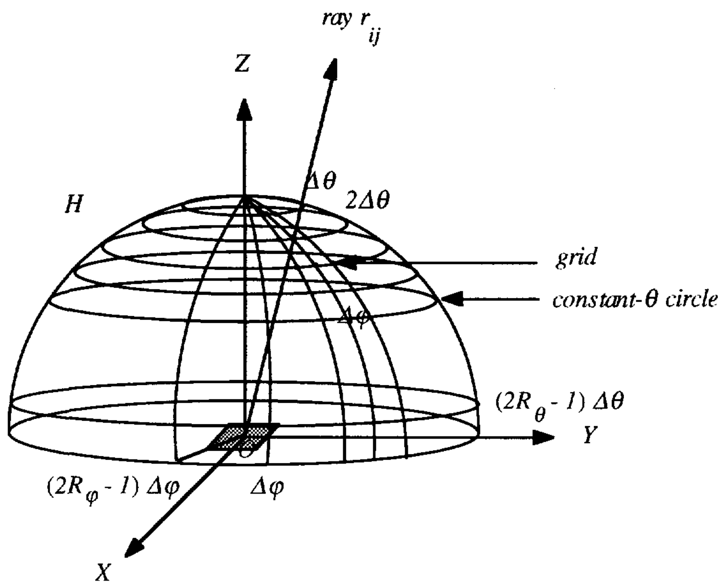
In the following, sampling concerns will be examined from two different perspectives:

1. Source Sampling

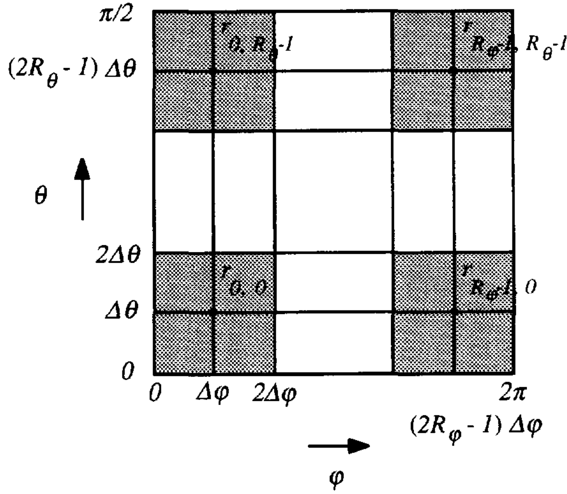
   In the previously discussion, only one sample point is taken at the center of the source patch over which a hemisphere or hemi-cube is placed. This amounts to reducing that source to a point. This may be quite erroneous, for instance, when the area of the source is quite large compared to the distance separating the patches (see Fig. 2.5a) or when the source is partially occluded by other patches (see Fig. 2.5b). To approximate an area source, several sample points must be taken from the source.

2. Receiver Sampling

   Being oblivious to the geometry of a scene, it may be appropriate to concentrate on those directions which are important and pay less attention to those directions which are less important. By casting more rays in important directions, the uniform sampling proposed above actually addresses this directional importance. The problem of this approach is that a small patch can be easily missed when it falls to the less important directions. To avoid this quite obvious artifact, more rays must be cast for sampling small patches.

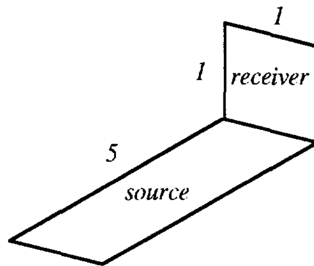


(a) On the hemisphere

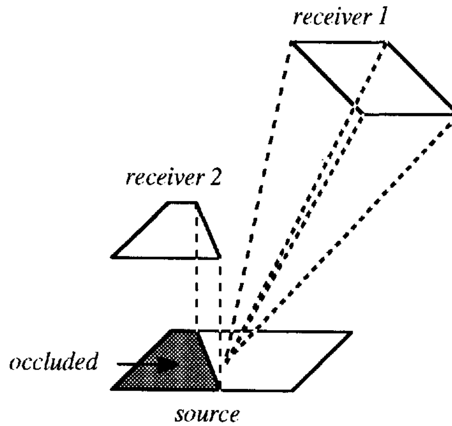


(b) In the (φ, θ)-plane

Figure 2.4: Uniformly distributed rays for a ray-casting based approach.



(a) large elongated source to smaller receiver



(b) *receiver* 2 missed due to source undersampling

Figure 2.5: Source sampling problems.

Recently, considerable effort has been devoted to the sampling problem. These include:

1. Wallace [WEH89] claimed that the number of sample points to approximate an area source depends on how close the source is to the receiver. In other words, the number of sample points required may vary from one receiver to another receiver. However, in the hemi-cube (or hemisphere) method, the number of sample points is fixed for all the receiving patches. In addition, the radiosities at the element vertices, which are used to actually render the final image, cannot be determined directly. The radiosity of each vertex must be determined by averaging the radiosities of the elements surrounding it. In view of this, he proposed a receiver to source sampling. Instead of performing a hemi-cube (or hemisphere) at a source, each element vertex in the scene is visited and form-factor is computed from the source to the vertex by shooting rays from the vertex to sample points on the source. The number of sample points on the source may vary from one vertex to the next, thus allowing area sources to be approximated as accurately as desired. Furthermore, this eliminates the sampling problem of small patches, since illumination is guaranteed to be computed at every vertex. The main problem of this approach are: (1) All the vertices must be retrieved from the large database for each selected source, representing an undesirable bottleneck (i.e., long latency) for parallel processing. (2) No efficient space partition technique can be used to accelerate the visibility testing. The shaft culling proposed in [HW91] still suffers from some problems such as the saving in the visibility testing would not trade for the cost in testing surfaces lying inside the shaft, and a shaft possibly contains almost the entire scene. (3) Rays are unstructured in some sense, which is difficult to be handled (retrieved

or generated) by hardware. For these reasons, this software oriented approach is not amenable to hardware implementation. Most seriously, the parallelism in the algorithm becomes insignificant due to the inherent long latencies.

2. Hanarahan [HSA91] pointed out that form-factors are prone to error, and therefore need only be computed and stored if the error is within a specified tolerance. He proposed a recursive refinement procedure which simultaneously decomposes a polygon into a hierarchy of patches and elements, and builds a hierarchical representation of the form-factor matrix by recording interactions at different levels of detail. It first estimates the mutual form-factors between two patches, and then either subdivides the patches and refines further, or terminates the recursion and records an interaction between the two patches. If either of the form-factor estimates is larger than a specified value F_ε, the patch with the larger form-factor undergoes a further subdivision Otherwise, the patches are allowed to interact at this level of detail, and the true form-factor can be approximated accurately by the estimate. Visibility only needs to be calculated as accurately as required for form-factors within the error tolerance, and roughly the same amount of work is done at each visibility test. The main problem of this approach are: (1) This approach works best for scenes with few large initial polygons. For scenes with many small initial polygons, it is wasteful to determine the level of interaction for each pair of polygons because many of them may be totally occluded by intervening polygons. (2) The pairwise method for computing form-factors works well for low-frequency shading intensities (minor intensity transitions) but not for high-frequency shading intensities (clearly visible shadow boundaries). To capture clearly visible shadow boundaries, the interaction between two patches will be driven to an unacceptable low level. (3) The visibility test fires unstructured rays between each pair of patches. As before, no efficient space partition technique can be used to accelerate the visibility testing and unstructured rays are generally difficult to be handled by hardware.

To conclude the discussion on the ray-casting based approach, the following general remarks can be made:

1. It provides a means for more flexible sampling. Source sampling can be approximated by placing several sample points on the source. Receiver sampling may be satisfied by casting rays following the directional importance. Furthermore, the artifacts due to undersampling of small patches can be solved by casting ultrahigh-resolution rays together with jittering the ray directions. In [KJ91] [KJ92] [KJW93] [Shi90], it is pointed out that it is nearly impossible to capture clearly shadow boundaries by computing patch-to-element or element-to-patch form-factors as in the radiosity method. Instead, a two-pass approach is proposed in which the radiosity pass only accounts for the indirect lighting. The patch refinement in this pass can be relatively coarse because the radiosity shading only shows minor intensity variations. From this point of view, we believe that the ray-casting based approach is sufficient for providing an accurate indirect interreflection.

2. Although this approach does not lead to optimal solution, it is very amenable to hardware implementation. Well-known space partitioning techniques [Cla76] [FTI86] [Gla84] [MHI88] [RW80] [TKM84] [WHG84] exist for accelerating the visibility testing. In addition, this approach lends itself to parallel processing, as will be addressed in chapter 3.

2.2.2. Radiosity Solutions

Given the reflectivity of each diffuse reflector and the emission of each light emitter, the radiosity of each patch can be obtained by solving the radiosity equation as given in Eq. 2.3 if the form-factor between each pair of patches is known. There are two restrictions that prohibit the use of standard solution techniques for linear systems. Firstly, we cannot compute and store the complete form-factor matrix for a typical scene containing thousands of patches, since the required storage may be prohibitively large. Secondly, we aim at a fast approximation instead of an exact solution to the radiosity problem. In the following, we shall discuss the mainstream of radiosity solutions.

2.2.2.1. Gauss-Seidel Iteration

In the conventional radiosity technique, the Gauss-Seidel method is used to solve the system of equations given in Eq. 2.3 one row at a time. The evaluation of ith row of the equations provides an estimate of the radiosity of patch i based on the current estimates of the radiosities of all other patches:

$$B_i = E_i + \rho_i \sum_{j=1}^{N} B_j F_{ij}. \tag{2.11}$$

This amounts to *gathering* the light from the rest of the environment. Due to the strict diagonal dominance of the form-factor matrix, the solution converges in a few iterations. However, you cannot display the radiosity of all patches until after the first complete iteration cycle and all the form-factors are pre-calculated and stored at the cost of $O(N^2)$, N being the number of patches.

2.2.2.2. Progressive Refinement

By virtue of Eq. 2.1, the contribution of the radiosity from patch i to the radiosity of patch j is given by:

$$B_j \text{ due to } B_i = \rho_j B_i F_{ij} A_i / A_j. \tag{2.12}$$

Thus we may solve the radiosity equation one column at a time by determining the contribution made by patch i to the radiosity of all other patches. This amounts to *shooting* light out from patch i into the environment as opposed to the gathering in the Gauss-Seidel iteration. It has the

advantage of performing a single hemisphere or hemi-cube over a patch (called the source) and adding the contribution from the radiosity of the source to the radiosity of all other patches. Hence the storage cost is at the order of $O(N)$ per iteration. By selecting different patches to be the source, the radiosity of other patches can be incrementally updated until a sufficient number of patches have shot their energy. This approach differs from the previous one primarily in two respects. First, the radiosity of all patches is updated simultaneously, providing intermediate results which can be displayed as the algorithm proceeds. Second, patches are selected as sources according to their energy contribution (unshot radiosity) to the environment, providing accurate results early in the solution process.

2.2.2.3. Overshooting Algorithms

In the shooting algorithm, whenever radiosity is shot from a source, this will increase the unshot radiosity of other patches. Those patches will be in turn selected for shooting later, and possibly a part of this radiosity will be sent back to the original patch. The idea of overshooting algorithms is to take contribution coming from later steps into account and to shoot more radiosity than is currently available on a source.

Generally, overshooting algorithms give a better convergence than the standard algorithms without overshooting. One example is successive over-relaxation which is a variant of Gauss-Seidel iteration. Instead of gathering a correct amount of radiosity, a factor α is added to take contribution coming from later steps into account. This algorithm is similar to the Gauss-Seidel iteration except Eq. 2.11 is modified to be:

$$B_i = (1 - \alpha) B_i + \alpha (E_i + \rho_i \sum_{j=1}^{N} B_j F_{ij}), \tag{2.13}$$

where α takes the value of 1.2 - 1.5.

Another example is called ambient overshooting [FP92], in which the ambient term introduced by Cohen [CCWG88] is used as an estimate for the additional amount of overshooting. More precisely, for each selected source, the radiosity $\Delta B_i + \rho_i$ *ambient* is shot instead of ΔB_i.

In [GHS92], Greiner made a comparison for the convergence of different radiosity solutions. The following results are excerpted from there:

1. Both the Gauss-Seidel iteration and successive over-relaxation work quite well in the long term but they show very bad initial results, which makes them insuitable for interactive rendering.

2. On the contrary, the progressive-refinement algorithm and ambient overshooting perform very well in the first few steps but the convergence becomes a bit slow later on. The progressive-refinement algorithm even performs better in the early stage of the iteration. This fact, together with the advantage of reducing storage cost have made the progressive-

refinement algorithm very appealing to interactive rendering.

2.3. Ray Tracing

Ray tracing is a simple but powerful algorithm that can naturally simulate complex lighting effects such as reflection, refraction and shadowing. To clarify this, it is helpful to show which paths of light reflection have been accounted for. As shown in Fig. 2.6, recursive ray tracing [Whi80] takes into account the direct diffuse (path *l-c-v*) and directly specular reflections (path *l-g-v*), as well as the indirect specular reflections (path *l-a-h-v*). However, it does not take into account the contribution of light that is specularly reflected by mirroring surfaces onto diffuse surfaces (path *l-d-f-v*). To capture this light, Arvo [Arv86] proposed a preprocessing pass called backward ray tracing, in which rays are shot from the light sources and the mirrored reflection is stored in illumination maps on the surfaces. The second pass then meets the first pass at those surfaces (at *f* in Fig. 2.6). The other type of reflection that has been neglected is the diffuse interreflection between surfaces (path *l-e-b-v*).

In the traditional ray-tracing algorithm, the global illumination is approximated by tracing rays from the viewpoint, through each pixel of the display screen, and into the environment. When a ray hits a surface, two things may happen:

1. If the surface is purely diffuse, then the light at that intersection point on the surface will be taken as the intensity of that pixel. This is done by casting a shadow ray from the intersection point to each light source. We then determine whether the surface will be illuminated by a light source or not in the following way. If the shadow ray can reach the light source without hitting any surface along the way, then the light source can illuminate the surface. Otherwise, the surface is in shadow relative to that light source.
2. If the surface is purely specular, then a reflected/refracted ray is generated at the intersection point. Again a shadow ray is cast from the intersection point to each light source to determine the light coming from each light source. The reflected/refracted ray is in turn traced to see if anything gives off light. This backward tracing is repeated until it ends up a purely diffuse surface, rays leave the environment or further contributions can be neglected. All these intensities carried with reflected/refracted rays will be added to give the final pixel intensity.

In the above discussion, it is assumed that one shadow ray is cast from an intersection point to each light source and also one reflected/refracted ray is generated for an intersection point on a purely specular surface. As a result, ray traced images will have sharp reflections, sharp refractions and sharp shadows. By distributing the directions of rays according to the analytic function they sample, ray tracing can also incorporate fuzzy phenomena (see [CPC84]).

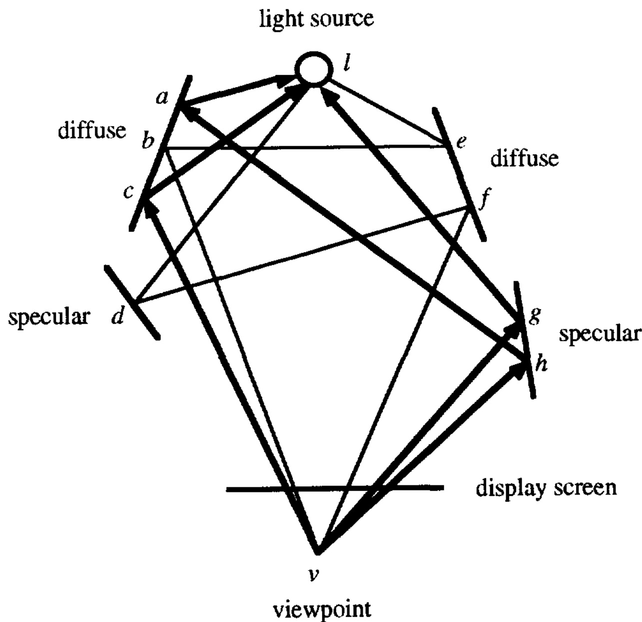


Figure 2.6: Different paths of light reflection.

2.4. Two-Pass Approach

As stated previously, the radiosity method is most suited for the display of diffuse environments, while ray tracing is most suited for the display of specular environments. Recently, several attempts have been made to integrate radiosity and ray tracing to render a complex environment. They either extend the conventional ray-tracing algorithm to include diffuse interreflection [Kaj86] [WRC88] or extend the conventional radiosity method to add in a specular component [WCG87] [SP89] [Hec90]. Our emphasis will be on the second approach on which the proposed two-pass approach is based. While using ray tracing for the specular reflection, the traditional radiosity method is used to compute the radiosity shading including lighting received both directly or indirectly from the light sources. One major shortcoming of this approach is that high radiosity gradients over patches, in particular clearly visible shadow boundaries, cannot be efficiently and economically solved. Methods in [CGIB86] [CF90] [Arv86] [Hec90] either require a large data structure or fail to provide the dynamic adaptation needed.

We observed that high radiosity gradients over patches are mainly due to the lighting received directly from the main light sources and most radiant patches. If we neglect the so-defined direct lighting in the radiosity pass, the radiosity shading may only show minor intensity variations. Thus, the patch refinement can be kept relatively coarse and solved efficiently and economically. In this way, the first pass only calculates the indirect lighting for each diffuse patch and is done by the conventional radiosity method, and the direct lighting is then calculated at the second pass by casting shadow rays as in the traditional ray tracing. In this approach, there are two crucial issues to be explored. The first, is to select light sources which will contribute to the direct lighting. The other, is to reduce the number of shadow rays. A detailed description of these issues is beyond the scope of this thesis. We can mention however that a source selection algorithm has been developed that can select potential sources globally or locally as per patch, and the number of shadow rays can be further reduced by applying an adaptive image refinement technique in combination with a shadow coherence method. For additional details we refer to [KJ91] [KJ92].

2.5. A Proximity Enforced Algorithm

For a ray-casting based approach, it requires a place large enough to store the complete database of a scene. A hierarchical memory system is generally assumed to meet this requirement. This hierarchy, which is used in almost all computer systems today, reflects one of computer design's truisms, "fast memory is expensive but is very limited in capacity, and slow memory is cheaper but can be quite large in capacity". Owing to the limited bandwidth of the memory system and the relatively low transmission rates in communication links, it is necessary to transmit only those data which are relevant to the execution of the program, store them in the fast memory and keep them stationary as much time as possible. This can be accomplished by the following strategy: (1) First of all, we should store most referenced data in the fast memory, while store less referenced data in the slow memory. (2) Secondly, the execution of the program that implements the ray-casing based approach should be enforced so that the data stored in the fast memory allow to be referenced as much time as possible. This leads to a so-called proximity enforced algorithm. This is based on the observation that the half spaces seen through sample points on the same patch or on the neighbouring patches with similar orientations will be very much alike. For clarify, we illustrate with a two-dimensional example in Fig. 2.7. The half spaces seen through sample points p_1, p_2 and p_3 are indeed very similar. For practical usage, three hemispheres may be performed at those sample points during the same progressive refinement step. The relevant data required by the first hemisphere can be stored in the fast memory, and most of them will be referenced by the second and third hemispheres due to the data-coherence property. Consequently, we can get rid of the long latencies of retrieving data from the slow memory via the slow communication links. In chapter 5, we will use *effectiveness* as a measure to see how well the so-called data coherence can be

exploited.

From the above discussion, we then modify the progressive-refinement algorithm. During one progressive refinement step, several patches instead of one single patch are selected as sources based on the criteria of unshot radiosity and *proximity*. The proximity criterion is added to enforce sample points to be kept close by so that data coherence can be maximally exploited. For instance, we can take advantage of the data structure (octree [Gla84] [MM83], binary tree [KM85], or a spatially enumerated auxiliary data structure [MHS86] [TI86]) storing patches together with patches' normal vectors to determine the proximity.

We are now investigating the convergence of this proximity enforced progressive-refinement algorithm. In the literature, there is a similar approach called the blockwise refinement [GHS92]. The difference is that the criterion of selecting patches is only unshot radiosity as in the traditional progressive-refinement algorithm. The results show that the blockwise refinement performs better than the traditional progressive-refinement algorithm during the initial stage of the iteration. We believe that the convergence of our approach will be similar to the blockwise refinement because patches in proximity to each other often receive similar radiosity. From this point of view, most probably they will be selected to be sources as in the block refinement.

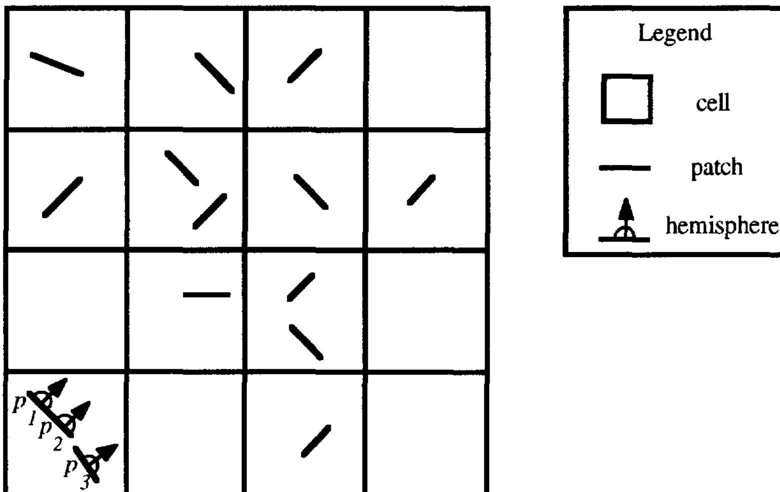


Figure 2.7: The half spaces seen through three sample points.

The same situation happens at the ray tracing pass. For intersection points on the same patch or on the neighbouring patches with similar orientations, the local environment seen by shadow rays (for shadow ray casting) and/or reflection/transmission rays (for reflection/transmission

ray casting), will be very much alike (see Fig. 2.8). This motivated us to use a patch-based ordering for shooting. By following the scan-line ordering, intersection points pertaining to the same patch can be grouped together for shadow ray casting and/or reflection/transmission ray casting. In this way, the relevant data required by an intersection point (to start with a shadow ray casting and/or a reflection/transmission ray casting) can be stored in the fast memory, and most of them will be referenced by other intersection points in the same group due to the data-coherence property.

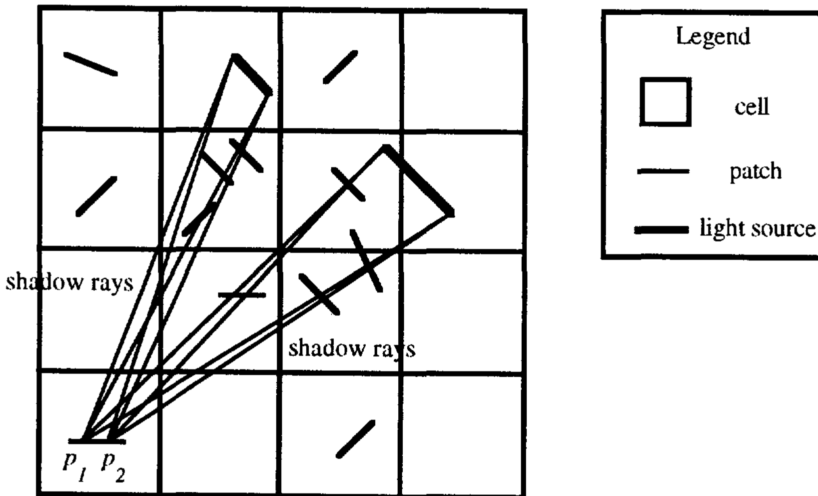


Figure 2.8: The local environment seen through two intersection points.

2.6. Concluding Remarks

In this chapter, we have presented a ray-casting based two-pass approach that serves as the basis for our target algorithm. A proximity enforced algorithm is then suggested by taking advantage of the data-coherence property at algorithmic level. In the radiosity pass, the most time-consuming computation is the form-factor computation. As for the ray-tracing pass, shadow ray casting and/or reflection/transmission ray casting would dominate in computation. They all rely on the ray-casting based approach. In this sense, any program that implements the two-pass approach will consist of a core program which performs the ray-casting procedure. From chapter 3 onwards, emphasis will be put on this ray-casting procedure.

Chapter 3

A New Space Partitioning Technique

3.1. Introduction

In chapter 1, we have argued that the minimum execution time of a parallel algorithm as a function of task size depends on both the topology of the architecture and the structure of the algorithm. If neither an architecture nor an algorithm is pre-specified in greater detail, then one will have to start-off from some initial algorithm-architecture pair and rely on some design methodology to carry it over into alternative pairs which supposedly approach gradually the optimum pair. The choice of an initial algorithm-architecture pair is crucial as it may otherwise not be easy to reach the optimum pair. In a general setting, the search for an optimum will most likely start-off from some sequential program which results from the designing of an algorithm that has been carried out on a standard workstation. However, in cases when the underlying problem resembles in some way or another problems from scientific computing, it will most probably be possible to cast the sequential program in the form of regular nested-loop program (massively parallel algorithm). Although such initial programs may be an oversimplification and wasteful in terms of operation counts, they have the advantage that they are easy to parallelize and naturally suggest a (almost) linear speed-up architecture. In other words, a systolic-like algorithm is, then, a good initialization. Consider a ray-casting procedure. We have chosen the initial algorithm-architecture pair to ly on the ideal linear speed-up line. We shall call this pair the naive pair.

As stated previously, the core computations involved in the ray-casting procedure is the ray-

patch intersections. Let p_i, $i = 0, 1, .., N - 1$, be N patches in a scene, and let r_{jk} $j = 0, 1, .., R_\varphi - 1, k = 0, 1, .., R_\theta - 1$, be $R = R_\varphi \times R_\theta$ rays uniformly distributed over (φ, θ)-plane (see Fig. 2.4b). The naive algorithm is given in Fig. 3.1.

Obviously, the sequential or single-processor execution of this program will take time $O(R \times N)$. On the other hand, a parallel or multi-processor execution of this program would be achievable in time almost zero, provided the number of processors is $R \times N$ and communication overhead is disregarded. But as the speedup-overhead tradeoff given in Fig. 1.1 predicts, this massively parallel algorithm will also have a large execution time as will be obvious from its tremendous amount of intrinsic communication. Nevertheless, the naive algorithm is a very useful algorithm to start-off the design of a realistic well-performing architecture because it is completely transparent in the sense that all possible parallelism as well as all possible overhead is explicit in it. As will become clear in this chapter and chapter 4, this property makes the task of reducing overhead while preserving as much as possible of the inherent parallelism much easier. In the remainder of this chapter we will analysis the naive algorithm, and discuss a conventional and a new space partitioning techniques.

Algorithm: *Naive*

for all patches p_i , $i = 0, 1, .., N - 1$

for all angles $\varphi_j, j = 0, 1, .., R_\varphi - 1$

for all angles $\theta_k, k = 0, 1, .., R_\theta - 1$

compute intersection(p_i, r_{jk})

end for

end for

end for

Figure 3.1: The naive algorithm.

3.2. A Naive Algorithm

The naive algorithm can also be cast in the form of a nested loop program (*NLP*). A particular form of which is shown in Fig. 3.2, where i_l, $l = 1, 2, ..., n$, are referred to as loop indices, and $f_{l,j}, f_{u,j}$ are referred to as integer-valued boundary functions, possibly involving loop indices. The constants m_l are increment or decrement steps of the loop indices. C_i , $i = 1, 2,$

..., k, are conditions on the loop indices. B_i , $i = 1, 2, ..., k$, are program bodies, which are either ordered sets of assignment statements or *NLPs* themselves. The collection of all if-then-else blocks is called the loop-nest body. If the loop indices are stacked in a column vector $I = [i_1, i_1, ..., i_n]^t$, then I is called an *iteration vector*. Here, the superscript ^{t} denotes transpose. Since the intended execution of an *NLP* is serial, that is, the iterations of loops will be executed one by one, in the order imposed by the loop indices, a mental execution of such a program is necessary to expose the intra-iteration dependencies which are more or less hiding the implicit parallelism in the program. In [DHW93], a methodology is proposed to map nested loop programs into parallel processor arrays for the case when f_l, the f_u and the C are affine functions. This methodology starts off with a conversion of an affine nested loop program to a single assignment program (*SAP*) which is a graphical representation of the program static *computational structure*.

```
for i_1 = f_{l,1} to f_{u,1}, step m_1
    for i_2 = f_{l,2} to f_{u,2}, step m_2
        •
        for i_n = f_{l,n} to f_{u,n}, step m_n
            if C_1 then B_1
            else if . . .
            •
            else if C_k then B_k
            end if
            •
            end if
        end for
        •
    end for
end for
```

Figure 3.2: A general *NLP*.

Definition 3.1 (Computational Structure, Dependence Vector) Let P be an SAP derived from an NLP as given in Fig. 3.2. The *computational structure* of P is an indexed directed graph $Q(N, E)$ defined in the n-dimensional iteration domain $\mathbf{I} = \{I \mid I = [i_1, i_1, ..., i_n]^t, f_{l,1} \le i_1 \le f_{u,1}, ..., f_{l,n} \le i_n \le f_{u,n}\}$. Each node $n_I \in N$ represents an iteration and is located at the point corresponding to the iteration vector $I = [i_1, i_2, ..., i_n]^t$. There is an edge $e_{IJ} \in E$ between two nodes n_I and n_J if node n_J requires a data from node n_I. The edges are labelled with a vector which is called a *dependence vector*. The edge e_{IJ} is labelled with the dependence vector $d_{IJ} = J - I$. ∎

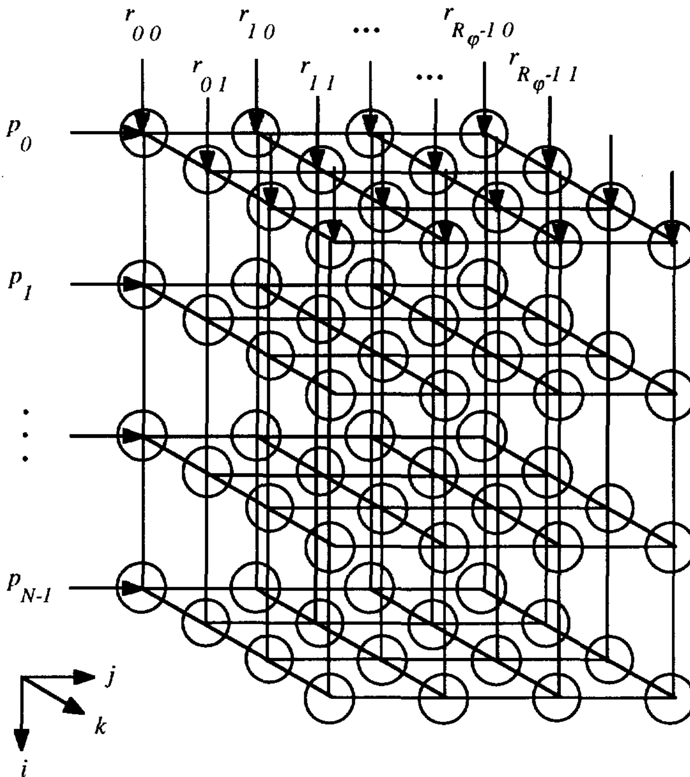


Figure 3.3: The computational structure Q for the naive algorithm.

If the number of iterations for each loop in an NLP is known at compile time, which is generally called a *manifest iteration*, then it is possible to derive the computational structure before program execution. This is essential to the applicability of a compile-time scheduling relying on graph-theoretic method. Now consider the naive algorithm. All the boundary functions are constant parameters which are known at compile time. The

computational structure of the naive algorithm can thus be easily drawn as shown in Fig. 3.3. A patch can be delivered to all the nodes on a (*j*, *k*)-plane by way of broadcasting. Similarly, a ray can be delivered to the nodes where they are used. It is thus seen that nodes in the graph of Fig. 3.3 do not depend on each other. Being free from data dependencies, each node in Fig. 3.3 can be assigned to a separate processor to perform an intersection computation whenever patch and ray data become available. As a result, the naive algorithm belongs to a class of massively parallel algorithms, which lends itself to a massively parallel implementation. However, they turn out to be rather inefficient in terms of utilization of resources.

3.3. The Conventional Technique

The computational structure in Fig. 3.3 results in many wasteful computations. This is due to the fact that the sets of input patches and rays are not ordered from the viewpoint of a sample point. These wasteful computations can be avoided by ordering the input sets of patches and rays in some way. In what we shall call the *conventional technique*, the 3-dimensional object space is partitioned into cells according to a particular encoding scheme, e.g., octree [Gla84] [MM83], binary tree [KM85] or a spatially enumerated auxiliary data structure [MHS86] [FTI86]. Each cell stores patch data of those patches which are intersected with the cell. In this way, patches are localized and ordered in the same way as the cells are enumerated.

Algorithm: *Serial*

for $i := 0$ **to** $R_{\varphi}-1$ **do**

  for $j := 0$ to $R_{\theta}-1$ do

    hit := *false*;

    while (*hit* = *false*) **do**

    begin

      cell := *cell traversal*(*ray*);

      for each *patch* in *cell* **do**

        hit := *compute intersection*(*ray*, *patch*)

    end;

Figure 3.4: The conventional technique.

The pseudo-code for the *Serial* algorithm that implements the conventional technique is given in Fig. 3.4. One ray is shot and tested against all patches in the first cell. If the ray hits some patches in this cell, distances are compared to determine the nearest intersection point. If there is no hit, then the next cell is traversed and tested again until there is a hit. As you can see, the outer loop in the naive algorithm that iterates through all the patches has now become the inner loop. Due to the test condition in the **while**-loop, the inner loop only iterates through those patches stored in the cells opened during the lifetime of a ray. Some saving in intersection computations can be made at the expense of cell traversals. One major problem of this approach which is common to any space partitioning technique is the inclusion of *data-dependent iterations*, as is explained below. In data-dependent iterations, the number of iterations is determined at run time and cannot be known at compile time. This makes the scheduling task cumbersome.

Let t_i be the minimum intersection distance of a ray found at cell i. We have the following formula for the distance to the intersection point, t_{min}:

$$t_{min} = min(t_1, t_2, t_3, ...).$$

The *min*(.) takes the minimum of its arguments. Due to the structure of the cells, we can state the following for the minimum intersection distance t_i within a cell i.

$$t_i > R_{i-1}$$

where R_{i-1} is the largest intersected distance of the previous cell, i.e., cell i-1, with the ray.

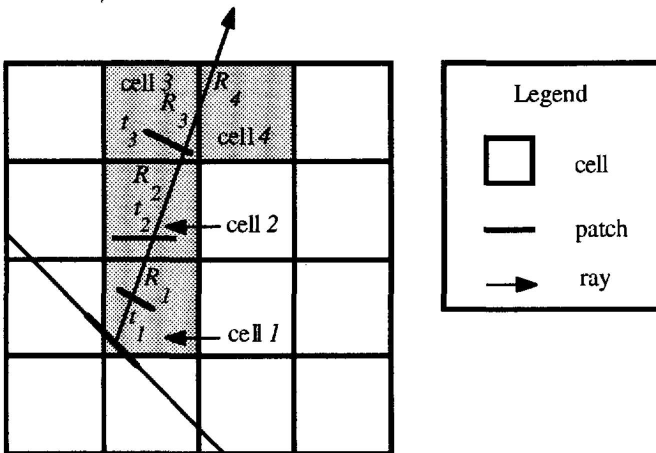


Figure 3.5: Cellwise space searching in the conventional technique.

We now introduce the concept of *lazy evaluation* for the minimum distance computation. The term lazy evaluation stems from the practice found in many programming languages when evaluating (Boolean) expressions. Take as an example the evaluation of the expression (a or b), where a and b may be complex expressions themselves. After evaluating a, if a is found to be *true*, then b no longer needs to be evaluated. This saves considerable time if the evaluation of b takes longer than the premature test on a. Hence we would like to defer the evaluation of b until the result of a is known. This will introduce a *control dependency* rather than a *data dependency*. A lazy evaluation equivalent of the **while**-loop in the algorithm of Fig. 3.4 is given in Fig. 3.6 hereafter.

In the conventional technique, the cellwise space searching in the calculation of intersection points as can thus be stated as a lazy evaluation by unfolding the **while**-loop in Fig. 3.4. Observe that there is data-dependent iterations in the **for**-loops enclosed in the square braces is not known at compile time. Also not known at compile time is the number of iterations in the **if-then-else** nest, which is another type of data-dependent iteration. Due to those data-dependent iterations, it is not possible to draw the computational structure before program execution.

Other shortcomings in this approach are:

1. Obviously, small cell sizes will result in fewer intersection computations per cell. This is because a ray needs to be tested against all the patches stored in a cell opened by the ray. However, although the number of intersection computations can be reduced by using small-sized cells, this will bring about an increasing number of cell traversals and vice versa. Therefore, it is very hard to balance workloads representing cell traversal and intersection computation due to the conflict of reducing one computation together with the other.

2. A patch might have to be retrieved many times from memory because neighbouring rays are most likely to traverse the same cells and even hit the same patches. This results in a lot of waste in terms of communication.

```
t_1 := infinity;

[ cell := cell traversal(ray);
[ for each patch in cell do
[        t_1 := compute intersection(ray, patch);

if t_1 ≤ R_1 then t_min := t_1
else
        begin
                [ cell := cell traversal(ray);
                [ for each patch in cell do
                [        t_2 := compute intersection(ray, patch);
                t_2' := min(t_1, t_2);
                if t_2' ≤ R_2 then t_min := t_2'
                else
                        .
                        .
                        .
        end;
```

Figure 3.6: A lazy evaluation equivalent of the **while**-loop in the *Serial* algorithm.

3.4. The Shelling Technique

To overcome the shortcomings of the conventional technique, a new space partitioning technique which we call the *shelling technique* [SDP90] [SLD91] [SDP91*b*] is proposed. This technique relies on a visibility ordering with respect to a sample point. The basic ideal is to opt for a *view-dependent* partitioning of the object space, as illustrated in Fig. 3.7. Compared to the conventional technique, patches are now confined to partitions instead of cells. As a consequence, a ray now only tests against patches residing in the partition where the ray lies.

As a first step, the hemisphere above the patch on which the current sample point lies, and which we call the source patch, is partitioned into wedge-type partitions. This space partitioning is clearly reflected into the resulting computational structure which is shown in Fig. 3.9. The blocks of circles correspond to partitions and the circles within a block correspond to ray-patch intersection computations within a partition. Unprimed, primed and double primed patch labels refer to the same physical patch which extends over a single, two or three partitions, respectively. There is obviously an improvement in operation counts compared to the naive algorithm, which would correspond to a single large block of intersection computations. Nevertheless, the small blocks in Fig. 3.9 have still an all-ray-to-all-patch relationship structure as in the naive algorithm. This is again due to the fact that patches residing in a single partition are not ordered by following the visibility ordering. There is thus still a waste of computations within each partition, since there is no visibility (depth) information whatsoever. In passing, we notice that if the number of partitions is maximized, then we are back to the conventional algorithm, whereas minimizing that number leads us to the naive algorithm. We will come later to the question as to what might be an optimal number of partitions. For the time being it is sufficient to notice that the partitioning introduced so far does not introduce any dependencies. We now proceed with resolving the question of how to reduce the waste in computations within a partition. Recall that this waste is due to a lack of depth information. So as a second step, we propose to provide such information by introducing a second partition which is along radial direction. Indeed, the northern hemisphere which is local to the source patch (on which lies the sample point relative to which the former partitioning was defined) is partitioned into concentric half spheres which define *shells* of uniform radius. This is shown in Fig. 3.8. The objective of this partitioning is as follows. A ray which reaches the spherical boundary of a shell is declared to have survived if it has not hit any patch it encountered on its cell traversing path. Otherwise, it is declared dead. Only survived rays can continue cell traversing beyond this shell. The computational structure going with this partitioning as well is shown in Fig. 3.10. In contrast to the azimuthal partitioning, radial partitioning does introduce dependencies.

On the other hand, the introduction of depth information will reduce the wasteful computations to a large extent. Yet within the space bounded by the two spherical boundaries of two consecutive shells and the two edges of the wedge partition, which we will call temporarily a

subshell, a subset of rays travelling through this space may not hit some or even all of the patches that is confined to its traversing path. Therefore, it would even save more computations if it would be possible to know in advance that a particular ray will not hit a particular patch in a subshell. To explore the possibility of saving computations in this sense, we go into a further refinement of visibility information by introducing a bounding area at the level of patches.

Thus, as a third step, we propose to view a patch as being enclosed in what we shall name a *Spherical Bounding Box* (*SBB*). Clearly, rays that are outside an *SBB* of a patch will not intersect this patch, and therefore ray-patch intersection computations for such rays do not have to take place. This bring about a nearly ideal computational structure as shown in Fig. 3.11. For clarity, the ideal computational structure is denoted by shaded nodes there. The unshaded part that remains wasteful is mainly due to the fact that it is difficult to isolate two patches close to each other in the depth direction by the shellwise searching. Theoretically, they can be isolated by using an infinitesimal shell radius. However, the depth complexity of a scene is in general low as pointed out in [CH92]. It is not worthy to isolate a few more patches at the expense of many more cell traversals.

In the above discussion, we attempted to explain the ideas behind the shelling technique by considering their main effects on the computational structure of the naive algorithm. However, as already alluded to, some dependencies should have come into the computational structures shown in Fig. 3.10 and Fig. 3.11 which is due to the fact that a ray's behavior in one shell will depend on the computational results of that same ray in previous shells. We will come to this point in a moment.

To conclude, we claim that the shelling technique can remedy the shortcomings of the conventional technique. This is because

1. Rather than relying on small-sized cells to screen out unnecessary patches, the *SBB* of a patch is used to screen out unnecessary rays. Regardless of the size of a cell, only rays within an *SBB* are used for computing intersection points. As opposed to the conventional technique, a median-sized cell suffices for our purpose. Moreover, a grid-ray cell traversal based on grids defined on the basis of medium-sized cells was proposed that can reduce the number of cell traversals considerably. It seems likely that workloads related to cell traversal and intersection computation can be balanced to some extent.
2. In the shelling technique, the retrieval of one patch allows for computing intersection points with a bundle of rays. One may assign a separate processor to each ray. The difficulty is that the number of rays allowed for intersection computations may change from patch to patch. This prompts a dynamic architecture that allows this run-time changing parallelism to be exploited. Pipeline processing seems more appropriate for this purpose because it can handle the changing in the number of rays in a natural way. Moreover, the pipelinability in computations can be better exploited in the shelling technique than in the conventional technique. This is because a ray can be retrieved much faster than a patch, allowing an Intersection Computation Unit (*ICU*) to perform at a higher rate.

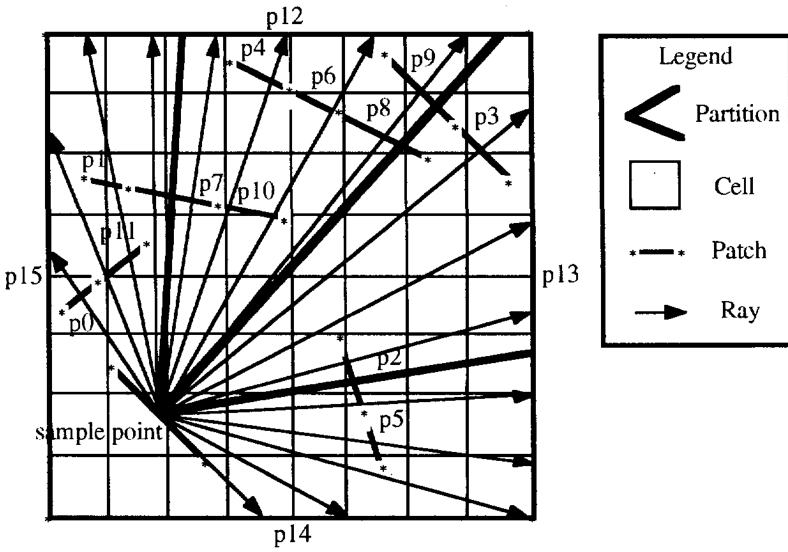


Figure 3.7: View-dependent partitioning of the object space.

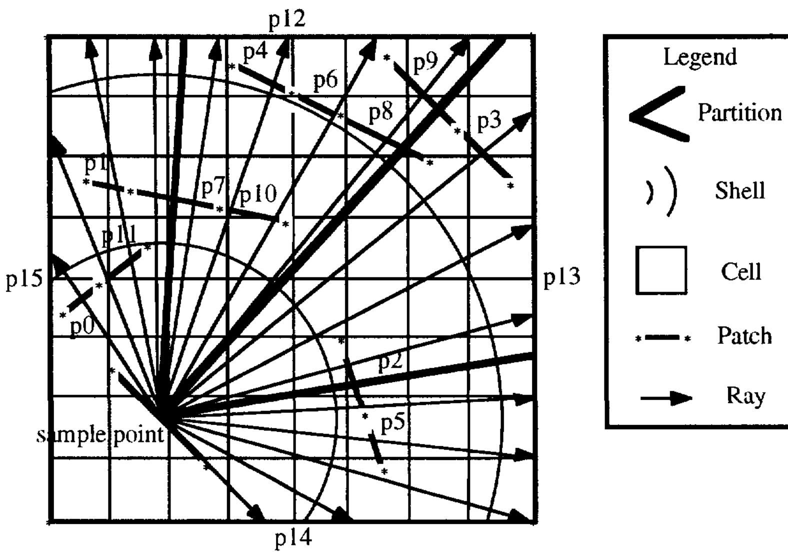


Figure 3.8: The combined use of shellwise searching and the view-dependent partitioning.

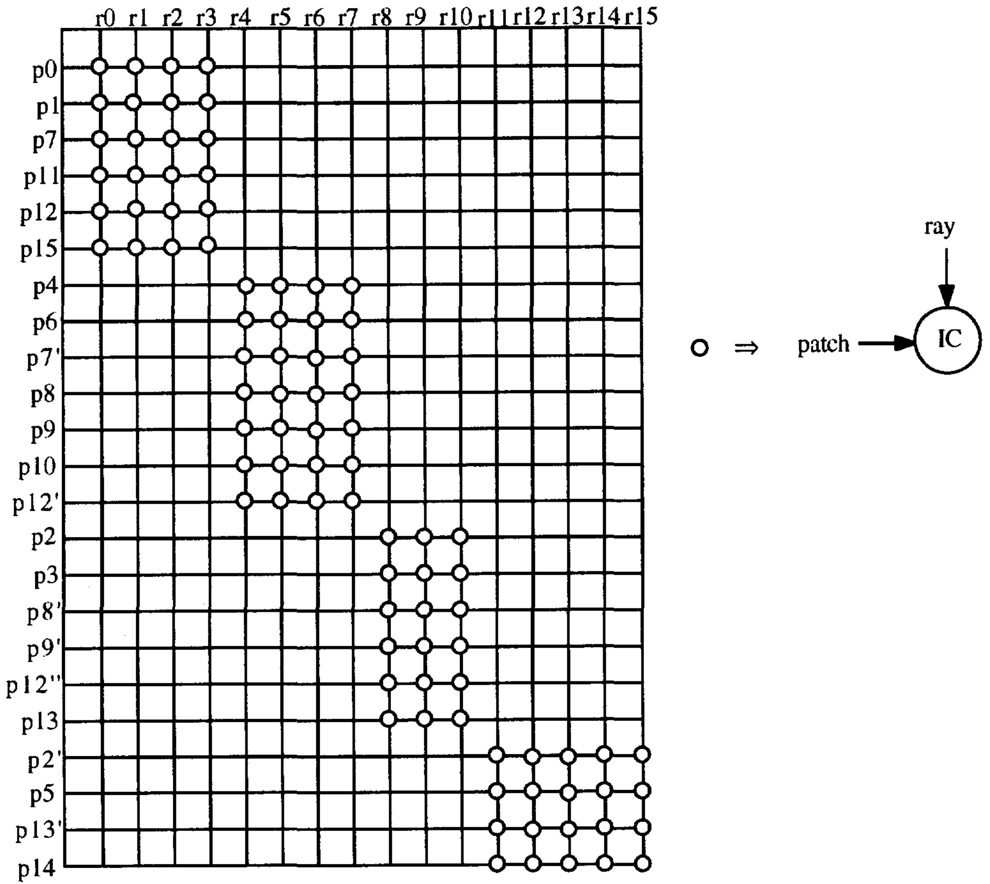


Figure 3.9: The computational structure for the view-dependent partitioning.

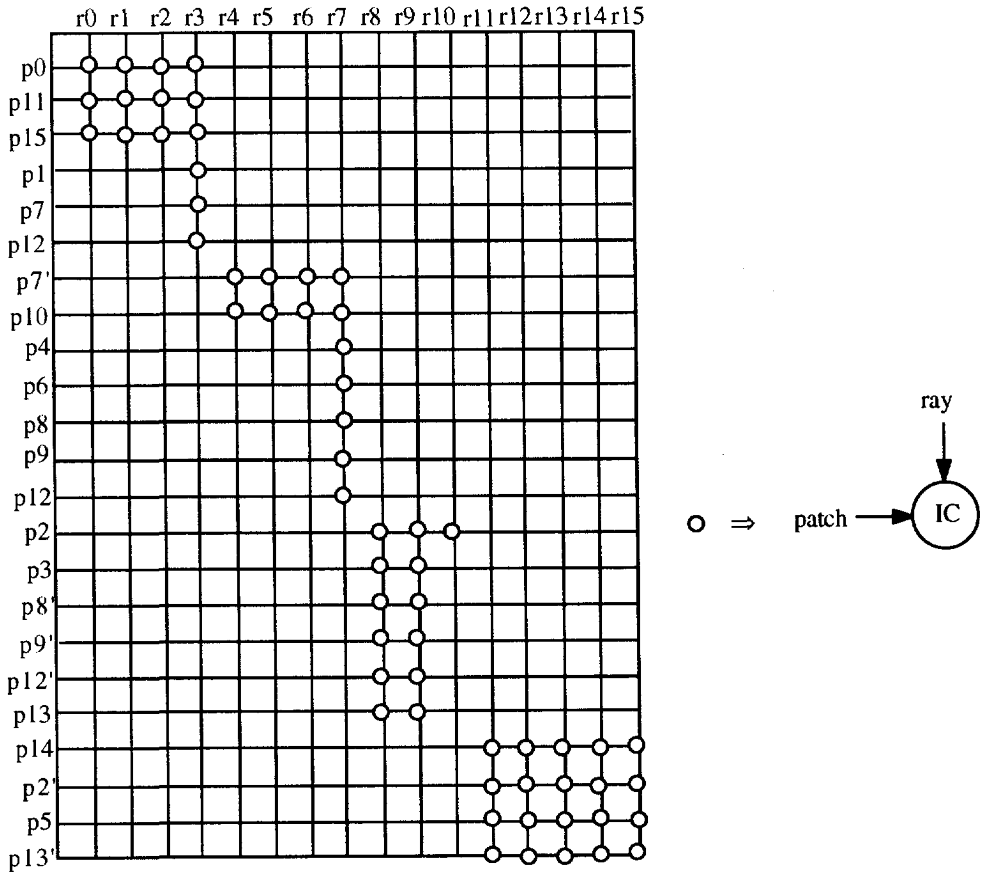


Figure 3.10: The computational structure for the combined structure in Fig. 3.8.

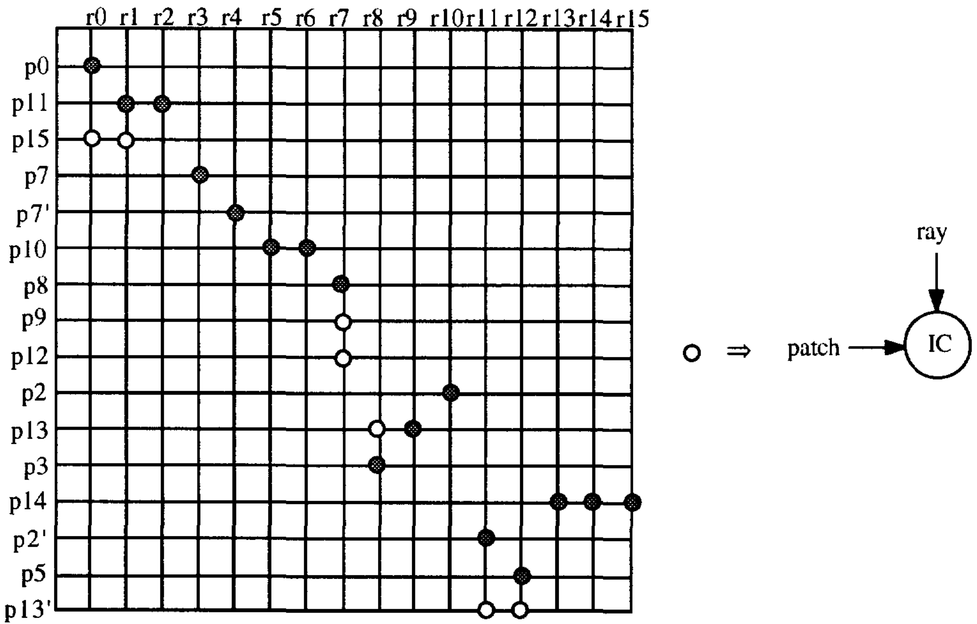


Figure 3.11: The resultant computational structure for the shelling technique.

3.4.1. Issues in the Shelling Technique

By and large, the shelling technique leads to an algorithm which is not so massively parallel than the naive algorithm, but is much efficient due to its build-in visibility ordering of the patches and the rays. In the previous section, we exposed the underlying ideas by visualizing their complexity reducing effects on the computational structure of the naive algorithm. In this section, we shall discuss some key issues in the shelling technique more fully and formally.

We first present the algorithm that implements the shelling technique with the aid of pseudo-code given in Fig. 3.12. For simplicity, we assume that only one partition participates in the wedge-type partitioning (see Fig. 3.7), which is the entire half-space seen through a sample point. The half-space is partitioned into a shell-like structure by a set of concentric spheres centered at the sample point (i.e., *build shell*()). By following that structure, the shelling technique proceeds with a shellwise searching until no more rays are left. For clarity, we distinguish two types of ray: cell-traversal ray (*ctray*) and intersection-computation ray (*icray*). Within a shell, a *ctray* is shot to search for the first cell (i.e., *cell traversal*(*ctray*)). Instead of testing against all the patches in this cell as in the conventional technique, each patch now tests against a bundle of *icrays* determined by the patch's *SBB*. This is done by first computing the *SBB* of a patch (i.e., *spherical bounding box*(*patch*). Then each remaining *icray* within the *SBB* computes intersection with the patch (i.e., *compute intersection*(*icray*, *patch*)). The *ctray*

will continue traversing to start over the same procedure until it reaches the current shell boundary (i.e., *continue traversing*(*ctray*, *shell*)). This is controlled by the condition *shell_boundary* in the inner **while**-loop. After iterating through all the *ctrays* belonging to the current shell via the **for**-loop, the ray memory will be updated to check if there is any survival ray (i.e., *update ray memory*(*icray*)). If this is the case, the above procedure is repeated for the next shell. Otherwise, it ends.

```
Algorithm: Shell

shell_done := false;

while (shell_done = false) do

begin

    shell := build shell();

    for each remaining ctray in shell do

        shell_boundary = false;

        while (shell_boundary = false) do

        begin

            cell := cell traversal(ctray);

            for each patch in cell do

                sbb := spherical bounding box(patch);

                for each remaining icray in sbb do

                    compute intersection(icray, patch);

            shell_boundary := continue traversing(ctray, shell)

        end;

    shell_done := update ray memory(icray)

end;
```

Figure 3.12: The shelling technique.

3.4.1.1. Shellwise Space Search

The shelling technique and the conventional technique are similar in the sense that they both search an area of space where a ray-patch intersection is highly likely, then perform any intersection tests. As a ray does not know about the patches in its path until it strikes one, the concept of searching thus comes up for this purpose. The conventional technique relies on a depth-first search in the sense that a ray makes a way into the space outwards until it hits a patch. An interesting question is: can we take advantage of the known space explored by a ray already shot, which has been ignored by the conventional technique. Due to the property of ray coherence, neighbouring rays will most probably open the same cells and even hit the same patches. It is unwise to search for the known space again, at least for the small bundle of neighbouring rays. This leads to the shellwise space searching as suggested in the shelling technique, which is by nature a breadth-first search. Basically, a ray enters an area of space bounded by a shell boundary to explore the unknown space. As it happens, one or more patches would be found there, representing a space which is known to contain patches. We can now perform intersection tests for each patch in the known space against a bundle of rays potentially hit the patch. After collecting all the remaining rays leaving the current shell, we proceed to the next shell to start over again until no more rays are left (see also Fig. 3.13). This constitutes of the idea of shellwise space search. Note that by using the conventional technique, those intersection tests belonging to the same patches can only take place by shooting neighbouring rays over and over again.

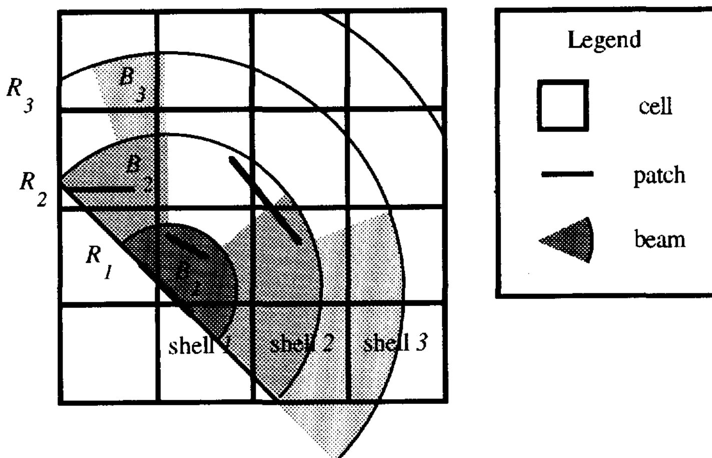


Figure 3.13: Shellwise space searching in the shelling technique.

As stated previously, a highly efficient computational structure can be derived by shellwise searching the space. This is similar to the lazy evaluation of the conventional technique as displayed in Fig. 3.6 by interpreting R_i as the radius of shell i. The program in Fig. 3.14 uses a beam-scheme to explain the concept of the lazy evaluation for the shelling technique. In this figure, B_i represents the bundle of rays (or beam) entering shell i. For simplicity, we assume that each shell only needs to be traversed once for each ray. Within *Shell 1*, each *ctray* in B_1 starts with a cell traversal. Each patch found tests against a bundle of *icrays* determined by the patch's *SBB*. Each *ctray* will continue traversing to start over the same procedure until it reaches the current shell boundary. When all the *ctrays* in B_1 complete the cell traversal and all the patches finish intersection computation, we start checking all the *icrays* in B_1 to see if there is any survival ray. This is done by comparing the current intersected distance of an *icray*, $icray.t_1$, with the current shell radius, R_1. A ray is declared to have survived if the former is larger than the latter. Otherwise, it is declared dead. A new beam, B_2, is formed if there is any survival ray (i.e., *build beam(icray)*). The above procedure is repeated until no more beams are left.

The shelling technique is basically an approach to make the massively parallel naive algorithm more efficient. Therefore, the shelling technique must preserve the inherent parallelism of the naive algorithm as much as possible. The program in Fig. 3.14 uses bundle of rays that traverse cells shellwise and intersects with patches, and is thus one possible parallel algorithm that implements the shelling technique. The potential difficulties of this approach are:

1. The amount of parallelism in computations is decreasing successively from inner shells to outer shells. This is best explained by the continuously reducing-sized beams (see Fig. 3.13). The difficulty is how to manage available resources to cope with this run-time changing parallelism.

2. A new beam can only be formed until all the computations involved in the current beam have been finished. This introduces a totally sequential processing among different shells.

3. As can be seen, the constructs enclosed in the square braces in Fig. 3.14 take the form of mixing up some data-dependent iterations. Indeed we can say that the computational structure of the shelling technique is not known at compile-time. This makes compile-time scheduling infeasible.

The first two difficulties can be solved by the use of ray-frustum casting, as will be discussed in a moment. The third is a general problem for any space partitioning technique. We can only rely on a runtime scheduling, which will be the main topic in chapter 4.

```
shell_done := false;

for each ctray in B_1 do
    cell := cell traversal(ctray);
    for each patch in cell do
        sbb := spherical bounding box(patch);
        for each icray in sbb do
            icray.t_1 := compute intersection(icray, patch);

shell_done := true;
for each icray in B_1 do
    if icray.t_1 ≤ R_1 then icray.t_min := icray.t_1
    else
        shell_done := false;
        B_2 := build beam(icray);

if shell_done := false then
    begin
        for each ctray in B_2 do
            cell := cell traversal(ctray);
            for each patch in cell do
                sbb := spherical bounding box(patch);
                for each icray in sbb do
                    icray.t_2 := compute intersection(icray, patch);
```

```
    shell_done := true;
    for each icray in B_2 do
        if icray.t_2 ≤ R_2 then icray.t_min := icray.t_2
        else
            shell_done := false;
            B_3 := build beam(icray);

if shell_done := false then
    begin
        .
        .
        .
    end
end;
```

Figure 3.14: Lazy evaluation equivalent for the shelling technique.

3.4.1.2. Grid-Ray Cell Traversal

In the shelling technique, we proceed with a shellwise space searching by following a shell-like structure, which is a set of concentric spheres centered at a sample point. Practically, this structure should be superimposed on an underlying data structure. From a hardware perspective, a spatially enumerated auxiliary data structure is chosen for this purpose. Basically, the three-dimensional object space is partitioned into uniformly distributed cells. Each cell stores patch data of those patches which are intersected with the cell. Conventionally, one ray is shot to find the first cell and tested against all patches stored there. If there is no hit, then the next cell is traversed and tested again until there is a hit. The shellwise space searching is similar to this cellwise space searching. The only difference is that each ray must traverse up to the current shell boundary instead of cell boundary. This is accomplished by comparing the ray's current intersected distance to a cell boundary with the shell radius. In this sense, the underlying cell-traversal algorithms of finding cells for both techniques are exactly the same. In the literature, many efficient algorithms have been proposed [FTI86] [Gla84] [MHI88] [TKM84]. In this section, our emphasis will be on the determination of the resolution of cell-traversal rays.

In the shelling technique, we claim that the density of cell-traversal rays can be much lower than that of intersection-computation rays. The density of intersection-computation rays is subject to a number of constraints like the accuracy of form-factors, antialiasing and resolution of images. For high quality, high resolution rendering, this density will generally be high. This is due to that fact that ray casting is a point sampling technique and if a uniform ray distribution is assumed, then small patches or object shape details will determine the ray sampling density. However, the situation is different for cell-traversal rays. Indeed, although cell traversing is also a sampling technique, here the ray density can be much smaller since it is directly depending on the sizes and the distribution of the cells which we have assumed to be fixed and uniformly distributed over space. Therefore, it would be overdone if the cell traversing would be performed with a ray density equal to the high density used for intersection computation. One may ask what the cell size should be and how it depends on the scene to be rendered. The answer is that, for the shelling technique, a medium-sized cell will do, because of the following:

1. For the conventional technique, a small-sized cell is crucial. This is because a ray needs to test against all patches stored in a cell opened by the ray. If a larger cell is taken, more intersection computations are required due to the fact that on the average more patches might be stored in a cell. Rather than relying on small-sized cells to screen out unnecessary patches, the shelling technique relies on the *SBB* of a patch to screen out unnecessary rays. Regardless of the size of a cell, only rays within an *SBB* are used for computing intersection points.
2. The *SBB* of a patch is effective for screening out irrelevant rays in angular directions. In the direction of depth, the shelling technique relies on a shell structure to isolate patches. If two occluded patches belong to different shells, they can be isolated because only remaining rays leaving a shell will be used for intersection computation. Unless those two patches are very close to each other, then they can only be isolated by using small-radius shells. This is not worthy because the depth complexity of a scene is generally low [CH92].

From the above discussion, we conclude that medium-radius shells and hence medium-sized cells suffice to isolate patches in the shelling technique. In view of this, we propose a grid-ray cell traversal based on low-resolution grids imposed by the structure of the medium-sized cell partitioning of the scene.

A prerequisite in applying grid-ray cell traversal is to use uniform cell partitioning of the scene. From a hardware perspective, this is a preferential choice rather than a restriction. This is because a *CTU* is much more efficiently traversed a uniformly partitioned space than other complex data structures like octree or macro region etc. We now discuss how to determine the density of grids in (φ, θ)-space for all the cells in space, as well as the corresponding grid rays that perform cell traversal.

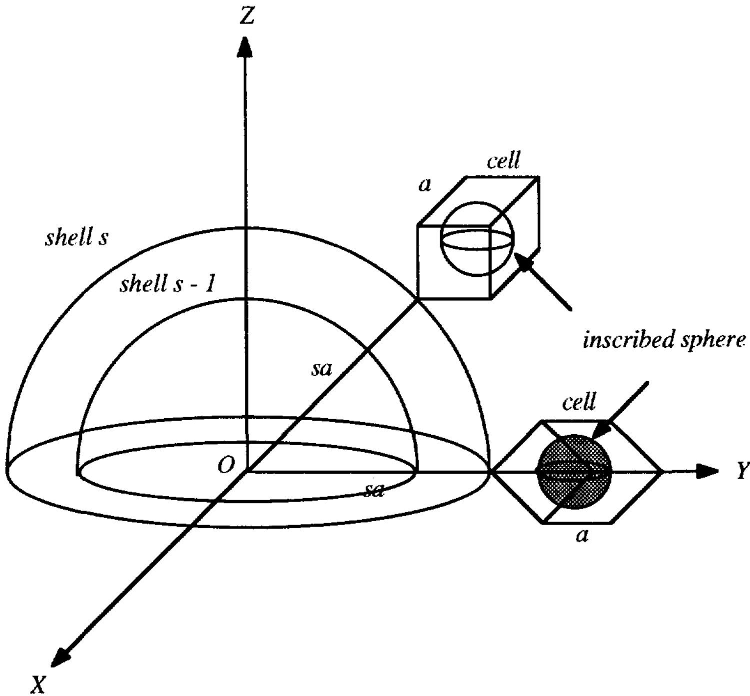


Figure 3.15: The derivation of φ_{grid} and θ_{grid}.

In Fig. 3.15, let O be a sample point on which a shell-structure is built, and let a be the size of uniformly distributed cells. Furthermore, let s and s - 1 be two shells with shell radii sa and $(s - 1)\,a$, respectively. Then the density of grid rays within shell s is determined in such a way that allows to open all the cells intersected with the region bounded by shell s - 1 and shell s. One possible solution to determining the density of grid rays within shell s is to define *inscribed spheres*[3] for all possible cells which just touch the shell s and project those inscribed spheres onto the spherical surface of shell s. After projecting, each inscribed sphere actually defines a circle on shell s. Then the density of grid rays is determined by taking the minimum sampling grids in terms of φ and θ angles to capture all possible projected circles. In [Hek93c], we have proven that the minimum sampling grids occur at the circle defined by projecting the shaded inscribed sphere as shown in Fig. 3.15 and are given by

[3] The inscribed sphere of a cell is defined as the largest sphere that is totally contained in the cell.

$$\begin{cases} \varphi_{grid} = 2\tan^{-1}\left(\frac{r}{\sqrt{2}}\right), \\ \theta_{grid} = 2\tan^{-1}\left(\frac{r}{\sqrt{2+r^2}}\right), \end{cases} \tag{3.1}$$

where

$$r = \frac{1}{\sqrt{-1 + (\sqrt{3} + 2s)^2}}.$$

For shell s, we can define a set of rays uniformly distributed over φ and θ angles, with angular spacing $\varphi_{grid}(s)$ and $\theta_{grid}(s)$ as given in Eq. 3.1 which can serve as cell-traversal rays. Due to the medium-sized cells, and in contrast to the density of intersection-computation rays, a rather low density of rays will thus suffice for cell traversing of the space.

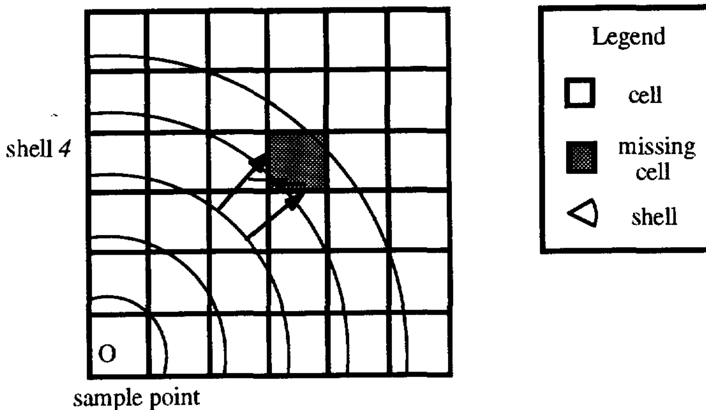


Figure 3.16: A missing cell.

In the shelling technique, a peculiar situation as depicted in Fig. 3.16 can happen. In shell *4*, the two grid rays both miss the shaded cell because the ray's current intersected distance to a cell boundary is larger than the shell radius. In case the cell contains a patch, it will not be considered for intersection computation in shell *4*. Although the patch will be found later in the next shell, some intersection-computation rays which should rather hit that patch will already have been declared as dead in shell *4*. To overcome this missing cell problem, each intersection-computation ray is assigned with a *shell_id* tag representing its version in the course of its lifetime in different shells. Also, each patch is assigned with a *shell_id* tag indicating to which shell it belongs. A set of intersection-computation rays can be determined by comparing each

ray's *shell_id* against a patch's *shell_id.* In the example of Fig. 3.16, the *shell_id* of missing patch is *4*. Hence, for computing intersection points, one must use the bundle of rays whose *shell_id* is *4* too (and not *5*). In this way, the missing patch (which will be found eventually in shell *5*) will recover what was lost in shell *4* and yield correct intersection points.

Having introduced the low-density grid rays, we shall now consider the issue of balancing the workloads going with cell traversal and intersection computation. For the conventional technique, it is very hard to balance the workloads representing cell traversal and intersection computation as pointed out in [TI86]. The difficulty lies in the conflict of reducing one computation together with the other. For instance, the number of intersection computations can be reduced by using small-sized cells, but this brings about an increasing number of cell traversals, and vice versa An important feature in the shelling technique is that it resolves this conflict. By using the shelling technique, the total number of intersection computations appears to be kR, where $k > 1$ and R is the total number of intersection-computation rays. It turns out that the value of k is insensitive to the cell size and is largely determined by the depth complexity of a scene. On the other hand, the number of grid rays and hence the number of cell traversals can be reduced considerably by increasing the cell size. The result is that cell traversal and intersection computation workloads can be balanced.

Consider a ray frustum bounded by two *constant-*φ planes, say $\varphi = \varphi_{min}$, $\varphi = \varphi_{max}$, and two *constant-*θ planes, say $\theta = \theta_{min}$, $\theta = \theta_{max}$, in polar coordinate system. Total number of grid rays up to shell S is given by

$$\sum_{s=1}^{S} \left(\left\lceil \frac{\varphi_{max}}{\varphi_{grid}} \right\rceil - \left\lfloor \frac{\varphi_{min}}{\varphi_{grid}} \right\rfloor + 1 \right) \left(\left\lceil \frac{\theta_{max}}{\theta_{grid}} \right\rceil - \left\lfloor \frac{\theta_{min}}{\theta_{grid}} \right\rfloor + 1 \right). \tag{3.2}$$

Assuming that the ray frustum is defined on grids with spacings φ_{grid} and θ_{grid}, then Eq. 3.2 can be simplified as

$$\sum_{s=1}^{S} \left(\frac{\Delta\varphi}{\varphi_{grid}} + 1 \right) \left(\frac{\Delta\theta}{\theta_{grid}} + 1 \right),$$

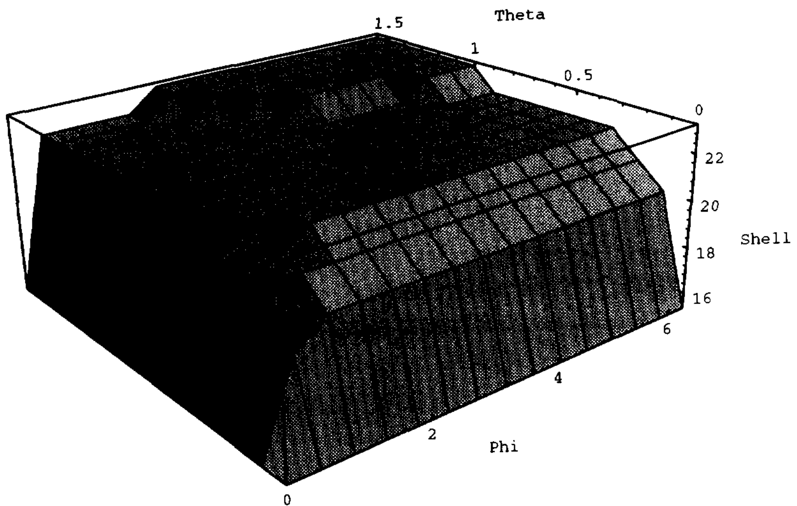
where $\Delta\varphi = \varphi_{max} - \varphi_{min}$ and $\Delta\theta = \theta_{max} - \theta_{min}$.

The number of grid rays can be reduced considerably by increasing the cell size but has little effect on the number of intersection computations. An interesting consequence is that a balancing point in terms of workloads can be obtained by a well-chosen cell size. Therefore, a cluster can be configured as a fixed number of *ICUs* and *CTUs*. In fact, we would like to come to a configuration which is as simple as one *ICU* and one *CTU.*

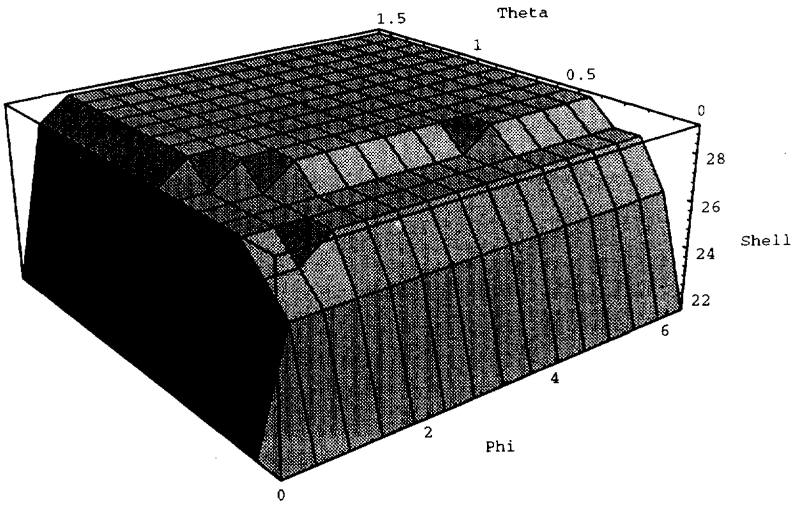
It should be noted that the size of a ray frustum is subject to a lower bound. Going beneath this bound, cell traversals are always more than intersection computations. Consider a ray frustum

containing only one intersection-computation ray which is not defined on grids. By Eq. 3.2, there exist *4* grid rays which may contribute more cell traversals than intersection computations for a sparse scene (i.e. $k \approx 1$). If this not the case, the workloads for cell traversal and intersection computation will be equal when grid rays traverse up to a certain shell-number. We shall call this the balanced shell-number, and is denoted as $S_{balance}$. The balanced shell-number can be derived by equating the workloads for cell traversal and intersection computation. It follows that

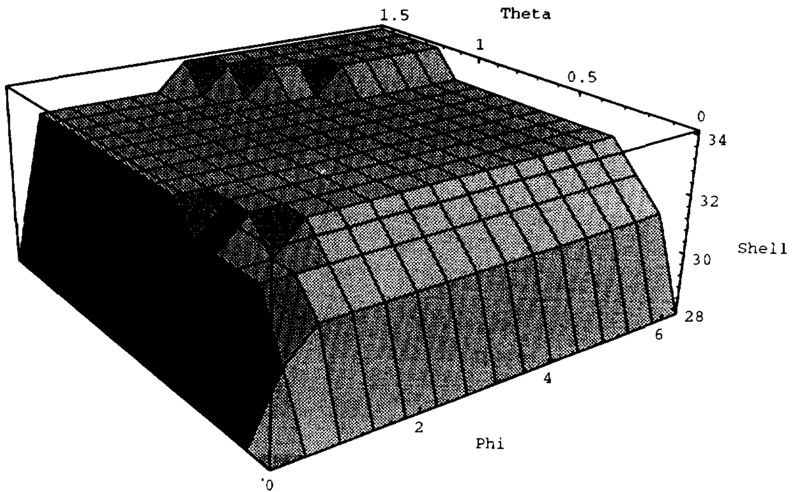
$$\sum_{s=1}^{S_{balance}} \left(\frac{\Delta\varphi}{\varphi_{grid}} + 1\right)\left(\frac{\Delta\theta}{\theta_{grid}} + 1\right) = kR. \tag{3.3}$$



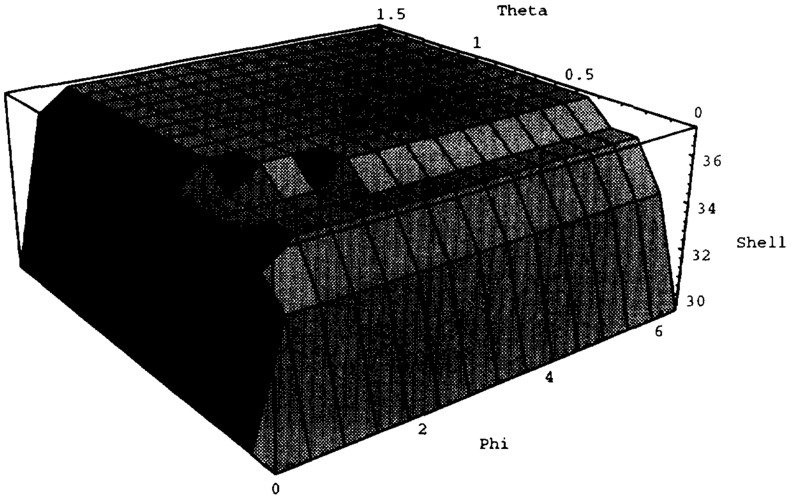
(a) $k = 1$



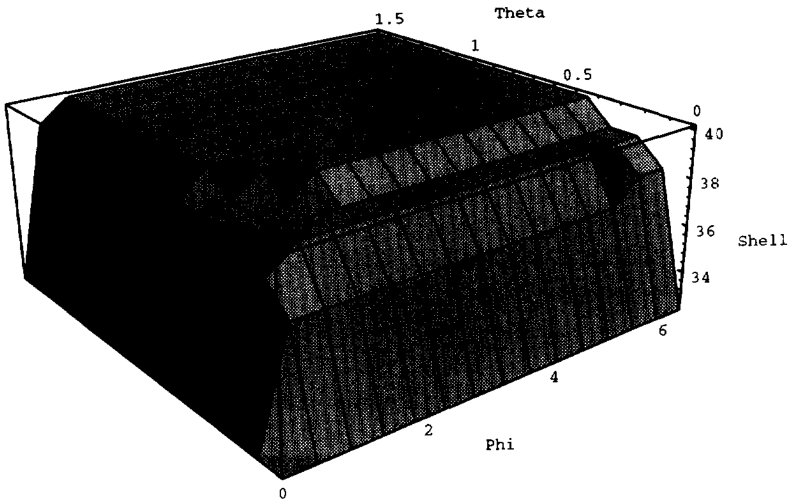
(b) $k = 2$



(c) $k = 3$

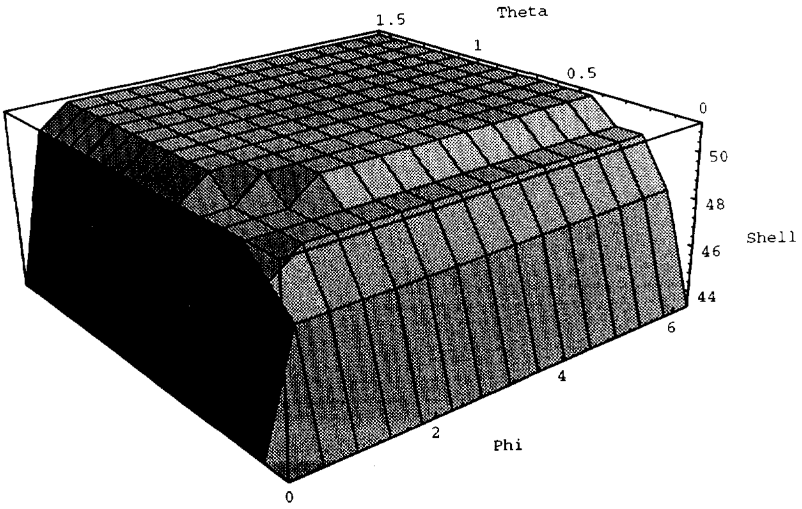


(d) $k = 4$

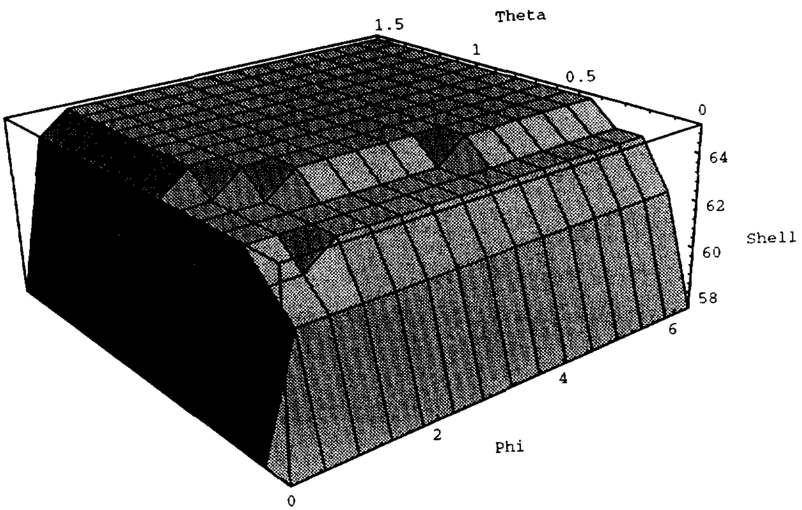


(e) $k = 5$

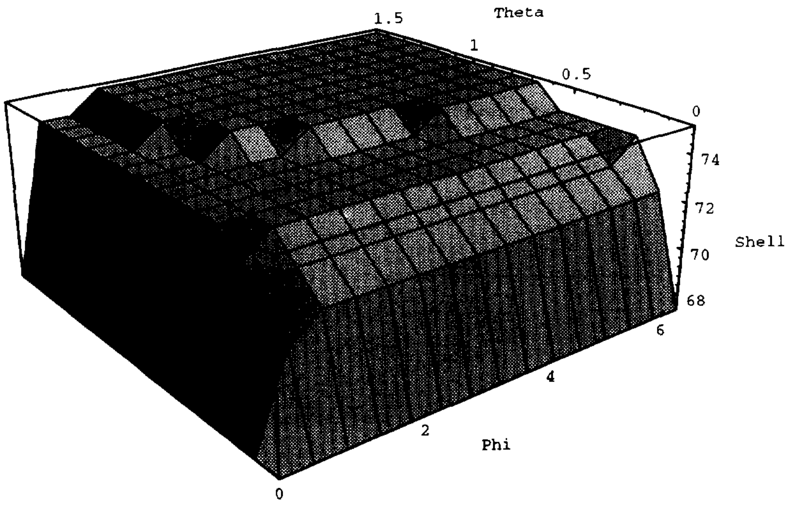
Figure 3.17: The balanced shell-number (Shell) vs $\Delta\varphi$ (Phi) and $\Delta\theta$ (Theta) for R = 100K.



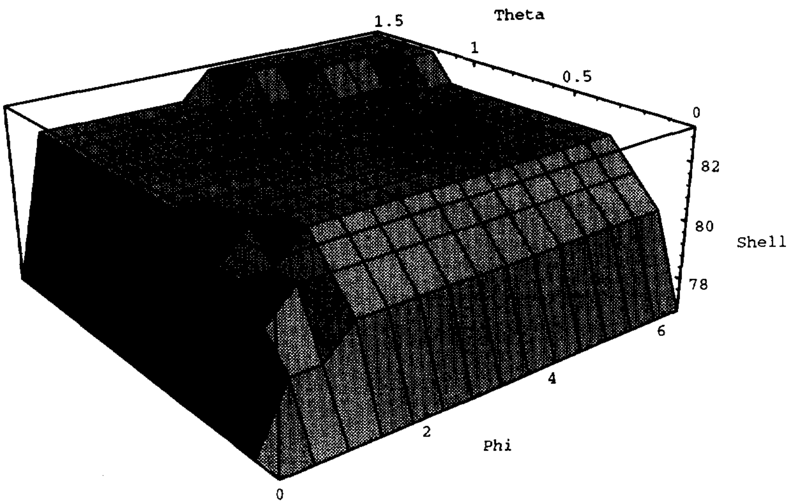
(a) $k = 1$



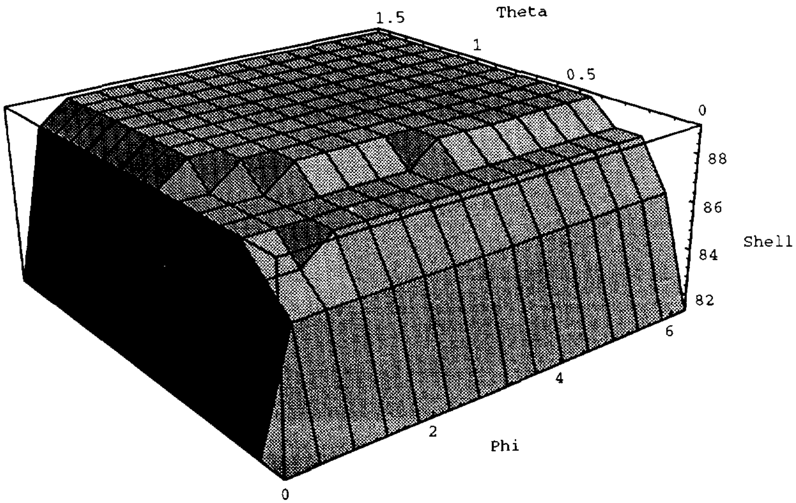
(b) $k = 2$



(c) $k = 3$

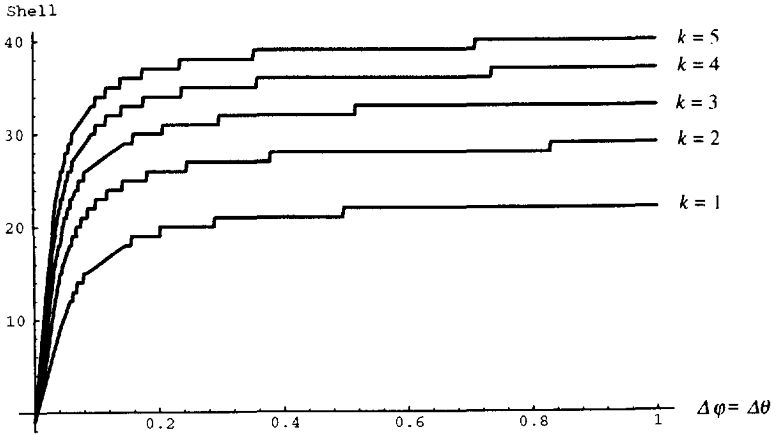


(d) $k = 4$

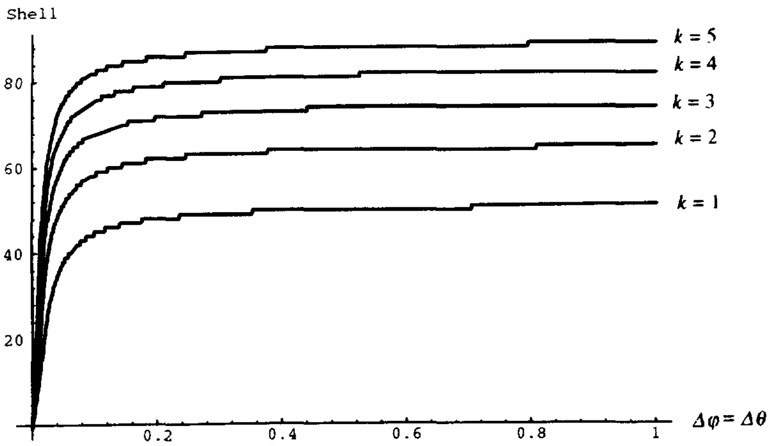


(e) $k = 5$

Figure 3.18: The balanced shell-number (Shell) vs $\Delta\varphi$ (Phi) and $\Delta\theta$ (Theta) for R = 1M.



(a) R = 100K



(b) R = 1M

Figure 3.19: The two-dimensional representation of Figs. 3.17 and 3.18 in case $\Delta\varphi = \Delta\theta$.

Figs. 3.17 and 3.18 show three-dimensional plots of the balanced shell as a function of $\Delta\varphi$ and $\Delta\theta$ for different values of R and k. For clarity, we also give their two-dimensional cuts in Fig. 3.19, where $\Delta\varphi = \Delta\theta$. These figures reveal some interesting features.

1. The balanced shell-number increases as the number of intersection-computation rays grows. This allows to use small shell-radius when the number of intersection-computation rays is large, representing a more efficient shell-structure which may reduce the number of intersection computations. When $k = 3$, the balanced shells for R = 100K and R = 1M, are 30 and 70, respectively. This allows to choose 0.05 ($\approx \sqrt{3}/30$)and 0.025 ($\approx \sqrt{3}/70$) as the cell size, where $\sqrt{3}$ is the worst case distance a grid ray may traverse.

2. For a complex scene, an efficient shell structure becomes very important in order to reduce the number of intersection computations. This is because the depth complexity of a complex scene is generally high. It thus requires an efficient shell-structure with small shell-radius to isolate patches. In contrast, the shell-structure is not so important for a simple scene since the depth complexity is quite low. This suggests that the shell-radius and so the cell size should be adjusted in accordance with the scene complexity. It is interesting that the balanced shell-number can be adjusted consistently according to the depth complexity. From Fig. 3.19, it is seen that the balanced shell-number increases as the k value increases. A complex scene often reflects a larger k value, and thus allows a

larger balanced shell-number, representing a smaller shell-radius as desired.

3.4.1.3. Bundle-Ray Intersection

Instead of testing against all the patches in a cell as in the conventional technique, each patch found in the shellwise space search tests against a bundle of rays determined by the patch's *SBB*. The implications of this bundle-ray intersection on the shelling technique are twofold: on the one hand this helps in building the visibility ordering on the ray side as irrelevant rays for a patch can be largely screened out by the patch's *SBB*, on the other hand the communication overhead can be amortized over many useful computations. The success of this bundle-ray intersection is directly attributed to exploiting the property of coherence. In the literature, different manifestations of coherence were used to accelerate ray tracing. They can be classified into the following categories.

1. Object Coherence

   Object coherence means that objects tend to be continuous and distinct objects tend to be disjoint in space. Since local neighbourhoods of space are most likely occupied by the same objects, they can be localized by creating a hierarchy of bounding volumes around them or by partitioning the object space using various encoding schemes like octree, binary tree, or a spatially enumerated auxiliary data structure. The performance of ray-tracing can be greatly improved upon by checking for intersection with simple bounding volumes and traversing the space in a judicious way instead of going through exhaustive search.

2. image Coherence

   Image coherence can be considered a consequence of object coherence after projecting an object onto a two-dimensional image plane. The local constancy of space gives rise to a similar property in the image, across which colours change only gradually.

3. Ray Coherence

   Ray coherence means that rays with nearly the same origins and directions will probably intersect with the same objects in the environment. This property has been exploited to reduce computation by considering bundles of rays (also called beams) that interact as a whole with objects in the environment.

The shelling technique exploits both object coherence and ray coherence. Object coherence is accounted for by means of uniformly distributed cells. Ray coherence is accounted for by means of a classification technique which determines potential rays that may intersect a patch. This can avoid ray-patch intersection computations for such rays do not have to take place. The question is how to select an appropriate classification technique. One may choose to use a fast method but may classify rays which are far too many. The other may pay a higher overhead by using a superior method that can classify potential hit-rays accurately. The tradeoff largely depends on the following considerations:

1. Traditionally, objects have been modelled by means of polygons to take advantage of machines that provide hardware acceleration for polygonal rendering. We aim at developing a rendering accelerator that supports both polygon and Bezier patch modelling of objects. We thus have to come up with a classification technique that can handle both polygon and Bezier patch.

2. Calculation of ray identifiers (i.e., ray addresses) is very frequently performed in any classification technique. To make this calculation simple, it is natural to choose an easy distribution function to master rays, subject to the constraint that the quality of the image must not be sacrificed.

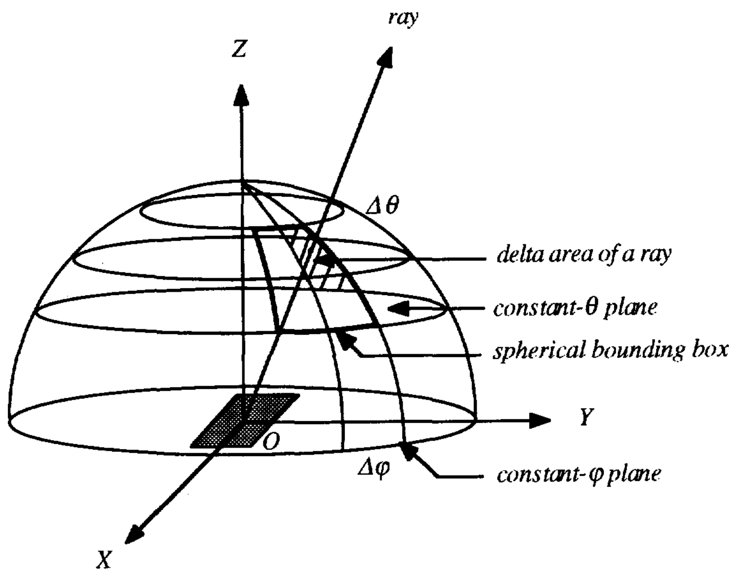


Figure 3.20: An example of *SBB* defined on the surface of a hemisphere.

To facilitate our further discussion, we define some basic terms in the following.

Let V be a set of points in a plane, the *convex hull* of V is the smallest convex object containing all the points. This is the ordinary convex hull used in the area of computational geometry, which is defined on a set of points. In the field of computer graphics, another kind of convex hull called *convex hull of directions* plays an important role.

Let $\overrightarrow{PQ}$ be a vector from the point P to the point Q. The direction of $\overrightarrow{PQ}$ is defined to be the unit vector $\overrightarrow{PQ} / |\overrightarrow{PQ}|$, where $|\overrightarrow{PQ}|$ is the magnitude of the vector $\overrightarrow{PQ}$. We now give the definition of convex hull of directions as follows:

Definition 3.2 (Convex Hull of Directions) Let $\overrightarrow{OP_0}, \ldots, \overrightarrow{OP_n}$, be a set of directions defined by a set of points $P_0, \ldots, P_n$ on a unit sphere centered at point O. The *convex hull of directions* $\overrightarrow{OP_0}, \ldots, \overrightarrow{OP_n}$, is defined as a minimum convex set on the surface of the unit sphere which contains $P_0, \ldots, P_n$. ■

Consider a ray-casting procedure. A unit hemisphere is placed around a sample point on a source patch such that the base of the hemisphere is perpendicular to the normal of the patch at the sample point. The surface of the hemisphere is discretized into delta areas by *constant-*φ and *constant-*θ planes uniformly distributed over (φ, θ)-plane with constant $\Delta\varphi$ and $\Delta\theta$. Rays are then cast from the sample point through the center of each delta area to the environment, as shown in Fig. 3.20. By definition, the convex hull of directions for a set of so-defined rays is the minimum convex set on the surface of the hemisphere containing all the centers of those delta areas corresponding to the set of rays. Alternatively, we could derive a spherical surface on the hemisphere, which is bounded by two *constant-*φ and *constant-*θ planes (see Fig. 3.20), that contains all the points. This is what we call the spherical bounding box of directions, as is defined below.

Definition 3.3 (Spherical Bounding Box of Directions) Let $\overrightarrow{OP_0}, \ldots, \overrightarrow{OP_n}$, be a set of directions defined by a set of points $P_0, \ldots, P_n$ on a unit sphere centered at point O. The *spherical bounding box of directions* $\overrightarrow{OP_0}, \ldots, \overrightarrow{OP_n}$, is defined as a spherical surface on the sphere, which is bounded by two *constant-*φ and two *constant-*θ planes in polar coordinate system, that contains $P_0, \ldots, P_n$. ■

Definition 3.4 (Spherical Bounding Box of a Patch) Let H be a unit hemisphere with center O and normal vector directed in Z direction, let P be a polygon or Bezier patch, and let P' be the projected image of P by projecting each point of P perspectively onto H. The *spherical bounding box of* P is the spherical bounding box of directions defined by all the points in P' and the point O. ■

Having introduced the definition of *SBB*, we now come to the comparison of classification techniques. In the literature, different ways of classifying potential rays that intersect a patch have been proposed. Two of them are the following:

1. Bounding Sphere

   Let P be a point, and let C be a sphere with center O, $O \neq P$, and radius r. Furthermore, let Q be a point inside C. Then the following inequality holds for the inner product of directions $dir(\overrightarrow{PQ})$ and $dir(\overrightarrow{PO})$.

$$(dir(\overrightarrow{PQ}), dir(\overrightarrow{PO})) > \sqrt{1 - (r/|PO|)^2},$$

   where $(dir(\overrightarrow{PQ}), dir(\overrightarrow{PO}))$ is the inner product of directions $dir(\overrightarrow{PQ})$ and $dir(\overrightarrow{PO})$. Taking a

sphere as the bounding volume of a patch, then we can classify potential hit-rays by checking the above inequality. Because a sphere is a rather loose bounding volume of a patch, it is likely that too many rays will be classified as potential hit-rays. Moreover, it is difficult to derive ray identifiers for the potential hit-rays which are confined to a circle on the hemisphere . This is because those ray identifiers are unstructured in some sense.

2. Convex Hull

   Alternatively, we can project a patch onto a unit hemisphere centered at the sample point then determine a minimum convex set on the surface of the hemisphere which contains the projected patch. Although this method allows to classify potential hit-rays accurately, it is not practical to use it because its speed and numerical stability are very poor. Again, it is difficult to derive ray identifiers for the potential hit-rays which are confined to the convex hull.

From a hardware perspective, identifying ray coherence requires *quickly* determining a *reasonable* bounding volume for a patch that allows *efficient* retrieving of rays. As mentioned in section 3.4, we may choose the *SBB* of a patch as the bounding volume. At least, this makes the retrieval of rays much easier. We shall discuss this in the following.

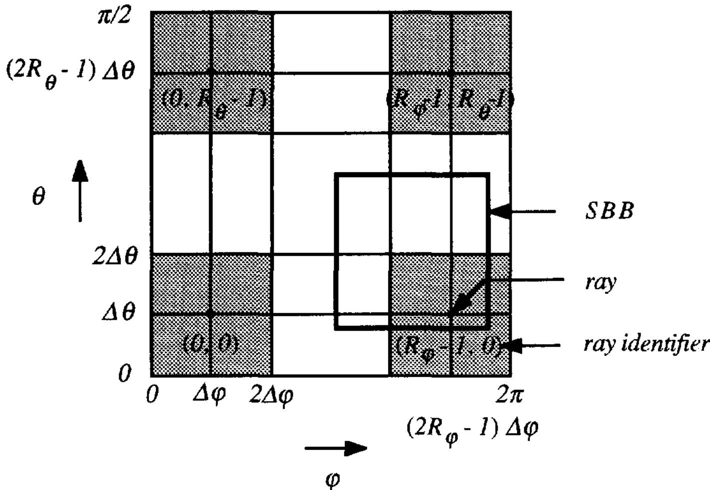


Figure 3.21: A rectangular window on the (φ, θ)-plane defined by an *SBB*.

Consider a ray-casting procedure. Let r_{ij} $i = 0, 1, .., R_{\varphi} - 1, j = 0, 1, .., R_{\theta} - 1$, be $R = R_{\varphi} \times R_{\theta}$ rays uniformly distributed over (φ, θ)-plane (see Fig. 2.4b). The $\Delta\theta$ and $\Delta\varphi$ can be

obtained as follows:

$$\Delta\theta = \frac{\frac{\pi}{2}}{2R_\theta},$$

$$\Delta\varphi = \frac{2\pi}{2R_\varphi}.$$

For addressing purpose, each ray is assigned with a two-dimensional index, (A_φ, A_θ), called ray identifier. For a ray defined on grid (φ, θ), its ray identifier can be computed as follows:

$$A_\theta = \frac{\frac{\theta}{\Delta\theta} - 1}{2},$$

$$A_\varphi = \frac{\frac{\varphi}{\Delta\varphi} - 1}{2}. \tag{3.4}$$

By definition, the *SBB* of a patch is a spherical surface on a unit hemisphere, and is bounded by two *constant-φ* planes, say $\varphi = \varphi_1, \varphi = \varphi_2$, $\varphi_1 \leq \varphi_2$, and two *constant-θ* planes, say $\theta = \theta_1, \theta = \theta_2$, $\theta_1 \leq \theta_2$. It defines a rectangular window on the (φ, θ)-plane, as shown in Fig. 3.21. By virtue of Eq. 3.4, the ray identifiers for boundary rays of the window are given by

$$A_{\theta_1} = \left\lceil \frac{\frac{\theta_1}{\Delta\theta} - 1}{2} \right\rceil, \quad A_{\theta_2} = \left\lfloor \frac{\frac{\theta_2}{\Delta\theta} - 1}{2} \right\rfloor,$$

$$A_{\varphi_1} = \left\lceil \frac{\frac{\varphi_1}{\Delta\varphi} - 1}{2} \right\rceil, \quad A_{\varphi_2} = \left\lfloor \frac{\frac{\varphi_2}{\Delta\varphi} - 1}{2} \right\rfloor, \tag{3.5}$$

where $\lceil . \rceil$ and $\lfloor . \rfloor$ represent ceil and floor operators, respectively. All the potential hit-rays can then be retrieved by addressing the window: $A_{\theta_1} \leq A_\theta \leq A_{\theta_2}$ and $A_{\varphi_1} \leq A_\varphi \leq A_{\varphi_2}$.

The question that remains to be answered is how to quickly determine a reasonable *SBB* of a polygon or Bezier patch. By definition, the *SBB* of a patch is a spherical surface bounded by two *constant-φ* and *constant-θ* planes, that contains the projected image of the patch on the hemisphere. Theoretically, we could project each point of the patch onto the hemisphere, then the *constant-φ* and *constant-θ* planes that bound the *SBB* are those planes taking the smallest and largest φ and θ values among all projected points. One may ask can we just count on the 4 vertices of a polygon or the 16 control points of a Bezier patch to determine those bounding

planes. It turns out that the smallest and largest φ values, as well as the largest θ value of the vertices (or control points) can be used for this purpose. Unfortunately, the minimum θ bounding plane of the *SBB* may not necessarily take the smallest θ value among the vertices (or control points). It is therefore necessary to find an efficient way to *estimate* the minimum θ of the patch. By *estimate* we mean that the derived minimum θ value must be less than or equal to the minimum θ of the patch. This is accomplished by first determining the smallest θ value among the vertices (or control points), and then subtracting a correction term ε_θ from the smallest θ value to get a value guaranteed to be less than or equal to the minimum θ of the patch. The next lemma provides that correction term.

Lemma 3.1: Let L be a line segment with endpoints $P\ (1, \varphi_1, \theta_1)$ and $Q\ (1, \varphi_2, \theta_1)$, where $0 \le \varphi_1, \varphi_2 \le \pi$ and $0 \le \theta_1 \le \pi/2$. Then the difference between θ_1 and the minimum θ angle of L, denoted as ε_θ, is given by

$$\varepsilon_\theta = \begin{cases} \dfrac{\pi}{2} & \text{if } \Delta\varphi = |\varphi_2 - \varphi_1| = \pi \text{ and } \theta_1 = \dfrac{\pi}{2}, \\ \theta_1 - \cos^{-1}\left(\dfrac{\cos\theta_1}{\sqrt{1 - \sin^2\theta_1 \sin^2(\frac{\Delta\varphi}{2})}}\right) & \text{otherwise.} \end{cases} \tag{3.6}$$

Proof: It is trivial to prove the case when $\Delta\varphi = \pi$ and $\theta_1 = \pi/2$. We are now dealing with the non-trivial case. In Fig. 3.22, let N be any point on L, and let M be the middle point of L. The θ angle of N is given by

$$\theta = \tan^{-1}\left(\frac{c}{z}\right).$$

Because $y = \tan^{-1} x$ is a monotonically increasing function of x when $0 \le y \le \pi/2$, it follows that the minimum θ angle of L, θ_{min}, happens at M, and is given by

$$\theta_{min} = \cos^{-1}\left(\frac{x}{\sqrt{x^2 + b^2}}\right).$$

Substituting x by $\cos\theta_1$, and subsequently b by $\sin\theta_1 \cos\Delta\theta/2$, we obtain

$$\theta_{min} = \cos^{-1}\left(\frac{\cos\theta_1}{\sqrt{1 - \sin^2\theta_1 \sin^2(\frac{\Delta\varphi}{2})}}\right).$$

Therefore

$$\varepsilon_\theta = \theta_1 - \theta_{min} = \theta_1 - \cos^{-1}\left(\frac{\cos\theta_1}{\sqrt{1 - \sin^2\theta_1 \sin^2(\frac{\Delta\varphi}{2})}}\right).$$

This proves the lemma. ■

The above lemma states that the minimum θ angle of a line segment with two endpoints lying on a unit hemisphere can be estimated by subtracting a correction term ε_θ from the smallest θ value of the two endpoints. We now come to the main theorem.

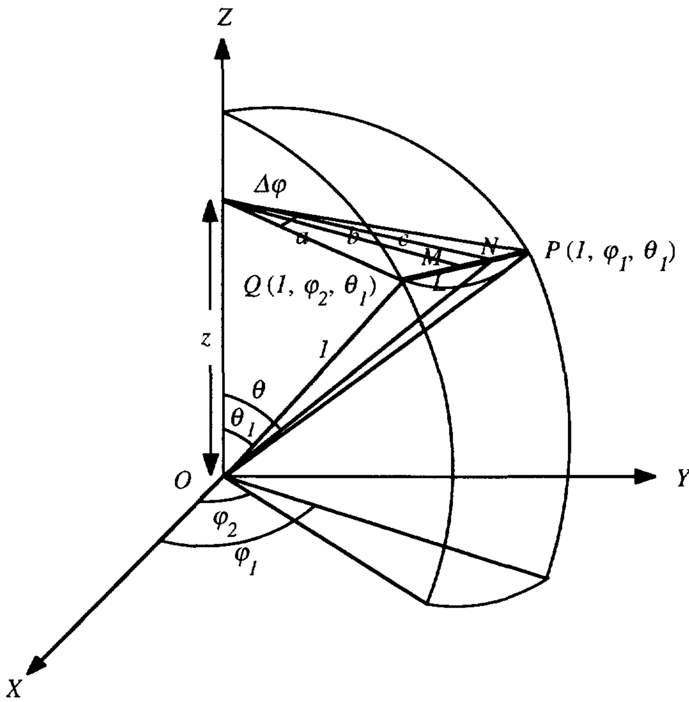


Figure 3.22: The derivation of ε_θ for the line segment L.

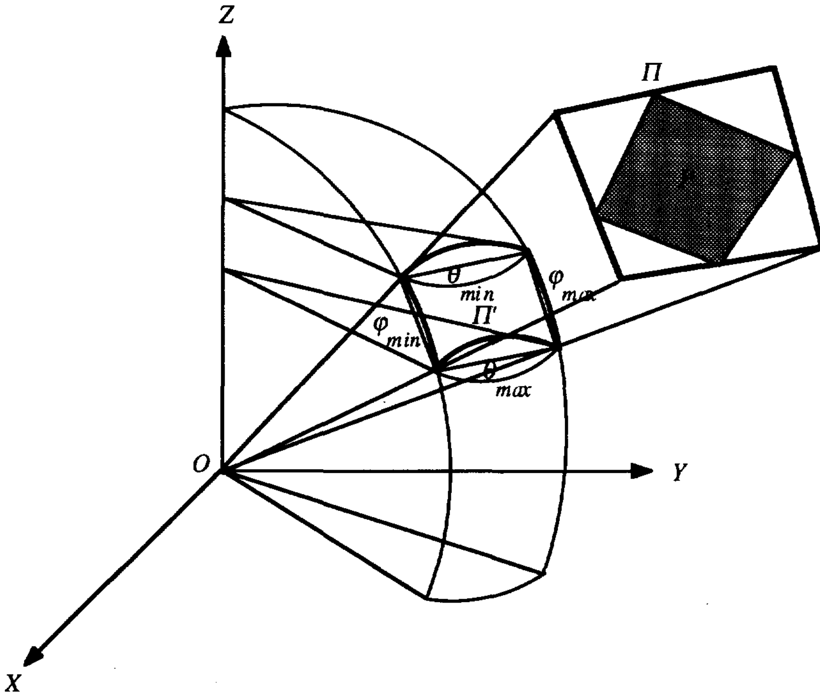


Figure 3.23: The maximal extent Π of P and its projected part Π'.

Theorem 3.1: Let H be a unit hemisphere on the plane $Z = 0$, centered at a point O. Let P be a polygon, and let P' be the projection of P onto H with O the center of projection. Furthermore, let φ_{min} and θ_{min} be the smallest φ and θ angles among the vertices of P, respectively, and let φ_{max} and θ_{max} be the largest φ and θ angles among the vertices of P, respectively. Then the spherical surface on the hemisphere, which is bounded by the following *constant-*φ and *constant-*θ planes:

$$\begin{cases} \varphi = \varphi_{min}, \\ \varphi = \varphi_{man}, \\ \theta = \theta_{min} - \varepsilon_{\theta}(\varphi_{max} - \varphi_{min}, \theta_{min}), \\ \theta = min(\theta_{max}, \frac{\pi}{2}), \end{cases} \tag{3.7}$$

is an *SBB* of P. The *min*(.) takes the minimum of the two arguments.

Proof: In Fig. 3.23, let Π be the maximal extent of the patch P such that the angular components of the vertices of Π are still bounded by θ_{min}, θ_{max}, φ_{min}, and φ_{max}. Let Π' be the projection of Π onto H with O the center of projection. Then P' is contained in Π'. If we can prove that the above four bounding planes take the minimum and maximum angular components of Π', then, by definition, S is an SBB of P.

It is clear that the minimum and maximum θ angles of Π' are θ_{min} and θ_{max}. Consider the minimum φ angle of Π'. By lemma 3.1, this angle is $\theta_{min} - \varepsilon_\theta(\varphi_{max} - \varphi_{min}, \theta_{min})$. The maximum θ angle of Π' is θ_{max}, because the error ε_θ on the hemisphere H is non-positive, but it must be clipped against H and thus is limited to $\pi/2$. This proves the theorem. ■

For practical usage, ε_θ is a rather complex function of two parameters θ_1 and $\Delta\varphi$. There is a clear need for further exploration to make it appealing for hardware implementation. We observe that ε_θ has the following properties:

1. In Eq. 3.6, consider the case when $\theta_1 \neq \pi/2$. It turns out that ε_θ is a monotonically increasing function of $\Delta\varphi$ for $\theta_1 = constant$.
2. Furthermore, ε_θ is bounded between 0 and $(\varepsilon_\theta)_{max}$ when $\Delta\varphi = constant$, and is given by

$$0 \le \varepsilon_\theta |_{\varphi_1 = onstant} \le (\varepsilon_\theta)_{max} = cos^{-1} \left(\frac{2\sqrt{1-C}}{2-C}\right), \quad (3.8)$$

where

$$C = 2\, sin^2 \frac{\Delta\varphi}{4}.$$

From Eq. 3.6, it follows that

$$\frac{\partial cos\, \varepsilon_\theta}{\partial \theta_1}\Big|_{\Delta\varphi = constant} = \frac{sin\, 2\theta_1}{\sqrt{1 - D\, sin^2\, \theta_1}} \left(\frac{D\,(1 - C\, sin^2\, \theta_1)}{2\,(1 - D\, sin^2\, \theta_1)} - C\right),$$

$$\frac{\partial cos\, \varepsilon_\theta}{\partial \Delta\varphi}\Big|_{\theta_1 = constant} = -\sqrt{\frac{D}{1 - D\, sin^2\, \theta_1}}\, \frac{C\, sin^2\, 2\theta_1}{8\,(1 - D\, sin^2\, \theta_1)}, \quad (3.9)$$

where

$$D = sin^2 \frac{\Delta\varphi}{2} = C\,(2 - C).$$

From Eq. 3.9, it follows that, if $\theta_1 = constant$, then $cos\, \varepsilon_\theta$ is a monotonically decreasing

function of $\Delta\varphi$ because

$$\frac{\partial \cos \varepsilon_\theta}{\partial \Delta\varphi}\Big|_{\theta_1 = constant} \leq 0.$$

This proves that ε_θ is a monotonically increasing function of $\Delta\varphi$ when $\theta_1 = constant$.

Consider the case when $\Delta\varphi = constant$. The maximal error $(\varepsilon_\theta)_{max}$ can be derived by setting

$$\frac{\partial \cos \varepsilon_\theta}{\partial \theta_1}\Big|_{\Delta\varphi = constant} = 0$$

and checking if

$$\frac{\partial^2 \cos \varepsilon_\theta}{\partial \theta_1^{\,2}}\Big|_{\Delta\varphi = constant} < 0.$$

From Eq. 3.9, we have

$$\frac{D\,(1 - C \sin^2 \theta_1)}{2\,(1 - D \sin^2 \theta_1)} - C = 0,$$

so that

$$(\varepsilon_\theta)_{max}\Big|_{\theta_1^*} = \cos^{-1}\left(\frac{2\sqrt{1 - C}}{2 - C}\right), \tag{3.10}$$

and

$$\theta_1^* = \sin^{-1}\sqrt{\frac{1}{2 - C}}. \tag{3.11}$$

This proves that ε_θ is bounded as stated.

A three-dimensional plot of ε_θ as a function of θ_1 and $\Delta\varphi$ is shown in Fig. 3.24. The effects of $\Delta\varphi$ on θ_1^* and $(\varepsilon_\theta)_{max}$ are shown in Fig. 3.25a and 3.25b. It is seen that θ_1^* monotonically increases from $\pi/4$ to $\pi/2$ when $\Delta\theta$ increases from 0 to π, and that $(\varepsilon_\theta)_{max}$ monotonically increases from 0 to $\pi/2$ when $\Delta\varphi$ increases from 0 to π. See also Table 3.1 for some specific values. The maximal error $(\varepsilon_\theta)_{max}$ is less than 10° when $\Delta\varphi$ is smaller than 90°. This motivates us to store the values of $(\varepsilon_\theta)_{max}$ with respect to $\Delta\varphi$ ranging from 0 to π in a table. Because ε_θ is a monotonically increasing function of $\Delta\varphi$ for $\theta_1 = constant.$, this allows us to quantize the whole φ range (i.e., from 0 to π) into a set of $\Delta\varphi_q$ values. Based on the $\Delta\varphi$ of a patch, instead of calculating ε_θ on-the-fly, we retrieve the value of $(\varepsilon_\theta)_{max}$ for that $\Delta\varphi_q$ closest to and larger than $\Delta\varphi$ from the table.

Table 3.1: Some specific values for $(\varepsilon_\theta)_{max}$.

$\Delta\varphi$ (degree)	θ_1^*(degree)	$(\varepsilon_\theta)_{max}$(degree)
10	45.05	0.11
20	45.22	0.44
30	45.50	0.99
60	47.06	4.12
90	49.94	9.88
120	54.74	19.47
150	63.04	36.07
180	90	90

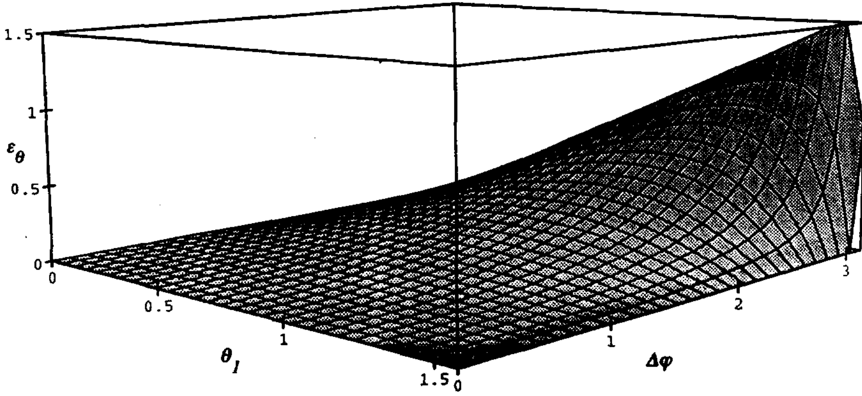
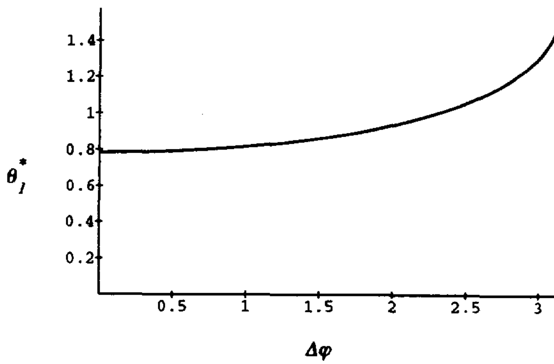
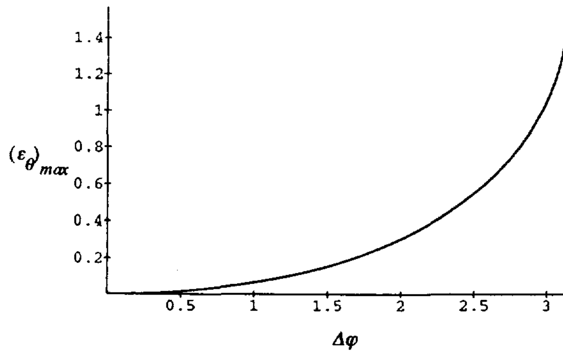


Figure 3.24: The three-dimensional graph of ε_θ vs. θ_1 and $\Delta\varphi$.



(a)



(b)

Figure 3.25: The figure of $(\varepsilon_\theta)_{max}$ vs. $\Delta\varphi$.

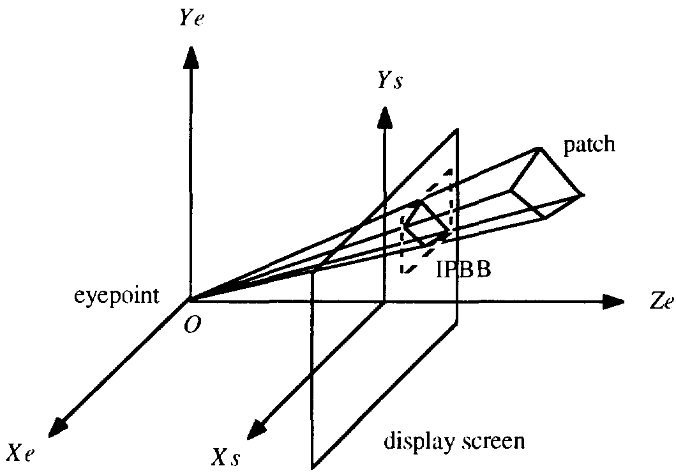


Figure 3.26: The image plane bounding box for determining primary rays.

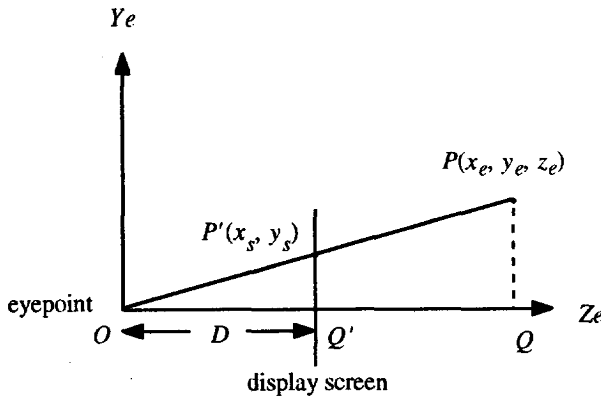


Figure 3.27: The derivation of (x_s, y_s) for perspective projection.

Let V and S be the origin (i.e., viewpoint) and the display screen in an eye coordinate system, respectively. Let P be a polygon, and let P' be the projection of P onto S with V the center of projection. By definition, the *IPBB* of P is a rectangle on S, which is bounded by two *constant-x_s* and two *constant-y_s* planes, that contains P'. The derivation of *IPBB* is much easier than that of *SBB*. We can simply project each vertex of P onto S with V the center of projection. Then the four *constant-x_s* and *constant-y_s* planes that bound the *IPBB* take the smallest and the largest x_s and y_s values among the four projected vertices. The coordinates (x_s, y_s) of the projected point p' (x_s, y_s) of a point p (x_e, y_e, z_e) can be easily computed. Indeed, consider the (Ye, Ze)-plane drawn in Fig. 3.27. The triangles $OQ'P'$ and OQP are similar, giving the relation $y_s/D = y_e/z_e$. A similar construction in the $XeZe$ plane yields $x_s/D = x_e/z_e$. After obtaining the coordinates (x_s, y_s) of all the vertices, it is trivial to determine the smallest and largest x_s and y_s coordinates that define the *IPBB*.

3.4.1.4. Ray-Frustum Casting

Recall that, the shelling technique proceeds with a shellwise space searching. Each patch found tests against a bundle of rays determined by the patch's *SBB*. As long as a patch becomes known, we may adhere to it and use a bundle of rays that traverse cells as usual. This leads to the ray-frustum casting. We first give the definition of a ray frustum.

Definition 3.5 (Ray Frustum) A *ray frustum* is defined as the region bounded by the following *constant-φ*, *constant-θ* or *constant-r* planes represented in polar coordinate system:

$$\begin{cases} \varphi = \varphi_{min}, & \varphi = \varphi_{max}, \\ \theta = \theta_{min}, & \theta_{max}, \\ r = 0, & r = r_{max}, \end{cases}$$

where $0 \leq \varphi_{min} \leq \varphi_{max} \leq 2\pi$, $0 \leq \theta_{min} \leq \theta_{max} \leq \pi/2$ and $r_{max} \geq 0$. ■

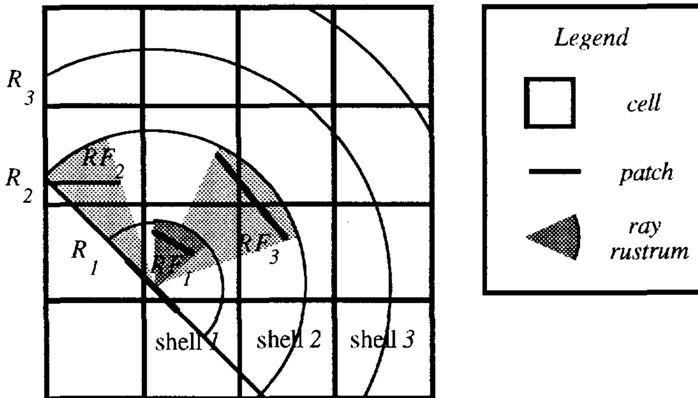


Figure 3.28: Ray-frustum casting in the shelling technique.

In Fig. 3.28, we use the example in Fig. 3.13 to demonstrate the ray-frustum casting. The key point is that ray frustums are now defined by patches instead of shells. The main advantage of this approach is that more parallelism can be exploited as ray frustums in different shells may be executed in parallel. In this example, being free from data dependencies, RF_1 and RF_2 can be executed in parallel. In this approach, the processing of ray frustums is reminiscent of wavefronts swept over entire space. The problems that remain to be answered are:

1. How to determine the patches that define ray frustums?

   The answer is rather involved and more details will be treated in chapter 4. Basically, they should be large patches from the viewpoint of a sample point. Otherwise, they are most probably hidden by intervening patches, representing a waste in computations. In order to find those patches, a low-density ray casting is used as a preprocessing step. Due to the low-density rays, we can say that most patches found in this step are large in some sense. Ray frustums are then built on the basis of those large patches.

2. How to compute ray frustum from a patch?

   By definition, a ray frustum is nothing but a region bounded by six planes in polar coordinate system. It is clear that the $r = r_{max}$ bounding plane can be derived from the

vertices (or control points) of the patch. The four *constant-φ* or *constant-θ* planes can be derived from the patch's *SBB*, as is discussed below.

3.5. A Formal Comparison

The underlying principle of the shelling technique is to establish parallelism that is profitable to exploit in the underlying architecture by virtue of the following dual strategy:

1. Visibility Ordering

   By following a visibility ordering, rays can search for relevant patches in a manner of backward ray tracing, resulting in a highly efficient computational structure as shown in Fig. 3.11. Most redundant parallelism in computations, prevailing in the naive algorithm, becomes to be obsolete.

2. Ray-Frustum Casting

   By grouping rays on the basis of the Spherical Bounding Box (*SBB*) of a patch, a ray frustum is formed and will be served as basis for casting. The implications of the ray-frustum casting on the shelling technique are twofold: (1) It helps in building the visibility ordering on the ray side as irrelevant rays for a patch can be largely screened out by the patch's *SBB*; (2) It is a versatile computational primitive that can mitigate the overhead of long latencies for patch requests.

Due to the inclusion of data-dependent iterations (see Fig. 3.14), the computational structure of the shelling technique is no longer manifest at compile time. Besides, the computational structure of one scene may be very different from that of another scene. All these prevent us from deriving a solution that is optimal in some sense by means of formal techniques like mathematical programming or graph-theoretic method. In this section, we shall make a formal comparison between ray-frustum casting and single-ray casting. To simplify the comparison, the following assumptions are made:

1. First of all, we assume that the three-dimensional object space is partitioned into a shell-like cell structure with respect to a sample point. A patch is stored into cells in which it resides. This allows to derive the space occupied by a shell easily, which may otherwise be rather involved.

2. Secondly, we assume that patches are uniformly and randomly distributed over the space. More precisely, we assume that for arbitrary points (x_1, y_1, z_1) and (x_2, y_2, z_2) in space, and randomly selected patch P, that the probability that P takes up (x_1, y_1, z_1) is equal to the probability that P takes up (x_2, y_2, z_2).

3. Finally, we assume that rays are uniformly distributed in (φ, θ)–plane with constant $\Delta\varphi$

and $\Delta\theta$ (see Fig. 2.4b). As each ray subtends $\Delta\varphi$ and $\Delta\theta$, it can be viewed as a ray frustum taking up a finite space.

For the sake of completeness, we briefly state the single-ray casting and the ray-frustum casting.

1. The single-ray casting

   Each ray within a shell performs intersection tests with all the patches in the space taken up by it sequentially. As a result, a ray may either die there or survive and leave for the next shell. When the latter happens, the ray continues testing as before until it hits a patch.

2. The ray-frustum casting

   All the rays within a shell perform intersection tests with all the patches in the space taken up by them in a pipelined fashion. As a result, a number of rays may die there and only remaining rays allow to leave for the next shell. The remaining rays continue testing as before until no more ray is left.

Consider a sector of space extending from a point on which a shell-like cell structure is built.

Let

- R_i : The number of rays in shell i of the sector; $R_1 = R$.
- P_i : The number of patches in shell i of the sector; $P_1 = P$.
- r : The radius of the first shell of the sector.
- K: Screening factor, that is the fraction of rays screened out in a shell.
- S: Surviving factor, that is the fraction of rays leaving a shell; $S = 1 - K$.
- T_λ: The pipeline period of an Intersection computation unit (ICU).
- T_r: The time spend on retrieving a patch (from a memory and possibly via a network or bus) which is assumed to be much larger than T_λ.

We aim at deriving a formula for the execution time of a sector processing. This derivation is based on the following observations:

1. The number of patches in a region is proportional to its volume (by assumption 2).
2. The screening factors of two shells are the same if the ratio of number of patches is the same as the ratio of densities of rays (i.e., the number of rays per unit volume) in those two shells (by assumptions 2 and 3).

In order to simplify the derivation, we are looking for a shell structure, possibly with variable shell-radius, such that the screening factor for each shell is the same. By observation 2, this can be accomplished by choosing an appropriate radius for each shell to keep the ratio of number of patches the same as the ratio of densities of rays in two consecutive shells. In general, rays become divergent in subsequent shells, an so their densities decrease accordingly. For

simplicity, we take the average density upon entering and leaving a shell as the density of rays in the shell. The ratio of the density of rays in the first shell and the second shell is given by

$$\frac{\frac{r^2}{2}}{\frac{((kr^2) + r^2)}{2}} = \frac{1}{k^2 + 1},$$

where kr is the radius of the second shell.

By observations 1 and 2, the screen factors of the first and second shells are the same if the following is satisfied

$$\frac{r^3}{((kr)^3 - r^3)} = \frac{1}{k^2 + 1}. \quad (3.12)$$

Solving Eq. 3.12, we have

$$k \approx 1.7.$$

By repeating this for the other shells, we derive a shell structure as shown in Fig. 3.29 in which all the screening factors are the same. By observation 1, we can have the number of patches in shell i, for all $i \neq 1$, from the ratio of the volume of shell i to that of the first shell, it follows that

$$P_i \approx i^2 P.$$

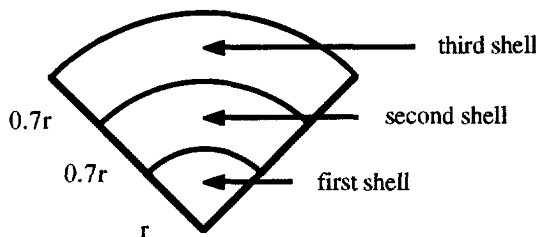


Figure 3.29: A shell structure to keep the screening factor equal.

When neglecting the cell-traversal time, we find the execution time, T_{rf}, for the ray-frustum casting to be

$$T_{rf} = (R\ P + SR\ 2^2\ SP + S^2 R\ 3^2\ S^2 P + ...)\ T_{\lambda}. \tag{3.13}$$

If we assume that a single bundle-patch intersection-time is balanced with the loading time of the next patch.

Now, from Eq. 3.13

$$T_{rf} \approx \frac{(1 + S^2)\ R\ P\ T_{\lambda}}{(1 - S^2)^3} \qquad \text{putting} \qquad C_{rf} = \frac{(1 + S^2)}{(1 - S^2)^3},$$

we get

$$T_{rf} \approx C_{rf} R\ P\ T_{\lambda}. \tag{3.14}$$

In Fig. 3.30, we show the effect of S on C_{rf}. C_{rf} starts increasing slowly when S increases, but goes up very quickly when S approaches 1. This implies that the overall time T_{rf} will increase very fast when the bundle of rays in the sector doesn't match with the ray coherence of patches.

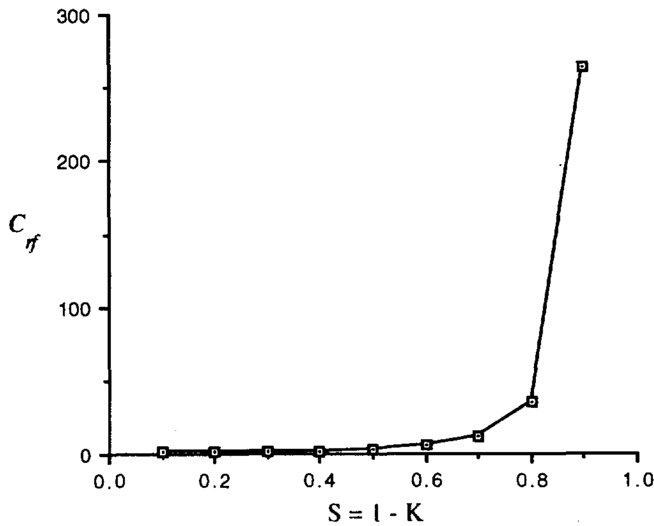


Figure 3.30: The effect of S on C_{rf}.

For comparison purpose, we look at the fraction t_{rf} of T_{rf} spent on shell i. and is given by

$$T_{rf} \approx \begin{cases} S^{2(i-1)} R\, i^2 P\, T_\lambda & \text{if } S^{i-1} R\, T_\lambda > T_r, \\ S^{i-1}\, i^2 P\, T_\lambda & \text{if } S^{i-1} R\, T_\lambda \le T_r. \end{cases} \tag{3.15}$$

In the above formula, the bundle-patch intersection-time may or may not be balanced with the loading time of the next patch.

For the single-ray casting, the time spent on shell i, t_r say, depends on $ADRC_i = KR/i^2P$, that is, the average degree of ray coherence of patches in shell i. In case the patches in the shell can be hit by at most one ray, that is, when $ADRC_i \le 1$, then t_r is determined by the retrieving time of patches in the space taken by the rays. Otherwise, t_r is the sum of two contributions: (1) the part hit by the rays, and (2) the part that remains no hit. Hence

$$t_r = \begin{cases} (S^{i-1} KR + S^i\, i^2 P)\, T_r & \text{if } ADRC_i > 1, \\ (S^{i-1}\, i^2 P)\, T_r & \text{if } ARDC_i \le 1. \end{cases} \tag{3.16}$$

Comparing Eqs. 3.15 and 3.16, we conclude

1. In case $ADRC_i \le 1$, there is no advantage at all from using the shelling technique because

$$\begin{cases} t_{rf} > t_r & \text{if } S^{i-1} R\, T_\lambda > T_r, \\ t_{rf} = t_r & \text{if } S^{i-1} R\, T_\lambda \le T_r. \end{cases} \tag{3.17}$$

2. In case $ADRC_i > 1$, if we can match the bundle of rays in shell i with $ADRC_i$, that is, if

$$S^{i-1} R = \frac{KR}{i^2 P},$$

then

$$t_{rf} = \begin{cases} \left(\dfrac{(S + K\, ADRC_i)\, T_r}{ADRC_i\, T_\lambda} \right) t_r & \text{if } S^{i-1} R\, T_\lambda > T_r, \\ (S + K\, ADRC_i)\, t_r & \text{if } S^{i-1} R\, T_\lambda \le T_r. \end{cases} \tag{3.18}$$

3. By Eq. 3.18, we must keep the screening factor K as high as possible, because in this case $ADRC_i > 1$. In the ideal case, $K = 1$, that is, $S = 0$, in which case

$$t_{rf} = \begin{cases} \dfrac{T_r}{T_\lambda} t_r & \text{if } S^{i-1} R\, T_\lambda > T_r, \\ ADRC_i\, t_r & \text{if } S^{i-1} R\, T_\lambda \le T_r, \end{cases} \tag{3.19}$$

justifies the usage of spherical bounding boxes to force K to be high.

Chapter 4

Mapping onto a Pipelined Parallel Architecture

4.1. Introduction

In order to make the algorithm execution fast on a parallel machine, it is required that the parallelism in the algorithm can be expressed effectively by a powerful programming language and the architecture of the machine should be able to support the parallel execution of the resulting program. For instance, parallelism in an algorithm can be fully expressed by data-flow programming with *Val* [AD79] [McG82] and *Id* [Nik87*a*] [Nik87*b*] and supported by dataflow machines. In recent years, substantial efforts are being exerted on developing parallel programming languages. Different parallel programming paradigms are evaluated. These include shared variables in *parallel Fortran* and *C*, message passing in *CSP/Occam* and in *extended Fortran* and *C* on hypercubes, single assignments in *SISAL* and data-flow programming with *Val* and *Id*. The detailed description is beyond the scope of this thesis. Our emphasis will be on finding an appropriate architecture that supports the parallel execution of an algorithm. This requires to know different styles of parallel computation pursued by a parallel machine. First of all, parallel computation can be characterized as *function-parallel* or *data-parallel* depending on the way it is partitioned and distributed [Ost87]. A function-parallel computation decomposes a program into modules of different functionality, which can be executed in parallel on multiple processors. This is suitable for an algorithm that can be programmed using many independent subroutines, e.g., flight simulation. A data-parallel

computation partitions a data domain into data segments and distributes them among multiple processors. Then each processor works on the data segment assigned to it independently. This is appropriate for an algorithm performing the same set of operations repeatedly and independently on a large set of data. Secondly, parallel computation can be characterized as *parallel execution* or *pipelined execution* depending on how it executes. Parallel execution exploits spatial parallelism by using multiple processors executing independent tasks. Contrastingly, pipelined execution exploits temporal parallelism by overlapping different tasks. To understand this, it requires a bit more explanation. A pipeline consists of a cascade of processing stages. Each stage behaves like a filter, which operates on its input data and passes output data to the succeeding stage. Successive tasks are streamed into the pipeline and get executed in an overlapped fashion. Some important characteristics of pipelined execution are listed as follows:

1. Through pipelining, communication with the external world usually only occurs at the start and end of the pipeline. This can reduce I/O bandwidth for outside communication, which is especially important for a parallel machine that communicates with the external world through a host computer.
2. Pipelining keeps the amount of parallel activity constant (i.e., equals to the number of stages in the pipeline) while significantly reducing the hardware requirement. It offers an economical way to realize parallelism.

The above recognition of parallel computation is very helpful for the practice of architecture design. Now, let's take a closer look at the algorithm that implements the shelling technique. We claim that it belongs to the class of data-parallel algorithms. Consider the most time-consuming computations, that is, intersection computations in the *Shell* algorithm (see Fig. 3.12). It is repeatedly executed for each participating *icray-patch* pair (i.e., for instance, the inputs to each node in Fig. 3.11). Although some dependencies have come into play due to the fact that a ray's behavior in one shell will depend on the computational results of that same ray in previous shells, different rays are still independent. This justifies the above claim.

Next, we shall discuss how to execute the data-parallel algorithm if somehow the partitioned data domain has been distributed among multiple processors. As a general principle, the object space is partitioned into wedge-type partitions as the one shown in Fig. 3.7. Then each partition is assigned to a processor. Computations in different partitions can be executed in parallel, whereas within each partition a pipelined execution is assumed. This leads to a *locally pipelined globally parallel* (*LPGP*) scheme. This choice is based on the following considerations:

1. Due to the property of ray coherence, neighbouring rays most probably intersect with the same patches in the environment. When using multiple processors executing neighboring rays (i.e., *locally parallel*), memory contention may occur due to the fact that multiple processors may attempt to access the same patches. Consequently, the parallelism in computation becomes insignificant because the same patches cannot be fetched

simultaneously by all processors for the lock-step manipulation. To resolve the memory contention, we may execute computations regarding neighbouring rays in a pipelined fashioned. Through pipelining, communication with the external world only occurs at the start and end of the pipeline. At the first time, it is indispensable to retrieving the patch data (i.e., patch geometry information) from the memory system. Upon receiving the patch data, the bundle of so-defined *icray-patch* pairs are streamed into the pipeline and get executed in an overlapped fashion. No more requests for the same patch are necessary. In this way, the parallelism in computation can be best exploited through the use of temporal parallelism because the communication overhead of retrieving a patch can be amortized over many computations.

2. Due to the limited size of a patch, it occupies a contiguous and bounded region in space. This implies that computations in partitions which are not in the geometric neighbourhood of the same patches often incur less memory contention. In case the granularity of a partitioning is not very fine, we can take advantage of spatial parallelism by using multiple processors executing computations in different partitions.

From the above discussion, we conclude that a pipelined parallel architecture is appropriate for the parallel processing of the shelling technique. By operating several pipelines simultaneously, parallel computation can exploit both spatial (multiple processors) and temporal parallelism (pipelines). We shall now come to the resource management problem in order to effect parallel execution.

1. How to partition a program into basic schedulable fragments and to partition the data domain into data segments for optimal execution?

   This is the grain-size[4] problem [Kru88]. We may have the greater opportunity for parallel processing when the size of each program fragment becomes smaller. However, a commensurate increase in the communication overhead due to latency and synchronization may dominate the useful computation and thus slow down the program execution. Such a situation becomes evident in our case. In the previous discussion, if the partitioning is running into the geometric neighbourhood of the same patches, then memory contention occurs due to the fact that multiple processors attempt to access the same patches. Consequently, the parallelism in computation becomes insignificant because the same patches cannot be fetched simultaneously by all processors for the lock-step manipulation. In this chapter, we shall discuss a geometry based grain packing to determine a suitable grain size.

2. How to schedule the program fragments and to allocate the data segments on a parallel machine to obtain the shortest possible execution time?

[4] We define a grain as one or more concurrently executing program fragments. The grain-size is the size of a grain.

This is the scheduling problem, which is known to be NP-complete [GJ79] [ISC84] [KN84], except in a few specific contexts with very unrealistic constraints. Hence various approaches have been proposed that all seek to obtain satisfactory sub-optimal solutions in a reasonable amount of time [Bok81] [Lo88] [MO86] [SE87] [ST85]. There are two basic considerations for the scheduling problem: (1) To which processors should the program fragments and their corresponding data segments be assigned? (2) In what ordering should the program fragments be executed on the assigned processors? In this chapter, we shall discuss a set of heuristics that serves for this purpose.

4.2. Geometry Based Grain Packing

In [KCN90*a*] [KCN90*b*], a *grouping* technique was proposed for balancing computation time and communication time by controlling the granularity through data partitioning and overlapping the operations through pipelining. This approach is limited to a class of data-parallel algorithms with *regular* computational structures[5] which are known at compile-time. The shelling technique leads to a data-parallel algorithm whose computational structure is neither regular nor manifest at compile-time (refer to section 3.4). Nevertheless, the grouping technique in [KCN90*a*] [KCN90*b*] may serve a good starting-point for solving the grain-size problem. We first give the definition of grouping borrowed from [KCN90*b*].

Definition 4.1 (Grouping, Groups, Base Node) Let $Q(N, E)$ be a regular computational structure. The *grouping* $G(Q, (d, s))$ of $Q(N, E)$ along a direction d of size s is to partition the set N into disjoint subsets N_I, $I \in \mathbf{I}$, called *groups*, such that, denoting by $|\ |$ the cardinality of a set

1. $|N_I| = s$, and
2. $\cup_I N_I = N$, and
3. for all $n_J \in N_I$: $J = I + rsd$, $0 \leq r < 1$ and $J \in \mathbf{I}$,

the node n_I is called the *base node* of the group N_I. ■

The result of a grouping can be represented as another computational structure called the *contracted computational structure*.

Definition 4.2 (Contracted Computational Structure) Let $G(Q, (d, s))$ be a grouping. The *contracted computational structure* $Q'(N', E')$ of $Q(N, E)$ with respect to $G(Q, (d, s))$ is a

---

[5] A computational structure is said to be *regular* if it posseses regular dependence vectors.

computational structure, where

1. each node in N' corresponds to one group in the grouping, and
2. each edge in E' corresponds to a dependence constraint between two nodes of Q'. ■

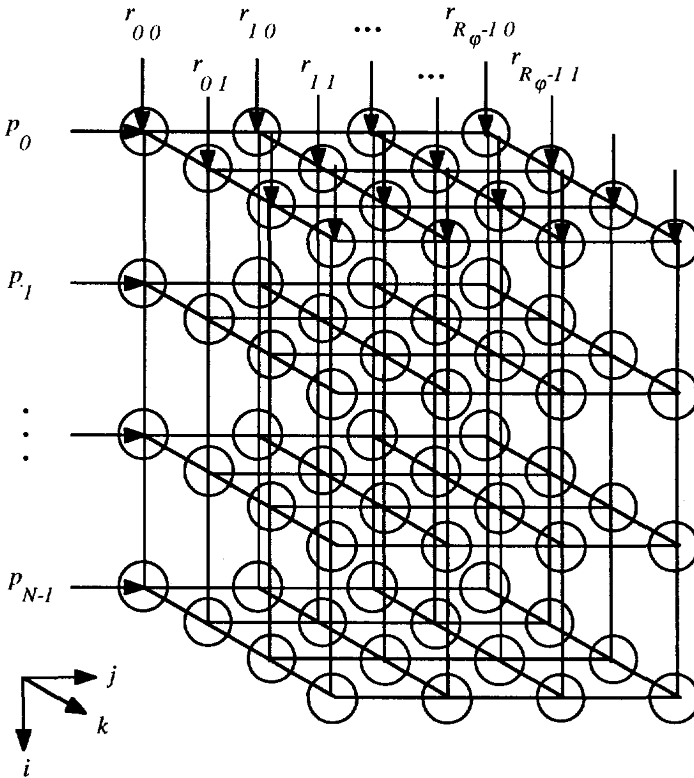


Figure 4.1: The computational structure Q for the naive algorithm.

The computational structure Q of the naive algorithm is shown in Fig. 4.1. Given the computational structure, our goal will be to schedule its execution on a parallel machine. One possible solution is to assign each node in the computational structure to one processor. This requires a way of delivering all patches and all rays to the relevant processors at the same time, so that all processors can perform intersection computations independently and simultaneously. This is clearly not practical due to nature limitations of communication bandwidths. Hence, the expected high execution speed due to the parallelism in computation will be obstructed severely due to communication overhead. To solve this, one may group the computational structure Q

along $[1\ 0\ 0]^t$ direction of size N, that is, the number of patches. The contracted computational structure Q' after grouping is shown in Fig. 4.2. We again assign each node in Q' to one processor. Now only one patch needs to be delivered to all processors simultaneously, bandwidth requirements can be relaxed considerably. However, a processor only performs one intersection computation after receiving one patch and one ray data. The communication overhead of delivering those data would limit the computational rate of a processor. Further grouping might be necessary to reduce that influence. Fig. 4.3 shows the contracted computational structure Q'' of further grouping Q' along $[1\ 0]^t$ and $[0\ 1]^t$ directions of sizes 2 and 2, respectively. By assigning each node in Q'' to one processor, a processor now performs four intersection computations after receiving one patch and four ray data. In other words, a delivery of one patch data can support up to four intersection computations.

In the naive algorithm, the best grain size can be determined by balancing computation time and communication time through grouping because the computation/communication ratio can be controlled by adjusting the size of the groups. For many *NLPs*, the amount of computation in one iteration might not be enough to balance the overhead in communication. In other words, a processor might spend more time in communicating I/O data than in computing the simple iterations. It is thus necessary to control the number of iterations executed between each round of communication through grouping. This a key step leading to the best grain size for an *NLP*. To determine this best size, first of all, it requires a clear model of communication cost which is derived based on the underlying processor interconnection and memory structure.

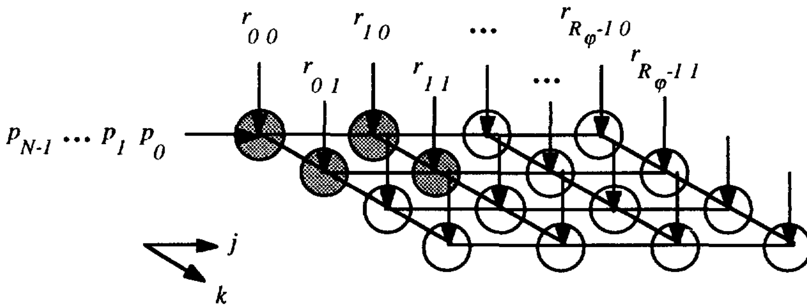


Figure 4.2: The computational structure Q' after the grouping $G(Q, ([1\ 0\ 0]^t, N)$.

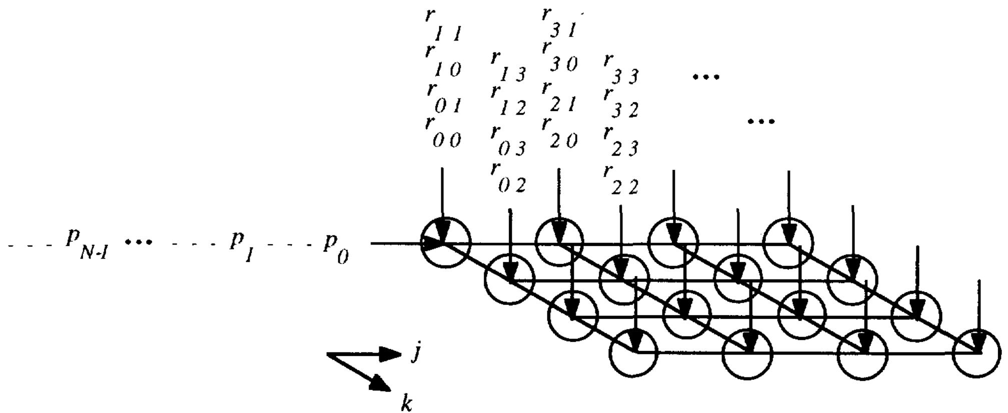


Figure 4.3: The computational structure Q'' after the grouping $G(Q', ([1\ 0]^t, 2), ([0\ 1]^t, 2))$.

Having introduced the concept of grouping by the example of the naive algorithm, we shall now come to the grain-size problem of the shelling technique. Suppose that we have the computational structure for the scene given in Fig. 3.9, and is shown Fig. 4.4 (redrawn from Fig. 3.11). It is very difficult to decide the best grain size by *explicitly* equating (i.e., balancing) computation time with communication time due to the lack of a clear model of communication cost. Except for some specific classes of applications, communication cost cannot be known until program execution time. For instance, latencies in data accesses and synchronizations may vary considerably due to contention in resources. Furthermore, the influence of data partitioning on the communication cost can also be significant. This is because remote memory accesses are much more expensive than local memory accesses in a distributed memory or shared memory with multiple memory hierarchies.

To determine a suitable grain size, we should somehow control the number of iterations executed between each round of communication. Our solution, called *geometry based grain packing*, is to *pack* iterations in such a way that communication cost can be *implicitly* reduced and parallelism in computation can be exploited through pipelining. It is based on the following definition:

Definition 4.3 (Basic Block) A *basic block* is a piece of pipelinable codes which may be entered only at the beginning, and when entered are executed *in a pipelined fashion* to the end of the block without performing any branch statement on exiting the block. ■

By packing intersection computations between a patch and the bundle of *icrays* within the patch's *SBB* into a basic block (the shaded box in Fig. 4.4), communication cost can be implicitly reduced and parallelism in computation can be exploited through pipelining. This can

be understood as follows. The patch data must be retrieved from the memory system at the first time. Upon receiving the patch data, the bundle of so-defined *icray-patch* pairs are streamed into the pipeline and get executed in an overlapped fashion. No more requests for the same patch are necessary. The parallelism in computation can be best exploited through the use of temporal parallelism because the communication overhead of retrieving a patch can be amortized over many computations.

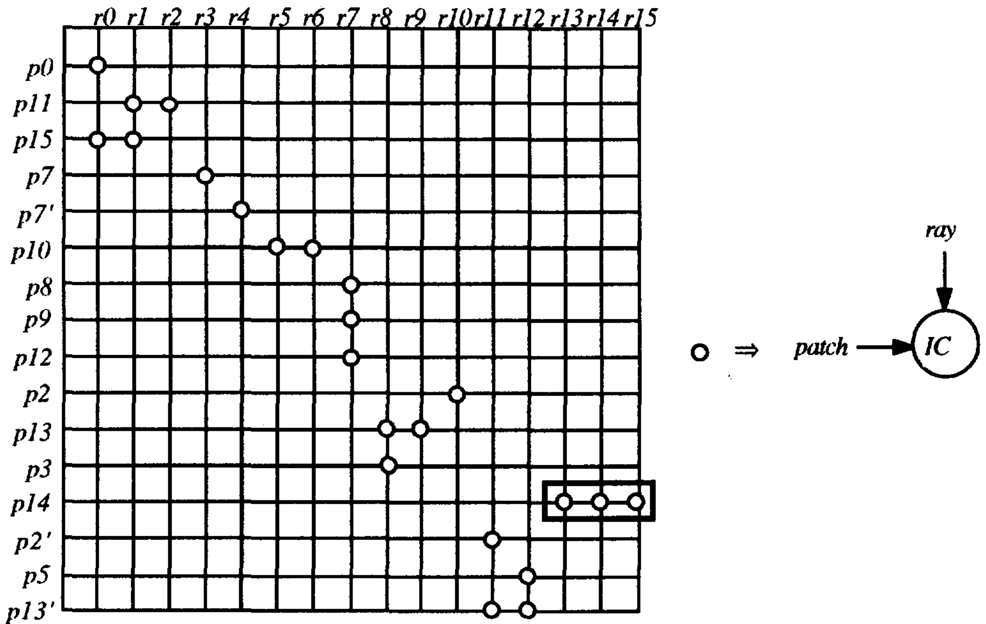


Figure 4.4: An example computational structure of the shelling technique.

4.3. Runtime Scheduling

The general scheduling problem is a resource management problem that determines a policy used to manage the access to and use of a resource by its various consumers. To obtain a high-quality policy, one requires detailed information regarding the resource and consumers at the time of scheduling. However, except for some specific classes of applications, most of those

details in the practical scheduling problem cannot be known until program execution time. For instance, a highly data-dependent program whose conditional constructs like branches varies drastically from one set of input data to another. Latencies in memory accesses, communications and synchronizations may vary considerably due to conflicts in resources. Furthermore, data partitioning in a multiprocessor system can be very important in the context of scheduling. This is because remote memory accesses are much more expensive than local memory accesses in a multiprocessor system with distributed memory or shared memory with multiple memory hierarchies. These dynamic behaviours make the scheduling problem very difficult to manage and analyze at compile time. In contrast to compile-time scheduling, runtime scheduling allows to manage resources for highly data-dependent programs and dynamic system environments. However, a major shortcoming of runtime scheduling is that a non-negligible runtime overhead will be introduced. To explain this, we first discuss several key issues regarding the collection and utilization of runtime information in runtime scheduling.

1. What is runtime information?

   The runtime information consists of dynamic program informations and dynamic system information. The dynamic program information can vary from rather rough characterization like the number of processes, or the average execution time of a process and the total amount of interprocess communications, to fine details like control and data dependencies among processes. The dynamic system information includes resource availability, resource utilization and workload distribution in the system environment. In this work, dynamic program information is estimated to a level that allows us to use the clustering technique described above. This requires knowledge of both the workload of each process and the total amount of interprocess communication between each process pair.

2. How to obtain runtime information?

   Obviously, dynamic program information can only be generated from the program itself. It can be generated by some runtime directives from the compiler/programmer or some runtime routines provided by compiler/programmer. In this work, we rely on the runtime information captured by some runtime routines which are well-tuned for our specific application.

3. How to utilize runtime information?

   This is actually the runtime scheduling problem, i.e., how to utilize runtime information to determine a better scheduling. As the scheduling algorithm is executed during the runtime of the application program, its execution time must be kept small as compared with the execution time of the scheduled application program. This leads to an unusual scheduling problem on which a runtime constraint is imposed. A sophisticated scheduling algorithm gives a superior scheduling, but the overall performance may be significantly bad due to its high overhead. One may choose a low overhead but less effective scheduling algorithm to obtain a better overall performance.

The above discussion makes clear that major runtime overheads in runtime scheduling are:

1. Runtime information gathering

   Runtime scheduling makes use of the runtime information gathered by some runtime routines which are well-tuned for a specific application. Intuitively, the more detailed and accurate the available information, the better scheduling can be obtained. However, detailed and accurate information also means more expensive to gather.

2. Scheduling

   As the scheduling algorithm is executed during the runtime of the application program, its execution time can incur considerable runtime overhead. In order to eventually gain in overall performance, seemingly time-consuming or time-indeterminate algorithms are not feasible even though they are sophisticated and may lead to a better solution in case of compile-time scheduling. This limits feasible scheduling algorithms to those with low complexity and highly efficient implementation.

4.4. Application-Specific Runtime Scheduling

We aim at providing a hardware accelerator board tailored for fast image rendering based on ray tracing and radiosity shading. By tailoring the characteristics of this specific application, we developed a technique called *application-specific runtime scheduling* (*ASRS*). It is application-specific in the sense that only a class of applications which are ray-casting based, can be applied. In general, runtime resource scheduling can be totally managed by the operating system or directed by the compiler. However, operating system management can only be based on system observables like the number of ready processes, past resource utilization, etc. It is difficult to achieve a good performance for lack of detailed information about the program. Compiler-directed runtime resource management is a combination of compilation techniques and runtime system techniques, which is quite interesting for a general purpose machine oriented towards a wide range of applications. For our purpose, an application-specific approach is more appropriate as runtime scheduling routines can be well-tuned for the specific application, which leads to a low complexity and highly efficient implementation. This is justified by the following considerations:

1. We can view a program execution as many serial-parallel execution *phases* probably intermingled together. Within each phase, the dynamic program information is relatively stable, but the dynamic program information varies drastically from phase to phase. Intuitively, the longer each stable phase, the more runtime overhead can be afforded for runtime scheduling. For this reason, we organize the execution of a number of ray-casting procedures pertaining to neighbouring sample points in the radiosity pass as a phase. In the

ray-tracing pass, we organize the execution of a number of ray-casting procedures pertaining to patch-based intersection points as a phase. This leads to a proximity enforced algorithm as discussed in section 2.5. In this way, runtime overhead can be amortized over the entire execution of the phase. Furthermore, the data management overhead coming from dynamic data allocation and access can be reduced considerably as data coherence can be exploited to a great extent.

2. Within each phase, we rely on a low-density ray casting to estimate the dynamic program information required for the scheduling of high-density ray casting. The overhead of runtime information gathering is negligible because the number of low-density rays is much less than that of high-density rays.

3. All the heuristics in the application-specific runtime scheduling are quite simple but efficient. This choice is in accordance with the requirements of low complexity and highly efficient for feasible scheduling algorithms as stated previously. For instance, the clustering technique is quite efficient as far as both balancing workload and reducing interprocessor communication are taken into account. It is quite simple: on the one hand the workload estimates are as simple as the lump-sum aggregates regarding cell-traversal and intersection-computation counts; on the other hand interprocessor communication can be implicitly reduced and workload can be explicitly balanced by means of packing and partitioning.

4.4.1. Basic Terms

The shelling technique leads to an algorithm whose computational structure is neither regular nor manifest at compile-time. It thus cannot take advantage of the *linear space-time mapping* used in systolic array design and the *grouping* used for regular computational structures to schedule its execution on a parallel machine. Moreover, a basic block becomes the basic unit of execution and allocation. In view of this, there is a clear need to use a different graph-theoretic model to schedule basic blocks on a parallel machine.

A ray-frustum process is defined as all the computations regarding cell traversal and intersection computation for a set of rays in a ray frustum. Fig. 4.5 shows the flowgraph of a ray-frustum process. It consists of two different computational blocks *CT* and *IC* possibly connected by two different kinds of edges. Edges with arc represent data communication requirements, while edges without arc represent the intended ordering for execution. For instance, the edge from a *CT* block to an *IC* block represents the requirement of sending rays and patches for intersection computation. However, the edge connecting two *CT* blocks stands for an intended ordering for execution. We intend to execute the first *CT* block and then the second *CT* block. Each of the *CT* and *IC* blocks is associated with a weight that represents the amount of cell-traversal and intersection computation in terms of operation counts, respectively. Each edge with arc is associated with a weight that represents the amount of communication required in terms of message length. A ray-frustum process may have many different representations. Fig. 4.5

demonstrates two possible representations for the same ray-frustum process *rf1*. By following a depth-first order, a patch will be found and contribute *6* intersection computations after traversing *2* cells. In addition, there are *8* cells that remain to be traversed. This results in the left flowgraph shown in Fig. 4.5. Similarly, we can work out the right flowgraph in Fig. 4.5 by following a breadth-first order. It is seen that different ways of searching may result in different cell-traversal and intersection-computation times. A set of ray-frustum processes can be represented as a *block graph.*

Definition 4.4 (Block Graph) Let RF be a set of ray-frustum processes. The *block graph* of RF is a directed graph $G(N, E)$, where

1. $N = \cup_{rf \in RF} (B_{ct,rf} \cup B_{ic,rf})$ is a set of nodes, where $B_{ct,rf}$ and $B_{ic,rf}$ are a set of CT and IC blocks in the ray-frustum process rf, respectively.
2. $E = O \cup D$ is a set of edges, where $O = \cup_{rf \in RF} (O_{ct,rf} \cup O_{ic,rf})$ is a set of edges representing the linear orders of blocks in the ray-frustum processes and $D = \{(b_{p,i}, b_{q,j}) \mid p, q \in N \text{ and } p \neq q\}$ is a set of edges representing data communication requirements. Each block in $O_{ct,rf}$ and $O_{ic,rf}$ represent a set of linear orders of CT and IC blocks in the ray-frustum process rf, respectively.
3. Each block in $B_{ct,rf}$ is associated with a weight that represents the amount of cell-traversal computation in the ray-frustum process rf and each block in $B_{ic,rf}$ is associated with a weight that represents the amount of intersection computation in the ray-frustum process rf.
4. Each edge in D is associated with a weight that represents the amount of communication required between two blocks. ■

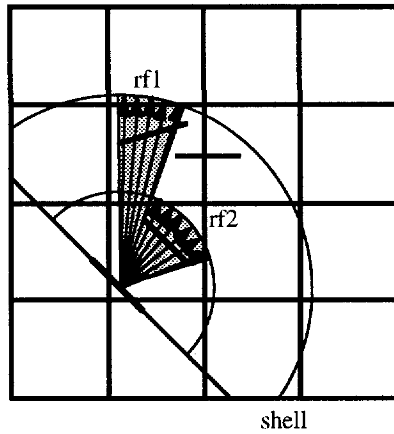
As pointed out in the above, a ray-frustum process may have many different representations. It is the scheduler that determines which one should be taken. It is thus not possible to characterize a block graph before a solution, that is, a *schedule*, is attempted. Moreover, a block graph may change from scene to scene. It is therefore useless to derive the block graph for a particular case. In this work, we make use of virtual block graphs to lead the way to the solution algorithm. To facilitate our further discussion, we define some basic terms in the following:

Definition 4.5 (Execution Profile) Let $G(N, E)$ be a block graph. The *execution profile* of $G(N, E)$ is a space-time diagram showing its actual execution. ■

Definition 4.6 (Intrinsic Execution Profile) Let $G(N, E)$ be a block graph. The *intrinsic execution profile* of $G(N, E)$ is an execution profile with zero communication time. ■

Definition 4.7 (Ideal Execution Profile) Let $G(N, E)$ be a block graph. The *ideal execution profile* of $G(N, E)$ is a space-time diagram showing its actual execution when: (1)

Minimum cell traversal and minimum intersection computation are assumed for each ray-frustum process; and (2) All communication is neglected. ■



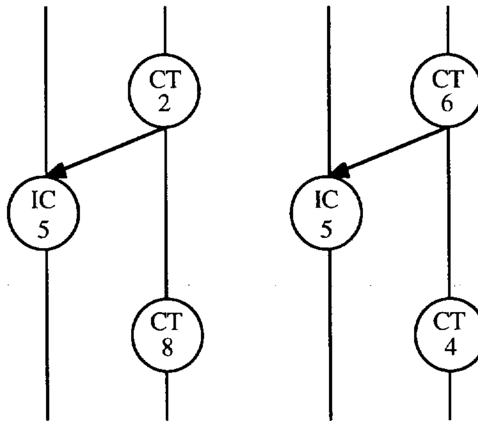
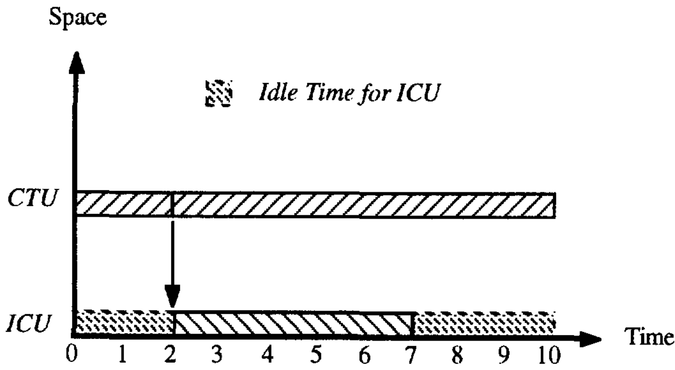


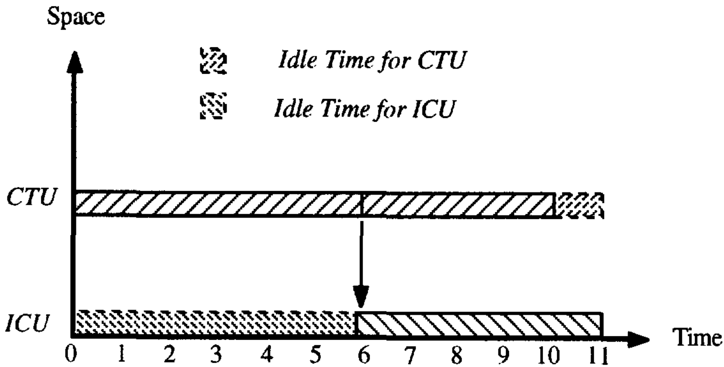
Figure 4.5: Two possible representations for the ray-frustum process *rf1*.

As an illustration, consider the example of Fig. 4.5 again. Two different intrinsic execution profiles for the ray-frustum process *rf1* are shown in Fig. 4.6. In this case, the block graph is executed by two processors: one cell-traversal unit (*CTU*) and one intersection-computation unit (*ICU*). *CTU* and *ICU* are responsible for the execution of *CT* and *IC* blocks, respectively. Due to data communication requirements (the arc in the figure) and different amount of cell-traversal and intersection computations, a processor might be idle for a portion of time during its execution. In the former case, the idle time for a processor can be interpreted as the time

waiting for some messages. For instance, the *ICU* has been idle for 2 or 6 time units while waiting for ray and patch messages. We shall call this the *message idle time* of the arc. Fig. 4.6 also shows the idle times for *CTU* and *ICU*. In this simple example, it is seen that the depth-search strategy is better than the breadth-first strategy. This is justified by the idle times for *CTU* and *ICU*.



(a) Depth-First Strategy



(b) Breadth-First Strategy

Figure 4.6: The intrinsic execution profiles for the example of Fig. 4.5.

4.4.2. Tackling the Scheduling Problem: Some Heuristics

As stated previously, we rely on *ASRS* that can handle dynamic and irregular computations in the shelling technique. A good schedule can be obtained through utilizing the runtime information gathered while keeping the runtime overhead sufficiently low. In order to loosely conform to the runtime constraint in the runtime scheduling problem, seemingly time-consuming and time-indeterminate algorithms have been rejected as feasible solutions even though this may not be conductive to optimality. Within the realm of suboptimal solutions, it is not clear whether efficient approximation algorithms exist. It is thus necessary to use a heuristic approach through empirical evaluation. In looking for a good and efficient heuristic, it is instructive to investigate some existing heuristics that are commonly used to tackle the scheduling problem. They include the following:

1. System Load Balancing

   Processes are assigned such that each available processor has the same number of processes.

2. Load Balancing

   Processes are assigned such that each available processor has equal workload.

3. Clustering

   Processes are assigned such that each available processor has roughly equal workload and the amount of interprocessor communication is minimized.

These heuristics use program information of various degrees of detail, revealing different algorithmic complexities. Consider the case of the shelling technique. System Load Balancing is the simplest as only the number of ray-frustum processes is required. Clustering requires a substantial amount of program information regarding the workload of each ray-frustum process and the total amount of communication between each ray-frustum process pair, and has the highest complexity among the three. In between the two lies Load Balancing that requires lump-sum aggregate regarding the workload of each ray-frustum process. Obviously, System Load Balancing is the fastest among the three as far as the runtime constraint is concerned. However, it is difficult to achieve satisfactory performance if only the number of ray-frustum processes is available in scheduling. As a matter of fact, two ray-frustum processes may differ in workloads representing cell traversal and intersection computation considerably. In [SE90], it is suggested to decouple the reduction of communication from workload balancing so that Clustering becomes quite simple but efficient. It is therefore more satisfactory to choose Clustering as the basis of subsequent evaluation.

Strictly speaking, simple lump-sum aggregates regarding the amount of computation and communication for each ray-frustum process are not sufficient to guarantee a good schedule. This is because precedence relations among ray-frustum processes have been ignored in scheduling. As an illustration, consider the block graph in Fig. 4.7. To simplify the discussion,

the communication overhead of sending messages has been neglected. The use of workload balancing strategy will assign the ray-frustum processes *rf2* and *rf3* to *CTU1* (35 cell-traversal computations) and *ICU1* (35 intersection computations), and the ray-frustum process *rf1* to *CTU2* (25 cell-traversal computations) and *ICU2* (25 intersection computations). Fig. 4.8 shows the execution profile of this schedule whose total execution time is 18% longer than that of the optimal execution profile shown in Fig. 4.9. The optimal execution profile is obtained by assigning the ray-frustum processes *rf1* and *rf2* to *CTU1* (40 cell-traversal computations) and *ICU1* (40 intersection computations), and the ray-frustum process *rf3* to *CTU2* (20 cell-traversal computations) and *ICU2* (20 intersection computations). This is due to the long message idle time on edge *d* of the ray-frustum process *rf3* that causes *ICU1* to be idle for a long time. This suggests that a good scheduling should take message idle time into account and it is better to overlap this idle period with other useful computations.

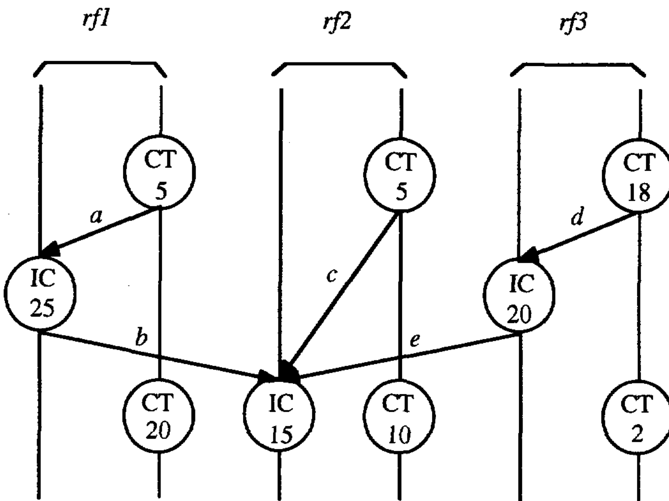


Figure 4.7: An example block graph.

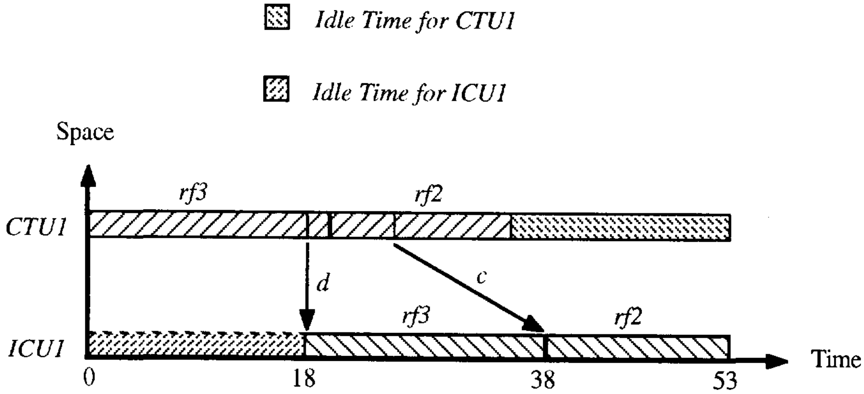


Figure 4.8: Execution profile using workload balancing strategy.

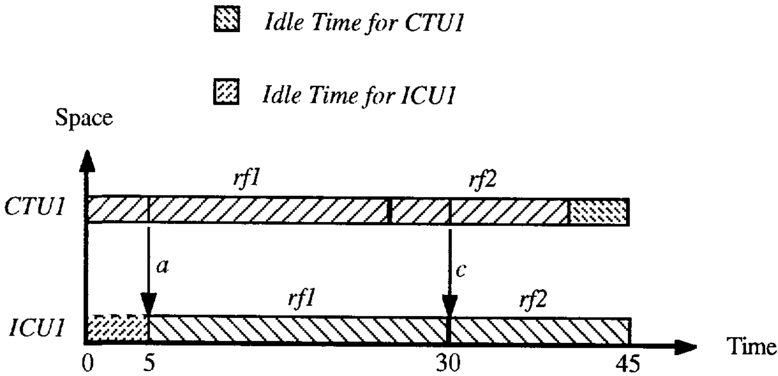


Figure 4.9: Execution profile under optimal scheduling.

It is very hard to achieve the optimal execution profile for a block graph consisting of a huge number of ray-frustum processes. The potential difficulties are the following: (1) It is unfeasible to characterize all ray-frustum processes which are probably intermingled together due to the runtime constraint; (2) The optimal execution profile requires an expensive global search to minimize idle times; and (3) The intervention of non-negligible communication overhead makes the situation even worse. Instead of pursuing this unlikely solution, we make one bold assumption, that is, message idle times can be perfectly overlapped with useful computations as more and more ray-frustum processes are packed and assigned to a processor. In looking for a packing scheme that supports this assumption, we make the following

observation. Due to the limited size of a patch, it occupies a contiguous and bounded region in space. This implies that only neighbouring ray-frustum processes will incur memory contention. This suggests to pack neighbouring ray-frustum processes into higher-level ray frustum in order to reduce communication overhead.

This leads to a *geometry based grain packing*. Conceptually, it can be viewed as a two-level grain packing. The low-level packing is done by defining ray frustums from patches as before. Neighbouring ray frustums can be further packed into a higher-level ray frustum, and is recognized as the high-level packing. As more and more neighbouring ray-frustum processes are packed, we may have the greater opportunity for overlapping message idles times with useful computations. After further packing, the scheduling problem becomes more tractable: (1) It is feasible to characterize all high-level ray-frustum processes; (2) It might be possible to search the large-grain block graph for the optimal execution profile; (3) The influence of communication overhead on scheduling becomes insignificant.

Due to the high complexity of the scheduling problem, *ASRS* breaks down the scheduling problem into four simpler subproblems and tackles each of them separately. First of all, we glance briefly through these subproblems. Next these subproblems will be outlined in more detail, and their implementation aspects will be discussed.

1. Runtime Information Gathering

   This is a preprocessing step that is used to prepare all necessary information for scheduling. At the beginning of each phase, a low-density ray casting is invoked to gather runtime information. These include: (1) workloads regarding cell-traversal and intersection computations, (2) cells and patches found in the low-density ray casting and (3) a list of ordered patches.

2. Clustering

   Based on the workloads regarding cell-traversal and intersection computations, ray-frustum processes are packed and partitioned into clusters in order to reduce intercluster communication and balance workloads. Accordingly, cells and patches found in the low-density ray casting are classified into their corresponding clusters. On each cluster, patches are given descending priorities according to the order of finding them in the low-density ray casting.

3. Assignment

   The assignment algorithm assigns the clusters formed in the clustering step to a network of processors. Meanwhile, the geometry information regarding cells and patches found in the low-density ray casting are distributed and pre-loaded to the local memories of their corresponding processors.

4. Local Scheduling

   Based on the priorities of patches set in the clustering step, the local scheduling algorithm defines local processes and determines their ordering for execution.

4.4.2.1. Runtime Information Gathering

In order to obtain a low complexity and highly efficient implementation for runtime scheduling, we adopt an application-specific heuristic in which runtime routines have been well-tuned for the specific application. To ensure performance gains, runtime information captured must be essential and the execution time of runtime routines must be kept small as compared with that of the application program. This largely depends on the scheduling technique being used and the program characteristics. *ASRS* consists of a set of heuristics that requires the program information regarding the workloads of each ray-frustum process. With regard to the program characteristics, it can be seen from Fig. 3.12 that *cell traversal* and *intersection computation* are the two most time-consuming computations in the shelling technique. It becomes clear that the runtime information should include workloads representing cell-traversal and intersection computations for each ray-frustum process. The question now is how can we have workload estimates without going through an expensive execution.

Due to the property of ray coherence, neighbouring rays will most probably traverse the same cells and even hit the same patches. It would therefore seem that workloads representing cell-traversal and intersection computations gathered by low-density rays may give us a good estimate of those for high-density rays unless the number of low-density rays is rather low.

The pseudo-code of the low-density ray casting is similar to the one given in Fig. 3.12. The main difference lies in: (1) the use of initial partitioning, (2) the resolution of rays, and (3) the declaration of ray hit.

1. As mentioned previously, the geometry based grain packing is essential to *ASRS*. Here we use it to reduce the runtime information required for scheduling. In the radiosity pass, the space is initially partitioned into a number of virtual ray frustums with equally spaced $\Delta\theta$s and $\Delta\varphi$s. As for the ray-tracing pass, ray frustums are by nature defined by limited number of light sources and/or specular surfaces. Instead of characterizing each individual ray frustum, runtime information is gathered with respect to those virtual ray frustums.

2. The low-density rays can be viewed as a subsampling of the high-density rays. For convenience, we denote *Hi_Low_Ratio* as the proportionality of the number of rays cast in the high-density and the low-density ray castings. To ensure performance gains, *Hi_Low_Ratio* must be kept as large as possible (so long as runtime information gathered is still meaningful).

3. Instead of going through elaborate intersection computation, all the rays within the *SBB* of a patch will be declared as hit. In this way, the low-density ray casting can be performed

on a machine without any dedicated intersection-computation hardware. This is of prime importance to the performance of our target architecture built around a rather cheap platform with limited computing power.

4.4.2.2. Clustering

The second and the most important subproblem in *ASRS* is clustering. Due to the conflicts of reducing intercluster communication with balancing workload, algorithms that try to simultaneously achieve both objectives turn out to be unfeasible. In contrast, our strategy is to explicitly attempt workload balancing among clusters, whereas the objective of reducing intercluster communication is achieved implicitly through the use of a geometry based grain packing. With this approach, the reduction of intercluster communication can be largely decoupled from workload balancing, which leads to a simple but efficient solution algorithm.

1. Grain Packing

   Due to the limited size of a patch, it occupies a contiguous and bounded region in space. This implies that only neighbouring ray-frustum processes will incur the intercluster communication if they are not packed into the same cluster. This suggests to pack neighbouring ray-frustum processes into clusters in order to reduce intercluster communication.

2. Partitioning

   After low-density ray casting, the lump-sum aggregates regarding workload estimates for some representative ray-frustum processes become available. Based on those workload estimates, virtual ray frustums are packed into clusters in such a way that workloads can be evenly distributed over clusters. We distinguish between two different types of clusters as there exist two different subprocesses in a ray-frustum process. A sector is a cluster referring to a cell-traversal subprocess while a section is a cluster referring to an intersection-computation subprocess. We now state workload balancing more precisely by saying that the workloads representing intersection computation for one section should be close to that of other sections and also close to the workloads representing cell traversal for each sector belonging to this section. For a given number of processors, in general we can use a *bin-packing* or *BSP* algorithm to balance workloads distributed over clusters. The latter is more preferable due to the following advantages: (1) the resulting clusters can be easily assigned to a network of processors with hypercube or mesh topology, and (2) any intermediate imbalance can become smooth afterwards without redoing the entire partitioning. *BSP* is a recursive bisection based on assigning a scalar field value to each partition. In our case, the scalar field value may represent lump-sum aggregate regarding workload estimate for cell traversal or intersection computation. Bisection then consists of partitioning a cluster into two clusters with median field value. In our case, bisection

consists of partitioning a cluster into two clusters with median workload estimate by taking alternatively constant-φ and constant-θ partition planes.

The *BSP* algorithm for building sections is given in Fig. 4.10, and is explained as follows:

1. In the low-density ray casting, an initial partitioning is used to gather runtime information. Let N_φ and N_θ be the number of partitions along φ and θ directions, respectively. Then, a set of initial clusters (i.e., virtual ray frustums) $C = \cup c_{ij}$, $i = 1, .., N_\varphi$ and $j = 1, .., N_\theta$, is formed. After the low-density ray casting, each cluster c_{ij} is associated with a workload wl_{ij} regarding the total number of intersection computations in the cluster. In addition, the number of processors N, preferably a positive power-of-2 number, serves as an input that determines the maximum level of bisection.
2. For each bisection level, say s, in *BSP*, 2^s intermediate parts, denoted as S_k, $k = 2^s - 1, .., 1, 0$, need to be bisected. Each intermediate part S_k will be bisected into two parts, C_L and C_H, each with approximately median workload.

Algorithm: ***BSP***

Input:

1. A set of initial clusters $C = \cup c_{ij}$, $i = 1, .., N_\varphi$ and $j = 1, .., N_\theta$.
2. A non-negative real number wl_{ij} representing the workload for a cluster c_{ij}.
3. A positive power-of-2 number N denoting the number of processors.

Output:

1. A set of N Sections.

Algorithm:

1. Create a set of N empty Sections $S = \cup S_n$:

$$S_n = \emptyset \text{ and } WL_n = 0,\ n = 0, .., N - 1.$$

2. Set S_0 and WL_0 to the set of initial clusters and its associated workload, respectively:

$$S_0 = C \text{ and } WL_0 = \sum wl_{ij}\ .$$

3. Set bisection direction $dir = 0$.
4. Set bisection level $s = 0$.
5. Repeat the following steps until $s = log_2\ N - 1$:

For each Section $S_k \in S$, $k = 2^s - 1, .., 1, 0$:

$S_k = \cup c_{ij}$, $i = i_l, .., i_h$ and $j = j_l, .., j_h$.

If *dir* is equal to *0*

Partition S_k along i direction into two parts, C_L and C_H, each with approximately median workload:

$C_L = \cup c_{ij}$, $i = i_l, .., i_m$ and $j = j_l, .., j_h$

$C_H = \cup c_{ij}$, $i = i_m, .., i_h$ and $j = j_l, .., j_h$.

Assign C_L and its associated workload to S_{2k} and WL_{2k}, respectively.

Assign C_H and its associated workload to S_{2k+1} and WL_{2k+1}, respectively.

Change bisection direction: $dir = 1$.

Increment bisection level by 1: $s = s + 1$.

Otherwise

Partition S_k along j direction into two parts, C_L and C_H, each with approximately median workload:

$C_L = \cup c_{ij}$, $i = i_l, .., i_h$ and $j = j_l, .., j_m$

$C_H = \cup c_{ij}$, $i = i_l, .., i_h$ and $j = j_m, .., j_h$.

Assign C_L and its associated workload to S_{2k} and WL_{2k}, respectively.

Assign C_H and its associated workload to S_{2k+1} and WL_{2k+1}, respectively.

Change bisection direction: $dir = 0$.

Increment bisection level by 1: $l = l + 1$.

6. Output all Sections, S_n, $n = 0, .., N - 1$.

Figure 4.10: The *BSP* algorithm for building sections.

In accompany with *BSP*, the problem data domain is partitioned into data segments accordingly. A consequence of partitioning the problem data domain is the decomposition of the address space: a data item is no longer referenced by one global address, but by the couple (*processor ID, local address*). We use a simple example to explain this. In an array, indices constitute a unique reference to an element. If the array is partitioned into blocks which are distributed over the processors in a regular way. The local indices within the block can be computed from the global indices and knowledge about the distribution. As we shall see, a simple algorithm is used to assign clusters to processors on a k-ary n-cube network[6]. Thus, the *processor ID* of a data can be easily determined if we can classify to which part the data belongs during *BSP*. We shall discuss this data classification in the following.

In the shelling technique, we have three major data sets, e.g. rays, cells and patches. We shall concentrate on classifying cells and patches into their corresponding clusters because the set of rays belonging to a cluster can be trivially determined from their defining sections/sectors. A precise solution would be to test cells and patches against each partition plane during *BSP*. Special care must be taken when they extend over a partition plane. This is done by comparing the portions of cells or patches in both parts and taking the one where the major portion resides. Due to the runtime constraint, it would be more appropriate to classify cells and patches by simply referring to their corresponding clusters. More precisely, cells and patches belonging to a cluster will be classified to one of the two parts in which the cluster lies. Only when a cluster is cut through by the partition plane, a precise test is used for all the cells and patches belonging to the cluster. This is done by comparing the portions of their *SBBs* in both parts and taking the one where the major portion resides.

The patch classification algorithm embedded in *BSP* is given in Fig. 4.11. For simplicity, we only show the modified step 5 in Fig. 4.10. We shall explain this in the following:

1. In low-density ray casting, an initial partitioning is used to gather runtime information. Let N_φ and N_θ be the number of partitions along φ and θ directions, respectively. Then, a set of initial clusters $C = \cup c_{ij}$, $i = 1, .., N_\varphi$ and $j = 1, .., N_\theta$, is formed. Let $R = \cup r_{ij}$, $i = 1, .., N_\varphi$ and $j = 1, .., N_\theta$, be a set of registers, in which each register $r_{ij} \in R$ corresponds to a cluster c_{ij}. During the low-density ray casting, a patch found in cluster c_{ij} will be stored into the corresponding register r_{ij}. This set of registers R storing patches found in the low-density ray casting will serve as the input to the patch classification algorithm.

2. For each bisection level, say s, in *BSP*, 2^s intermediate parts, denoted as S_k, $k = 2^s - 1, .., 1, 0$, need to be bisected. As explained before, each intermediate part S_k will be bisected into two parts, C_L and C_H, each with approximately median workload.

3. The *cluster IDs* of patches stored in the corresponding registers of S_k will be set according

[6] A k-ary n-cube network is a network with cubes of n dimensions and k nodes in each dimension. Note that rings, meshes and hypercubes all fall into this network.

to:

a. The partition plane doesn't cut through clusters.

In this case, the *cluster IDs* of patches stored in registers whose corresponding clusters belonging to C_L or C_H will be set to $2k$ or $2k + 1$, respectively.

b. The partition plane cuts through clusters.

In this case, the *cluster IDs* of patches stored in registers corresponding to those clusters will be determined by comparing the portions of their *SBBs* in C_L and C_H. If the major portion resides in C_L, then their *cluster IDs* will be set to $2k$. Otherwise, the *cluster IDs* will be set to $2k + 1$.

Algorithm: *Patch Classification*

Input

1. A set of registers $R = \cup r_{ij}$, $i = 1, .., N_\varphi$ and $j = 1, .., N_\theta$, in which each register r_{ij} $\in R$ stores the patches belonging to c_{ij} during the low-density ray casting.

Output

1. The set of registers R storing patches with their *cluster IDs* set.

Algorithm

5. Repeat the following steps until $s = log_2\ N - 1$:

For each Section $S_k \in S$, $k = 2^s - 1, .., 1, 0$

$$S_k = \cup c_{ij},\ i = i_l, .., i_h \text{ and } j = j_l, .., j_h$$

If *dir* is equal to 0

Partition S_k along i direction into two parts, C_L and C_H, each with approximately median workload:

$$C_L = \cup c_{ij},\ i = i_l, .., i_m \text{ and } j = j_l, .., j_h$$

$$C_H = \cup c_{ij},\ i = i_m, .., i_h \text{ and } j = j_l, .., j_h$$

For each $c_{ij} \in C_L$

If cluster c_{ij} is cut through by the partition plane, i.e., $i = i_m$

For each patch p stored in register r_{ij}

Determine the portions of the *SBB* of patch p in both parts

If the portion of the *SBB* of patch p in C_L dominates

Set the *cluster ID* of patch p to $2k$

Otherwise

Set the *cluster ID* of patch p to $2k + 1$

Otherwise

For each patch p stored in register r_{ij}

Set the *cluster ID* of patch p to $2k$

For each $c_{ij} \in C_H$

If cluster c_{ij} is cut through by the partition plane, i.e., $i = i_m$

For each patch p stored in register r_{ij}

Determine the portions of the *SBB* of patch p in both parts

If the portion of the *SBB* of patch p in C_L dominates

Set the *cluster ID* of patch p to $2k$

Otherwise

Set the *cluster ID* of patch p to $2k + 1$

Otherwise

For each patch p stored in register r_{ij}

Set the *cluster ID* of patch p to $2k + 1$

Figure 4.11: The patch classification algorithm.

Besides the *processor ID*, the *local address* of a patch needs to be determined in order to have a complete *foreign pointer*. Normally, the *local address* of a patch is a unique sequence number

on each cluster. In our case, patches on each cluster are given descending priorities according to the order of finding them in the low-density ray casting.

4.4.2.3. Assignment

The third subproblem in *ASRS* is assignment. After clustering, the assignment algorithm assigns the clusters of processes to a network of processors in such a way that the total amount of communication is low. Meanwhile, the partitioned data segments are distributed and pre-loaded to the local memories of their corresponding processors. The assignment problem belongs to a class of difficult combinatorial optimization problems known as the Quadratic Assignment Problem. In looking for a good and efficient assignment algorithm, we investigated the following two strategies:

1. Heaviest-to-Nearest Strategy

   A good assignment algorithm is based on a *heaviest-to-nearest* strategy. The idea behind this approach is that if the heaviest communicating cluster pairs are assigned to the nearest (i.e., physically the closest) processors whenever possible, then the total amount of communication should be low. This approach gives a good solution but at the expense of a rather large execution time. The heaviest-to-nearest algorithm is given in Fig. 4.12.

2. A Simple Strategy

   A simple strategy is to assign clusters directly to a network of processors without regarding the amount of intercluster communication. As described above, the clustering is done by a *BSP* technique. Since a partitioning of depth n provides 2^n clusters, it is convenient to assign those clusters directly to processors on a network with a k-ary n-cube network. Due to the property of object coherence, a patch can only extends over a bounded region in space. In case the granularity of a partitioning is not very fine, the traffic patterns can be highly localized because most traffic requirements are issued to only geometric neighbourhood. As a result, this simple approach can loosely conform to the heaviest-to-nearest strategy unless the granularity of a partitioning is rather fine. For this reason, this simple strategy is used in the assignment algorithm.

Algorithm : *Heaviest-to-Nearest Assignment*

Input:

1. A set of N clusters $C = \cup c_i$, $i = 0, .., N - 1$.

2. A non-negative real number cl_{ij} representing the intercluster communication between c_i and c_j.

3. A processor graph representing a network of N processors.

Output:

1. A one-to-one mapping from clusters to processors.

Algorithm:

1. Mark all clusters *unassigned* and all processors *unallocated.*

2. Repeat the following steps until all clusters are assigned.

If there is no assigned cluster or no communication between assigned clusters and unassigned clusters:

Locate the heaviest communicating but unassigned cluster.

Assign it to an unallocated processor with the most communication links.

Mark the unassigned cluster *assigned* and the unallocated processor *allocated.*

Otherwise:

Locate the heaviest communicating cluster pair consisting of one unassigned cluster and one assigned cluster.

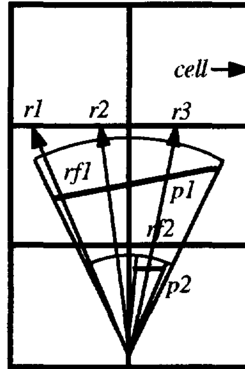
Assign the unsigned cluster to the nearest unallocated neighbour of the processor allocated for the assigned processor.

Mark the unassigned cluster *assigned* and the unallocated processor *allocated.*

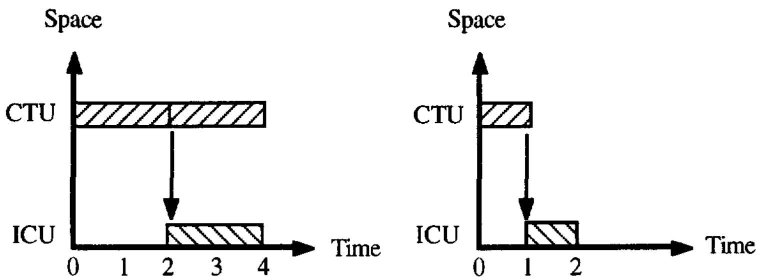
Figure 4.12: The heaviest-to-nearest strategy .

4.4.2.4. Local Scheduling

ASRS consists of a set of heuristics based on the assumption that message idle times can be perfectly overlapped with useful computations when more and more ray-frustum processes are packed into a cluster and assigned to a processor. This assumption may break down if the execution ordering for local processes is inappropriate, thus the execution time may increase undesirably. This raises the fourth subproblem in *ASRS* called local scheduling. The purpose of the local scheduling algorithm is to define local processes and determine their ordering for execution.



(a) A Cluster



(b) Ideal Execution Profiles for *rf1* (left) and *rf2* (right).

Figure 4.13: A simple example.

The local scheduling algorithm takes the optimal solution based on an ideal execution profiles as a target. This is because the ideal execution profile stands for the best ever possible execution. As far as the runtime constraint is concerned, it is very hard to achieve the optimal solution for a totally unknown space. In the first place, the fulfilment of the ideal execution profile may be violated as a ray doesn't know about the patches in its path until it strikes one. At least, the earliest time to find a patch cannot be guaranteed if a ray doesn't know where is the patch.

Hopefully, some representative ray frustums become known to the scheduler after low-density ray casting. An interesting question is to what extent the known information can be used to guide the local scheduling algorithm. For a better understanding of the problem, it is helpful to illustrate with a simple example. Consider the local scheduling of a cluster as shown in Fig. 4.13a. As you can see, there are two ray frustums *rf1* and *rf2* whose ideal execution profiles are given in Fig. 4.13b. In this example, we can quickly come to the optimal solution as shown in Fig. 4.14a. We make use of the optimal solution to evaluate a certain solution, which can lead the way to the solution algorithm. For this purpose, we discuss the following three strategies.

1. Depth-First Strategy

   In order to find the patch defining a ray frustum as early as possible, a depth-first search is used. This amounts to defining one subshell for a given ray frustum defined by a patch. In this example, we have two subshells, say *subshell1* and *subshell2*, defined by *rf1* and *rf2*, respectively. If *subshell1* is executed first, then the resultant execution profile as shown in Fig. 4.14b clearly deviates from the optimal solution by introducing one more idle time for *CTU* due to one extra intersection computation. This extra intersection computation comes from ray *r3* which may otherwise hit patch *2* and die if a breadth-first search is taken.

2. Breadth-First Strategy

   With this strategy, one or more subshells may be defined for a given ray frustum, depending on the number of shells over which the ray frustum extends. In this example, two subshells will be defined by *rf1*. The above extra intersection computation can be saved by using this strategy. However, it can be seen from Fig. 4.14c that one more idle time for *ICU* is introduced as compared with the optimal solution. This is because the earliest time to find patch *1* no longer holds.

3. Speculative Strategy

   From the above discussion, we can conclude with a speculative strategy with regard to a ray-frustum process: (1) If there is a high probability for the existence of patches, a breadth-first search is used to reduce unnecessary intersection computations; (2) Otherwise, a depth-first search is used instead in order to find a patch as early as possible. In this example, two subshells as shown in Fig. 4.14d are defined by *rf1*: (1) one attempts a breadth-first search due to the existence of patch *p2*, and (2) the other attempts a depth-first search as the region is empty. This is equivalent to say that *rf2* must be executed before *rf1*. This strategy leads to the optimal solution.

The problem that remains to be answered is how to determine the execution ordering when two ray frustums defined by two unoccluded patches are assigned to one processor. Consider the example in Fig. 4.15. As you can see, there are two ray frustums defined by two unoccluded patches *p3* and *p4*. The optimal solution suggests to start with *p4* as it can contribute much more intersection computations than *p3*. The difficulty lies in that it is difficult to determine the

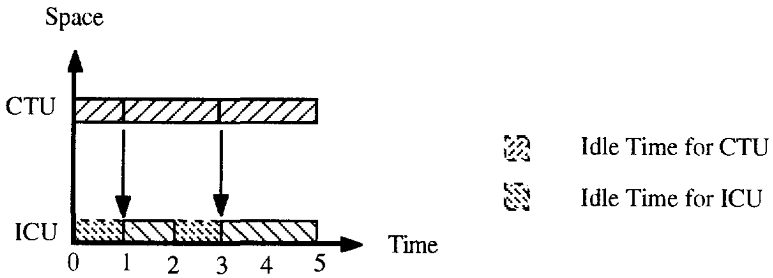
number of intersection computations that a patch can contribute especially when the patch is occluded by other patches. We adopt the following heuristic to minimize possible processor idle time.

1. Preprocessing

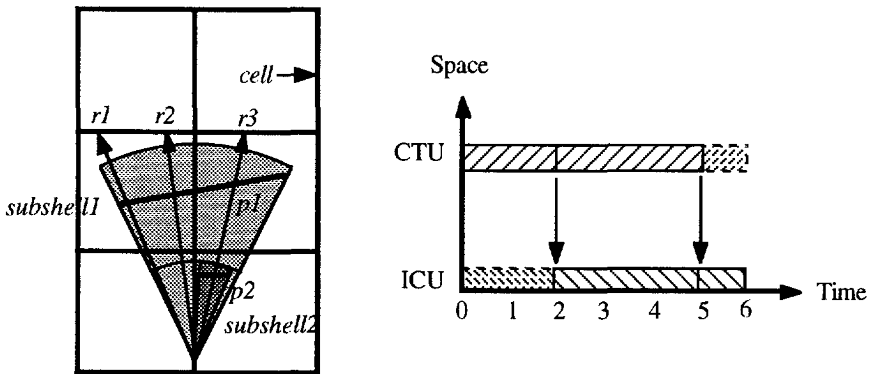
   A breadth-first search in the low-density ray casting brings about lists of patches roughly ordered shellwise and, most importantly, in a consistent way. By *consistent*, we mean that the ordering of patches in different shells correlates with the way we traverse the space consistently. During clustering, patches on each cluster are given descending priorities according to the order of finding them in the low-density ray casting. Consider the example in Fig. 4.15. By following the scan-line ordering, we traverse the two-dimensional space from the right to the left. First of all, patches in shell *2* (i.e., *p3*, *p1*) must be found earlier than patches in shell *3* (i.e., *p4*, *p2*). Secondly, in shell *2*, patch *p3* should be found earlier than *p1*. Similarly, in shell *3*, patch *p4* should be found earlier than *p2*. Thus it is reasonable to assume that patches are given descending priorities as *p3*, *p1*, *p4*, and *p2* instead of a contradicting ordering given as *p3*, *p1*, *p2*, and *p4*.

2. The Local Scheduling Algorithm

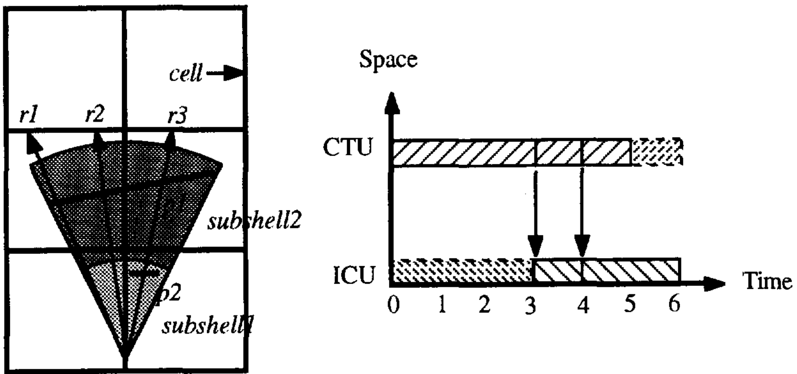
   Based on the speculative strategy, each ray frustum of a patch will be split into one or more subshells. This can be understood as follows. The regions defined by patches found in low-density ray casting are termed *highly probable hit* regions as rays most probably hit patches there (the shaded regions in Fig. 4.15). The other regions are termed *low probable hit* regions as they are most probably empty. Consider the example in Fig. 4.15 again. As explained in the above, patches are given descending priorities as *p1*, *p3*, *p2*, and *p4*. For patch *p3* or *p4*, only one subshell is defined as the region is assumed to be empty. In contrast, the ray frustum of *p2* is split into two subshells, say *subshell1*, *subshell2*, due to the existence of the *highly probable hit* region defined by patch *p1*. By virtue of the specular strategy, *subshell1* and *subshell2* attempt a depth-first and breadth-first searches, respectively. The consistency in the ordering of finding patches is of prime importance to processor idle time. To demonstrate this, consider the contradicting ordering given as *p3*, *p1*, *p2*, and *p4* in the above example. In this case, processor might be idle due to the data communication requirements representing rays flow from the subshell defined by *p1* to the subshell (i.e., *subshell2*) defined by *p2*. This is because a ray's behavior in one shell will depend on the computational results of that same ray in previous shells. To overcome this, we should delay the execution of subshells defined by *p2* as late as possible by executing the subshell defined by *p4* instead. This leads to the ordering given as *p3*, *p1*, *p4*, and *p2* as we suggested in the preprocessing step.



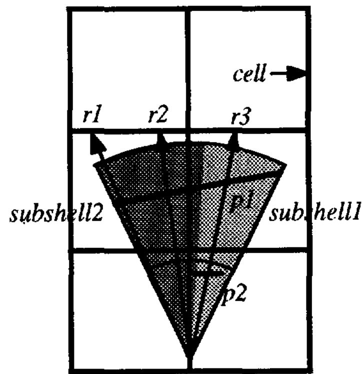
(a) Optimal solution based on ideal execution profiles.



(b) Depth-First strategy



(c) Breadth-First strategy



(d) Speculative strategy

Figure 4.14: Execution profiles for different local scheduling strategies.

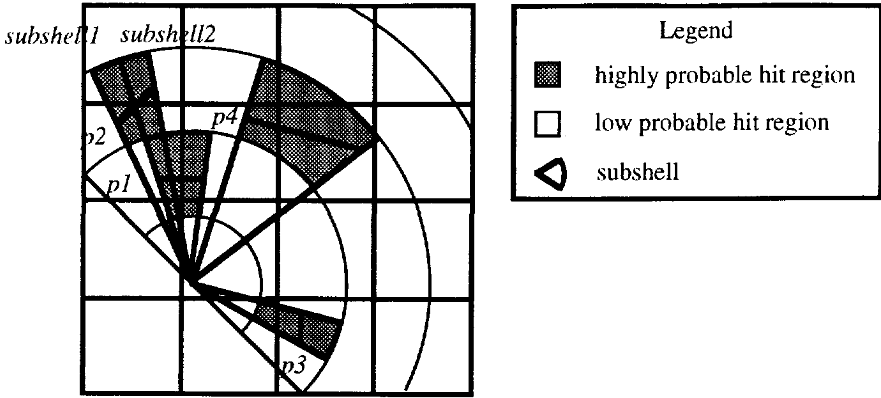
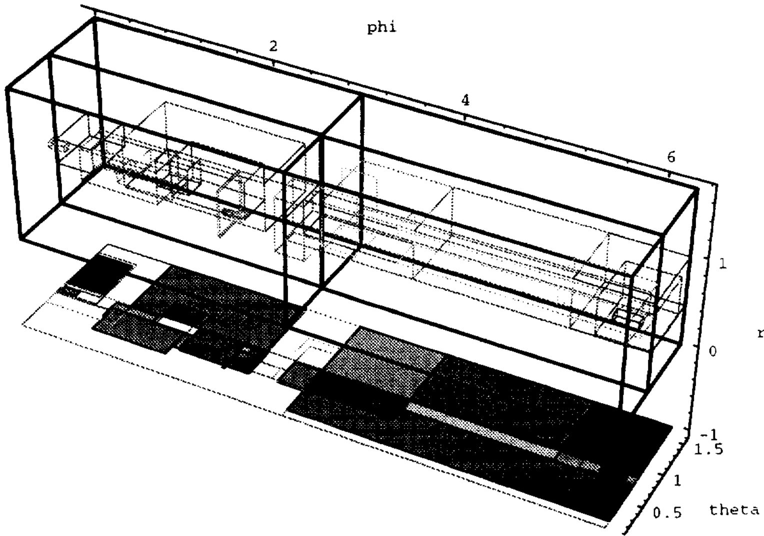
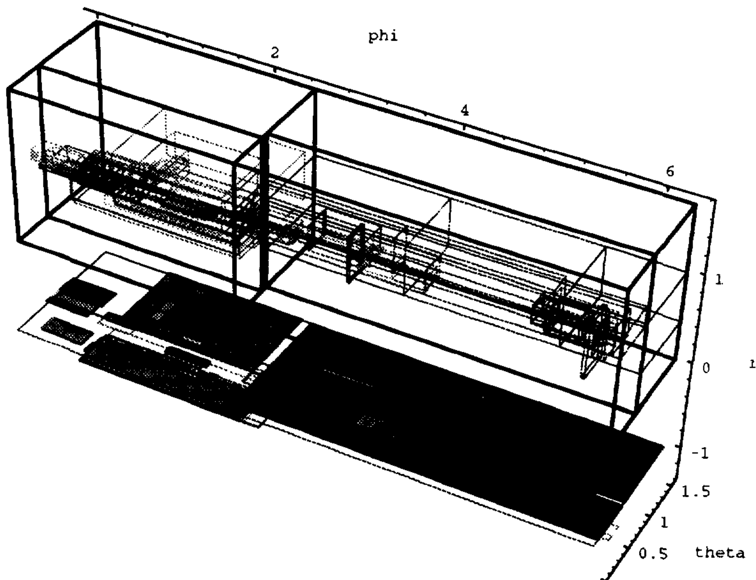


Figure 4.15: An example shows highly and low probable hit regions.



(a) *lobby*



(b) *room16.27*

Figure 4.16: The resultant clusters for two practical scenes.

4.5. Concluding Remarks

We have present in this chapter an *application-specific runtime scheduling* (*ASRS*) for mapping the *shelling technique* onto a *pipelined parallel architecture*. *ASRS* consists a set of heuristics which are simple but efficient, including: (1) runtime information gathering, (2) clustering, (3) assignment and (4) local scheduling. Finally, we demonstrate with two practical scenes, *lobby* and *room16.27* as shown in Plate 1 and Plate 2 to end this chapter. Figs. 4.16a and 4.16b show the resultant clusters viewing from a sample point on the ceiling of scenes *lobby* and *room16.27*, respectively. The 4 clusters and patches (actually patches' SBBs) found in the low-density ray casting are displayed as cubes in spherical coordinate system. To show the effects

of applying the patch classification algorithm, patches are orthographically projected on to the bottom plane and the shaded parts represent their portions inside a cluster. Several desirable features can be observed as follows:

1. Scene *lobby* has a few patches (227) which can be seen plainly through the sample point. As a result, the space is partitioned almost equally into 4 clusters (see Fig. 4.16a). In contrast, scene *room16.27* has 1297 patches stacked each other in the space, particularly in the lower left section (see Fig. 4.16b). The clustering takes effect clearly in this case.
2. As can be seen from the figure, patches are assigned to clusters where their major portions reside. This justifies the patch classification algorithm.

Chapter 5

The Radiosity Engine

5.1 Introduction

Many algorithms from computer graphics lend themselves to parallel implementation. Parallel architectures can be divided into two classes: (1) Single Instruction stream Multiple Data stream machines (SIMD), and (2) Multiple Instruction stream Multiple Data stream machines (MIMD). Usually, there are two reasons for pursuing SIMD machines: (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 is often very low. One noted example is Pixel-Planes 4 [Fuc85]. It is an array of 512×512 pixel-processors. The front end of the system first transforms a polygon into the eye coordinate system, and encodes the polygon's edges in linear equations of the form $Ax + By + C = 0$. Pixel-processors then evaluate the linear expression $F(x, y) = Ax + By + C$ simultaneously for all pixels. All the pixels outside the bounding lines (i.e., the sign of F is negative) will be disabled. Only those inside participate the further visibility and shading calculations. Pixel-Planes 4 offers massive parallelism in terms of a processor per pixel. However, it turns out to be rather low utilization of pixel-processors. When rendering 100-pixel polygons, Pixel-Planes 4 disables all the pixels outside a polygon, and hence all these pixels' processors remaining idle until the next polygon arrives. The utilization of pixel-processors is as low as 0.04% (i.e., $100/512 \times 512$) for a 512×512 pixels display screen. Consider a complex scene that is composed mostly of very small

polygons, and in which many polygons cover only a handful of pixels. The situation becomes even worse. The same problem arises when one attempts to use the Connection Machine [TMC87] for computer graphics.

MIMD machines useful for graphics span a wide range of spectrum: (1) mainframe computers connected by wide area networks, (2) workstations connected by local area networks, (3) multiprocessors consisting of processors communicating through a shared memory, and (4) multiprocessors consisting of processors each with a local memory, communicating through an interconnection network. We shall discuss some typical examples along this line.

1. Unlike the massively parallel SIMD machines, all current graphics workstations, e.g., the Silicon Graphics IRIS workstation [AJ88] [Akl89], the HP 835 SRX [McL88], the Alliant visualization system [Tor87], the Titan graphics supercomputer [DHM88], the Stellar GS 1000 [ABM88] and Apollo DN10000 [KV90], can only support tens or even fewer processors. For instance, the Silicon Graphics IRIS workstation [AJ88] [Akl89] makes use of a tree of hardwares: (1) A Polygon Processor decomposes a polygons into vertically *oriented trapezoids*; (2) Edge Processors determine the ends of the various vertical spans of the trapezoids, and broadcast them to 5 Span Processors making interpolations; (3) Each Span Processor handles every fifth column of pixels, and broadcasts them to 4 Image Engines making hidden-part elimination (Z-buffer algorithm); (4) In all, there are 20 Image Engines, each of which manages the twentieth of the pixels in the frame buffer. Although the utilization of processors is generally very high, the scalability of the system is low due to the limited bandwidth provided by the single bus and the memory structure.

2. Pixel-Planes 5 [FPE89] is the successor of Pixel-Planes 4, and tries to overcome the low utilization problem. The basic unit of the system is an array of 128 × 128 pixel-processors called Render. The display screen is partitioned into a number of disjoint subscreens, each assigned to a separate Render. Although polygons lying in different subscreens can be processed simultaneously by multiple Renders, the utilization of processors still can be quite low (0.6% for 100-pixel polygons). This system can support 8-10 Renders connected by a high-speed ring network (160 Mword per second).

3. Links-1 [NOK83] is perhaps the first massively parallel MIMD machine used for computer graphics. It consists of 64 unit computers interconnected with a root computer. A number of unit computers constitute a pipelined computer and such pipelined computers works in parallel, all controlled by the root computer. A pipeline consists of: (1) a sorting process to find all the objects penetrated by a ray and to sort them in depth, (2) a ray-tracing process to calculate the ray-object intersection point, and (3) a shading process to compute the shade of the corresponding pixel on the display screen. The whole image data is partitioned into subsets such that each is to be processed on a distinct pipeline. The root computer then broadcasts the code to be executed and the partitioned image data to the unit computers configured in the parallel pipelined scheme. Images produced on this machine generally

took on the order of a minute or more. When using 64 processors, 30-fold speedup over a single processor has been reported. The 256-processor Links-1 has been built at Osaka University.

4. Others are massively parallel MIMD machines consisting of hundreds or even thousands processors communicating over networks with multiple connections per processor [HMS86] [Sei85]. However, they will not achieve their potential until better programming environments emerge.

From the above discussion, we can conclude that:

1. The existing SIMD machines provide orders of magnitude more processors than other machines. However, they offer very little power per processor and often very low processor utilization.
2. The existing MIMD machines offers orders of magnitude greater power per processor but supports orders of magnitude fewer processors.

We have chosen a combined radiosity and ray-tracing algorithm for high quality rendering. However, this approach is demanding orders-of-magnitude more processing power for a single processor if we wish to make a state-of-art image in real-time or even interactive time. An obvious answer to this dilemma lies in parallel processing. In the first place, we are faced with the two fundamental issues for any parallel machine, that is, latency and synchronization. Secondly, existing graphics algorithms already posed some difficulties to both SIMD and MIMD architectures. In this work, considerable effort has been devoted to finding a highly efficient and effective parallel implementation of the target algorithm. We came up with a good algorithm-architecture pair, that is, the shelling technique and a pipelined parallel architecture. The shelling technique has been discussed in great details in chapter 3. In this chapter, we shall discuss some essential issues of the target architecture such as memory structure, network design, synchronization mechanism and the functionalities of the system.

5.2 System Architecture

5.2.1. System Configuration

The target system is made of a host computer and the radiosity engine [SD92] . The application program and the scheduler are running on the host. The application is a combined ray-tracing and radiosity algorithm that proved to be a powerful method for high-quality rendering. The ray-casting based rendering algorithm can be suitably decomposed into two separate tasks: shading and form-factor computations. In the radiosity pass, the shading task initiates hemisphere shooting and updates patches' radiosities. In the ray-tracing pass, the shading task initiates primary and secondary rays, performs local light reflection, texture filtering, and anti-aliasing, and is responsible for pixels' final shading. In both passes, the form-factor computations are best suited for parallel processing on the radiosity engine due to the huge amount of inherent parallelism. However, having a large amount of parallelism in computations is not sufficient to guarantee good speed-up. Parallelism in computations can only be realized by appropriate utilization of concurrency in the underlying hardware. Once a particular architecture is determined, concurrency utilization depends totally on how hardware resources are allocated and managed. This is a resource management problem. For this purpose, a scheduler is running together with the application in the host. In order to prepare for runtime information for the scheduler, a low-density version of form-factor computations should be carried out by the host. Based on the runtime information gathered, the scheduler partitions the computations into a number of clusters in such a way that workloads can be more or less balanced and inter-cluster communication can be reduced. Meanwhile, the scheduler partitions the problem data domain into data segments and distributes them over processors on the radiosity engine. This pre-loaded data scheme is supported by the system bus interface connecting the radiosity engine and the host. The system bus interface provides the communication protocols to distribute data over the local bus, to allow the addressed processor to read the data. For global memory references, processors can send requests to the system bus interface via the local bus. In this case, the system bus interface first checks the cache before directing the requests to the host.

The radiosity engine consists of a set of processors connected by a mesh network. The individual shaded boxes in Fig. 5.1 represent single processors. Each processor contains a Cell Traversal Unit (CTU), a Memory Unit (MEM), an Intersection Computation Unit (ICU), a local bus interface and a network bus interface. The terms processor and cluster will be used interchangeably when appropriate.

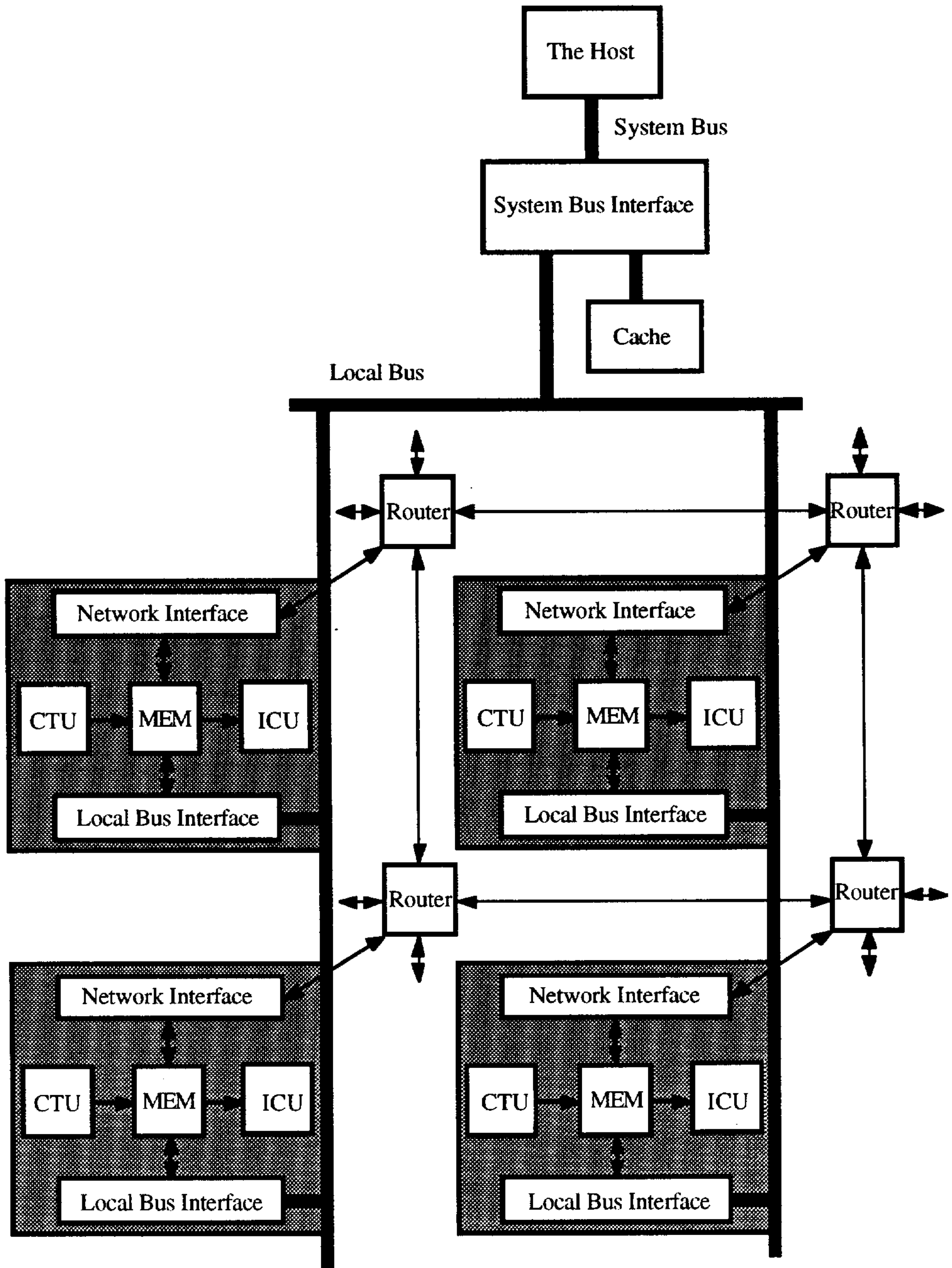


Figure 5.1: The system configuration.

5.2.2. Memory Structure

In most conventional computers with virtual-memory management systems, cache memories are often used as high-speed buffers between processors and main memory to capture those portions of the contents of main memory which are currently in use. Since cache memories are typically 5 to 10 times faster than main memory, they can reduce the average memory access time if properly designed. The success of cache memories is attributed to the property of *locality of reference*. Locality of reference has two components: *temporal locality* and *spatial locality*. In temporal locality, there is a tendency for a process to reference in the near future those elements of the reference string referenced in the recent past. In spatial locality, there is a tendency for a process to make references to entries in the neighbourhood of the previous reference. In the ray-tracing algorithm, an analogous situation arising from a new manifestation coherence called data coherence was exploited. When tracing rays by following a scan-line order, the phenomenon is in evidence that rays traced through adjacent pixels will traverse similar regions of space, and give rise to references to similar subsets of the model database. The effect of spatial locality arises from the data structures used for acceleration because successive references of the ray-tracing program are made to entries contained in a local neighbourhood of the object space. In [GP89], a hierarchical memory which is divided into a *resident set*[7], cache, and main memory was proposed for exploiting data coherence. In this case, the use of data coherence is at a rather low level which is known as a single-ray process. Yet another level of data coherence can be exploited in the shelling technique, as is explained below.

In order to reduce the impact of runtime overhead on the overall performance, each stable phase must be kept as long as possible. To achieve this, we organize the execution of a number of ray-casting procedures pertaining to neighbouring sample points (in radiosity pass) or patch-based intersection points (in ray-tracing pass) as one phase. It appears that the half spaces seen through neighbouring sample points or patch-based intersection points convey similar appearance (see Fig. 5.2). Thus, it is advantageous to pre-load patches found in low-density ray casting to the local memories of processors. This is because: (1) those patches remain to stay locally until a new phase starts and can account for a large percentage of references made during a phase, and (2) local memory references are much faster than main memory references.

[7] A *resident set* is a memory which is faster than a cache as it can be addressed directly and with minimum overhead. Normally, this set should contain the most referenced subset of the model database.

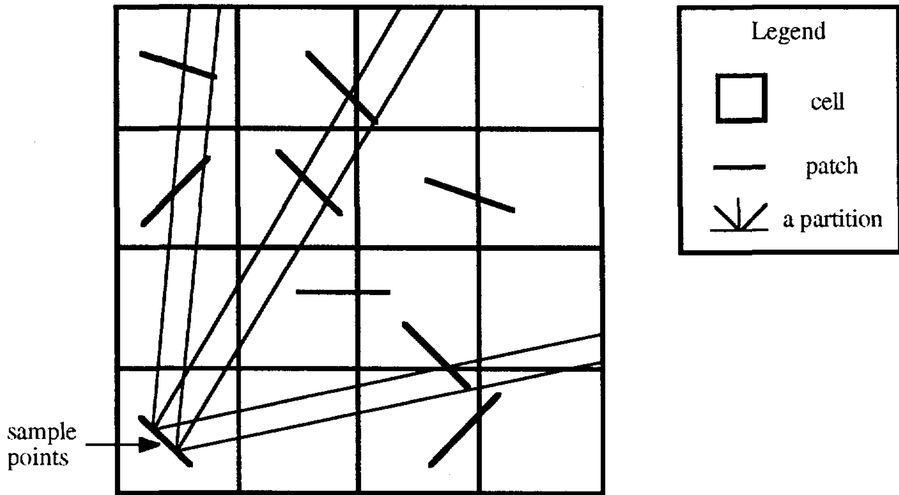


Figure 5.2: The data coherence between two sample points.

5.2.2.1. Basic Terms

The success of our memory structure is attributed to the data-coherence property which is analogous to the property of locality of reference in virtual memory systems. With these similarities, we borrow some terminologies from there to facilitate the description our memory structure.

Definition 5.1 (Reference String): In virtual memory systems, the sequence of references made by a program in execution can be represented by a *reference string* $RS(T) = r(1)\ r(2)...r(T)$, where $r(t)$ is the virtual address generated at time t. ■

Suppose that a database contains a set of N patches, denoted as $\boldsymbol{P} = \{P_1, P_2, ..., P_N\}$, and let $RS(T) = P(1)\ P(2)...P(T)$ be a reference string that represents the sequence of references made in the course of a ray-casting procedure, where $P(t)$ is a patch referenced at time t.

Definition 5.2 (Frequency Function): A *frequency function* f: $\boldsymbol{P} \rightarrow \boldsymbol{Z}$, where $\boldsymbol{Z}$ is the set of all positive integers, returns the number of occurrences of a patch in $RS(T)$. ■

Definition 5.3 (Usage Function): A *usage function* U: $\boldsymbol{J} \rightarrow \boldsymbol{P}$, where $\boldsymbol{J} = \{j \mid j \in \boldsymbol{Z} \text{ and } 1 \leq j \leq N\}$, returns the usage sequence of a patch in $\boldsymbol{P}$. That is, $U(1)$ is the most frequently used patch in $\boldsymbol{P}$ and $U(N)$ is the least frequently used patch in $\boldsymbol{P}$. ■

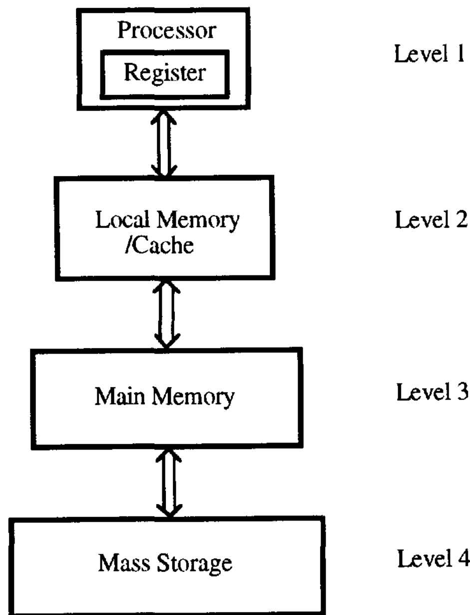


Figure 5.3: A hierarchical memory system.

5.2.2.2. A Hierarchical Memory System

In order to take advantage of the data-coherence property at all levels, a hierarchical memory system has evolved in which a very fast register that acts as the first level of the hierarchy (see Fig. 5.3) [SD92]. Access to this register is very fast because it is on the processor chip. Each patch in a cell opened by a *CTU* is retrieved and stored in the internal register of an *ICU*. Afterwards, each remaining ray within the *SBB* of the patch accesses this high-speed register that allows the *ICU* to perform at its peak. At the second level, between the processor and slower main memory, is a local memory and a cache. We assume that a local memory is faster than a cache as it can be addressed directly and with minimum overhead. Each cluster is physically associated with a local memory which is pre-loaded with a small subset of the model database that can account for a very large fraction of the references made during a phase. Similar to [GP89], we identify the patches found by the low-density ray casting as the resident set to be stored in local memories. Besides, a cache which is global to all the clusters is used for caching patches read from main memory. In this scheme, local memories account for long-term behaviour in the data-coherence property, whereas a cache accounts for short-term behaviour in the data-coherence property. The third level of the hierarchy is main memory where the

application program and the model database is stored. Typically, this memory is much larger than local memory/cache but also slower than local memory/cache. The fourth level of the hierarchy is mass storage. It is used to implement a so-called virtual memory system, which gives the processor the illusion that main memory is much larger than is the case. For a big scene containing millions of patches, mass storage can be used to store the complete model database in case main memory is insufficient for this purpose.

We first describe how to select the resident set to be stored in local memories. Next, different policies in the cache design will be discussed.

At the beginning of each phase, a unit hemisphere is placed over a reference point which is the center of a source patch. A low-density ray casting is then invoked to gather runtime information. We could identify patches found by the low-density ray casting as the resident set to be stored in local memories. Those patches are often large because they were found by low-resolution rays. Hence, they can account for a large fraction of the references made to local memories when casting high-density rays from other sample points during the same phase. There is a basic attribute to measure the goodness of a resident set, called the *effectiveness* of the resident set. Suppose that the frequency function f_h with regard to all the instances of high-density ray casting in a phase is known, and let U_h be the usage function derived from f_h. If we are allowed to select the resident set based on U_h, then an optimum solution can be obtained. In general, this will not be the case. Let f_l and U_l be the frequency function and the usage function with regard to the low-density ray casting, and let N be the total number of patches found in the low-density ray casting. Instead of selecting the optimum resident set $R_h(k) = \{U_h(1), U_h(2), ..., U_h(k)\}$, only $R_l(k) = \{U_l(1), U_l(2), ..., U_l(k)\}$ can be selected by the low-density ray casting, where $1 \leq k \leq N$. Then, the effectiveness $E(k)$ of $R_l(k)$ is defined as

$$E(k) = \frac{\sum_{i=1}^{k} f_l(U_l(i))}{\sum_{i=1}^{k} f_h(U_h(i))} \quad \text{for } 1 \leq k \leq N. \tag{5.1}$$

Note that the $E(k)$ is a measurement of the goodness of a resident set during a phase. Because the resident set is selected based on a so-called reference point, the $E(k)$ may become to be ineffective if a sample point in the phase is fairly far away from the reference point.

5.2.2.3. Cache Design

In general, there are four placement policies: direct, fully associative, set-associative, and sector mappings, used in designing a cache (see [HB84]). 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 special hardware for an associative search of address tags. On the contrary, fully associative mapping is the most flexible one in the sense that 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.

5.2.3. Network Design

A data-parallel algorithm partitions its problem data domain into data segments and distributes them among the processors, and let each processor work on the data segment assigned to it independently. The benefit of this pre-loaded data scheme is performance. After data pre-loading, all memory references become local memory references which are much faster than global memory references. Our target system consists of the host and a number of processors connected by an interconnection network. The local memory of each processor is pre-loaded with a subset of the model database that is determined by a low-density ray casting. Some relevant patches which are too small may not be captured in the low-density ray casting due to the limited number of rays used. Some relevant patches which are too large may be wanted by many processors but can only be stored in one processor due to the limited capacity of local memories. This incurs communication requirements between the host and the radiosity engine or between two processors on the radiosity engine. In the current implementation, the use of a high bandwidth system bus (100 Mbytes/sec) together with a global cache in the System Bus Interface may satisfy the former requirement. We shall now discuss the latter case.

If the data required by one processor is stored in a remote processor, inter-processor communication is required to move in the data via an interconnection network. The performance of the interconnection network is therefore essential to the performance of the system. The performance of an interconnection network is greatly influenced by the routing algorithm and the switching technique used in the network, as well as by the topology of the network. Our discussion of these issues concentrates on network topologies since the others are implementation issues.

In general, network design decisions are to be made by analyzing and comparing a large number of possible network topologies. However, it is possible to correlate certain interconnection topologies with certain traffic patterns to yield better performance. For instance, a linear array is an ideal choice for a program consisting of cascaded tasks. In our case, it turns out that a mesh-connected network is a good choice. Due to the limited size of a patch, it occupies a contiguous and bounded region in space. In case the granularity of a partitioning is not very fine, a patch most probably belongs to one or a few contiguous partitions. As discussed in section 4.4.2.3, the assignment algorithm assigns neighbouring clusters to neighbouring processors so that most traffic requirements are only issued to neighbouring processors. This implies that an interconnection network like mesh that supports moderately to

processors. This implies that an interconnection network like mesh that supports moderately to highly localized traffic patterns would be appropriate.

For comparison purpose, we choose ring, mesh, torus, and hypercube as our network candidates. Although the ring network, predictably, is outperformed by most networks, it is the most appropriate interconnection for those applications most concerned with board space. At the other extreme, the hypercube network supplies high bandwidth at all network sizes and any traffic pattern, but requires significant board space to implement, which grows with the size of the network. List below is a discussion of the characteristics of those network candidates:

1. The ring network

   The ring network is a simple ring, where each processor is connected to its two neighbours by independent half-duplex bidirectional links, as shown in Fig. 5.4a.

2. The Illiac network

   The Illiac network is a two-dimensional mesh on which each grid represents a processor connected with its Up, Down, Right, and Left neighbours by half-duplex bidirectional links. The boundary links wrap around to connect processors on the other side, in the manner of the Illiac IV, as opposed to the torus topology discussed in the below. Fig. 5.4b shows a 16-processor mesh network.

3. The torus network

   The torus network is also a two-dimensional mesh similar to the Illiac network but with the boundary links connected in a different manner, as shown in Fig. 5.4c.

4. The hypercube network

   An n-dimensional hypercube (n-cube) consists of $N = 2^n$ processors constructed as follows: the processors are addressed distinctly by n-bit binary numbers, $b_{n-1}b_{n-2} \dots b_1b_0$, form 0 to $2^n - 1$; two processors are directly connected via half-duplex bidirectional link if and only if their binary addresses differ in exactly one bit position. Note that the hypercube network has recently received increased interest due to the commercial availability of multiprocessor systems based on this topology (Intel iPSC, NCUBE, etc.). Fig 5.4d shows a 16-processor hypercube network.

Another important characteristic of the above two-dimensional mesh (i.e., Illiac or Torus) is that their boundary wraparound connections are consistent with the boundary wraparound sections built by the scheduler. As described in section 4.4.2.2, the *BSP* algorithm for building sections bisects the space by taking alternatively constant-φ and constant-θ partition planes. This results in sections which are wrapped around and touched at the $\varphi = 0$ partition plane. It would be adequate to wrap around the boundary links to meet the traffic requirement there.

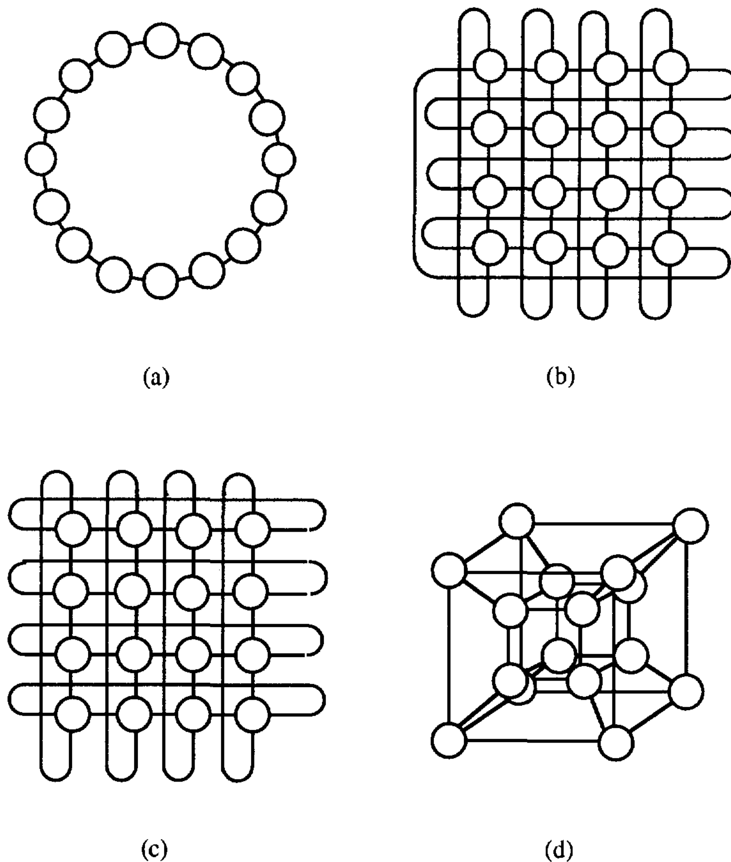


Figure 5.4: Network candidates (a) Ring. (b) Illiac. (c) Torus. (d) Hypercube.

5.2.3.1. Network Performance

We have experimented on a number of practical scenes to investigate the traffic patterns between each pair of clusters for a partition. It turns out that most traffic requirements are issued to neighbouring processors. For comparison purpose, we introduce a network performance metric to measure the static performance of the network candidates. Note that this comparison is based on static measurements. In section 6.2, we shall discuss the effects of different network topologies on the system performance, which is consistent with the comparison made here.

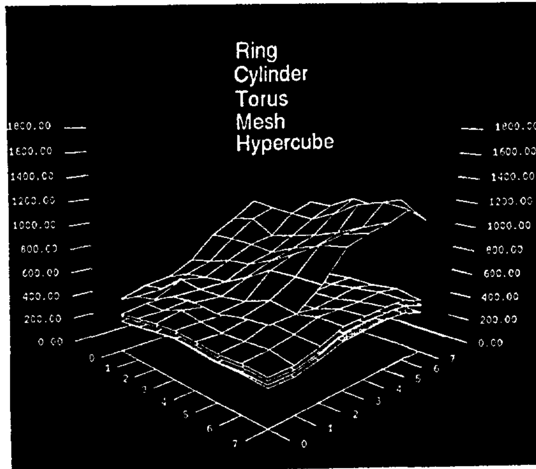
We first describe the test conditions for two practical scenes, *lobby* and *room16.27*, as depicted in Plate 1 and Plate 2. We partition the ceiling into 64 elements by using binary partition, then define 64 sample points at the center of each element. A static partition is determined by the scheduling based on the first sample point (called reference point) near the center of the ceiling and a specified number of clusters. The other 63 sample points keep on using the same partition to amortize the scheduling overhead over many computations. In this way, patches are pre-loaded to the local memories based on the static partition. If the data wanted by one processor is stored in a remote processor, inter-processor communication is required to move in the patch. The situation may become even worse when the granularity of a partitioning is very fine and/or a sample point is far-away from the reference point.

We use the sum of the *number of hops* for each message must traverse to reach its destination node to measure the performance of a network. Another metric would be the *number of network cycles* required, which is the sum of the number of hops each message must traverse to reach its destination divided by the total number of links in the network. However, it can only be viewed as an upper bound on the performance as 100% utilization for each link is rarely achieved. This is particularly true when the routing chip can only support a simple routing strategy, which is generally the case for VLSI implementation. Moreover, a larger number of links often reflects a higher physical cost. Figs. 5.5 and 5.6 show the total number of hops required for scenes *lobby* and *room16.27*, respectively. The network types are labelled by following the sequence of the curves shown in the figures.

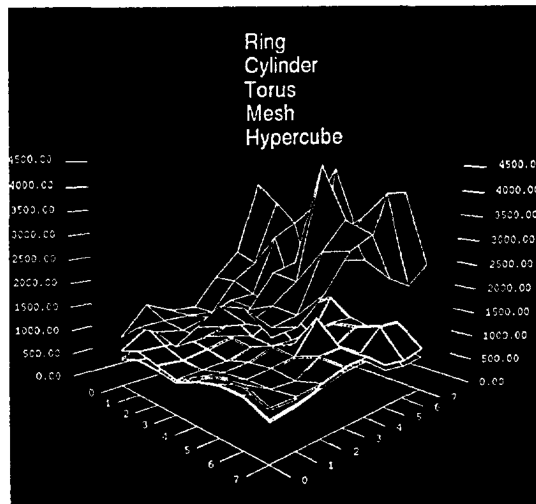
Fig. 5.7 shows the number of processors at a certain distance (i.e. hops) away from a source node in a 16-processor network. In case traffic only issued to closest two neighbours, these networks would all generate the same number of hops, because they all have at least two neighbours within one hop. However, for uniformly distributed traffic, the ring network would generate much higher network load than in the others because it doesn't provide as a dense interconnection. This explains why it is outperformed by all networks with all configuration sizes. The situation becomes even worse when the number of processors is increased. So we dropped the ring network.

The hypercube network shows good performance for all cases because it provides a smaller depth (closer processor proximity, higher locality). A major problem of this network is its large waist (i.e., interfaces per processor), which grows with the size of the network, possibly taking away board space from the processors.

The two-dimensional mesh (i.e., Illiac or Torus) shows reasonable performance as expected. This is because it can absorb the network load due to the moderately to highly localized traffic generated. Furthermore, the two-dimensional mesh provides the best performance of all candidates for small configurations (up to 8-processor network). We conclude that the two-dimensional mesh is a good choice. In the following, we shall focus on some design issues of a mesh-connected network.

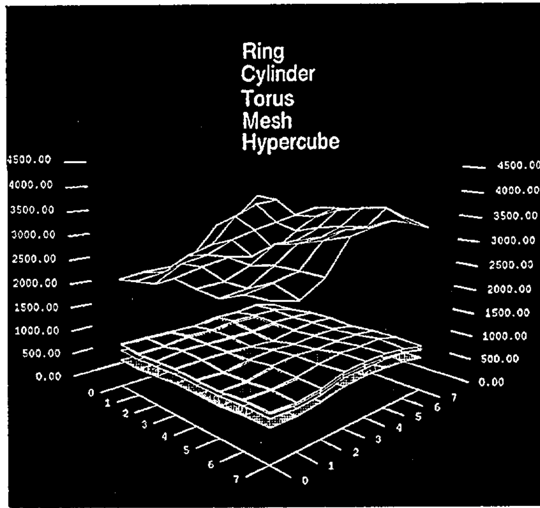


(a) $N = 32$

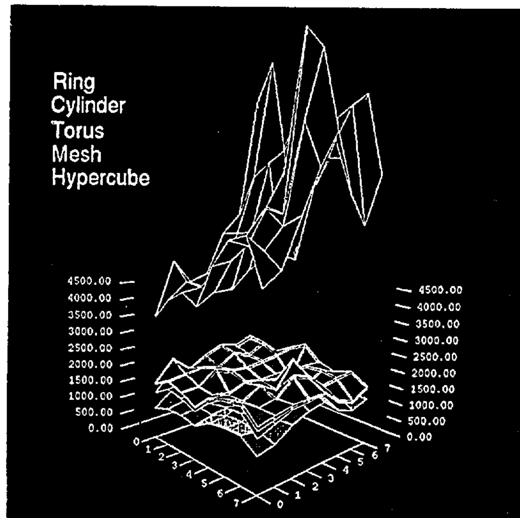


(b) $N = 64$

Figure 5.5: Number of hops for scene *lobby*.



(a) $N = 32$



(b) $N = 64$

Figure 5.6: Number of hops for scene *room16.27.*

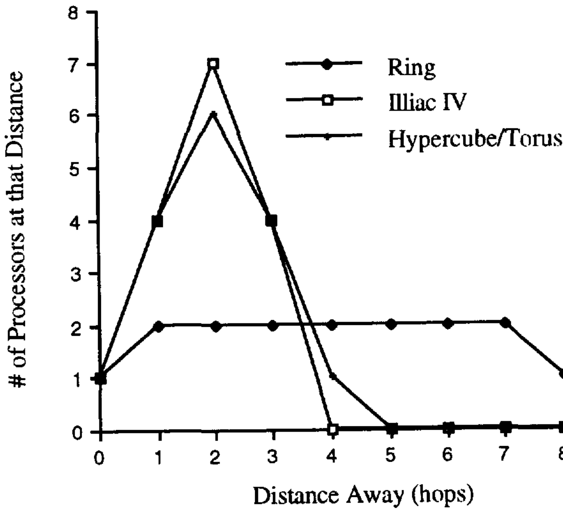


Figure 5.7: Proximity of neighboring processors (16-processor network).

5.2.3.2. Design Issues for a Mesh-Connected Network

The primary design requirements for a mesh-connected network are low latency and deadlock free.

1. Low Latency

   It has shown that low dimension k-ary n-cubes along with wormhole routing meets the goal of low latency [Dal87]. For this reason, it becomes the most promising switching technique and has been adopted in more advanced multicomputers. Wormhole routing breaks a message into *flits*. As soon as the header flit(s) has been received, the next channel on the route can be selected and the remaining flits of the message can be forwarded down the channel in a pipeline fashion. It is possible for the first flit of a message to arrive at the destination node before the last flit of the message has left the source. Because most flits contains no routing information, the flits in a message must remain in contiguous channels of the network and cannot be interleaved with the flits of other messages. When the header flit of a message is blocked, all of the flits of a message stop advancing and block the progress of any other message requiring the channels they occupy. This can be solved by breaking a long message into a number of packets. Each packet has a header carrying routing information. This allows packets from different messages to be interleaved on the same channel. The second generation transputer IMS

T9000 uses this scheme that provides a total of 80 Mbytes/sec bidirectional bandwidth. Another example is the Caltech mesh routing chip (MRC) developed at the Caltech that can support a total of 240 Mbytes/sec bandwidth.

2. Deadlock Free

   Deadlock in an interconnection network occurs when no message can advance toward its destination because the queues of the message system are full. It can occur in a network unless the routing algorithm is designed to avoided it. For regular networks like trees, hypercubes and meshes, optimal deadlock free routing algorithms have been developed. A detailed description is beyond the scope of this thesis.

   In the current implementation, we propose to use MRC to build a mesh-connected network. The MRC routes messages moving through the mesh at high speed. As soon as the MRC examines the routing information of a message, it selects the direction in which the packet is routed. If the MRC finds the X displacement specified in a packet is nonzero, it forwards the packet to the next MRC in the X direction. The MRC continues routing the packet in the X direction until the X displacement goes to zero. Then it begins with routing the packet in the Y direction until the Y displacement goes to zero, then the MRC routes the packet to the connecting processor. With this simple routing scheme, a single mesh network cannot meet the requirement for deadlock free routing of requests and replies. To avoid deadlock, it is necessary to have two completely separate networks for requests and replies. Note that the two networks need not be physically separate, but must contain dedicated resources (buffers, handshake lines, etc.) so that blockage of the request network does not affect the reply network. This is supported by another chip called mesh interface module (MIM) that provides an 8-bit interface to the MRC and a 32-bit interface to the processor. It includes separated transmit and receive FIFOs and separated handshake lines, etc.

5.2.4. Synchronization Mechanism

A parallel algorithm is a set of concurrent processes which may operate simultaneously and cooperatively to solve a given problem. To ensure that a parallel algorithm works correctly and effectively to solve a given problem, processes interact to synchronize and exchange data. A typical situation happens in a concurrent loop with *data dependencies* across iterations. One iteration *must* wait, if necessary, until the corresponding data has been written by another iteration. In our case, what's synchronized is at control level. Because only remaining rays will be used for cell traversal and intersection computation, cell traversal and intersection computation have to synchronize at some point to enforce those control dependencies. So the rule of synchronization is not as stringent as it's general stated.

5.2.4.1. A Fine-Grained Synchronization Mechanism

In the shelling technique, the use of barrier synchronizations at shell level allows parallel execution of all the remaining rays within a shell. There are some difficulties with this approach: (1) It is difficult to exploit this run-time changing parallelism by assigning a separate processor to each ray stream; (2) The amount of parallelism could be drastically reduced for outer shells as rays may die successively; and (3) The parallelism in computation may become insignificant due to the high communication overhead from resource conflicts. Pipeline processing seems more appropriate because it can handle changes in the number of ray streams in a natural way. In addition, the communication overhead can be amortized over a large number of intersection computations. Due to the property of *object coherence*: local neighbourhoods of space tend to be occupied by the same object, multiple processors can be used for processing distinctly different subspaces to achieve higher system performance. This leads to a highly pipelined parallel architecture. It is undesirable to use the barrier synchronization at shell level due to the high latency time of a pipeline circuit. Instead, we define subshells as local processes that have to be synchronized by using the barrier. Moreover, a fine-grained event synchronization is introduced that can support fine-grained parallelism with low cost.

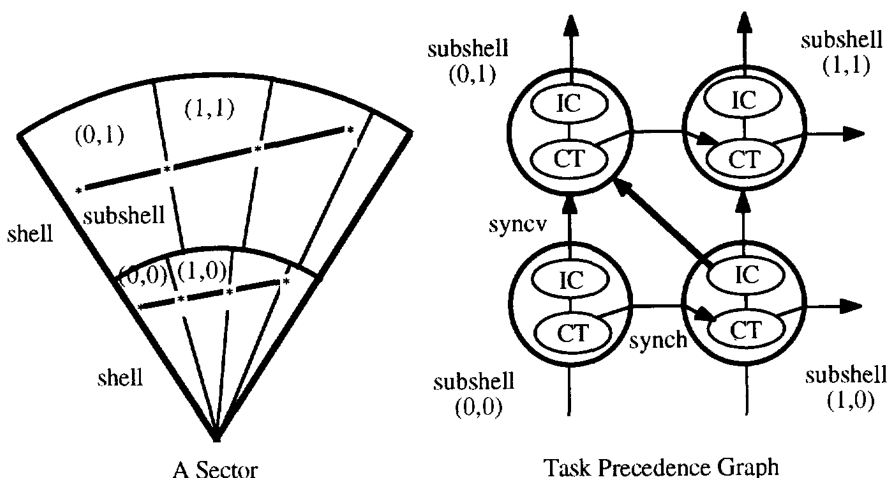


Figure 5.8: Task Precedence Graph for subshells.

The fine-grained event synchronization is supported by an I-structure[8] memory [AI87] [AT80]. 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[9] for subshells as shown in Fig. 5.8. 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 test against the rays leaving subshell (0,0) but must leave the part of rays that have left subshell (1,0) to be determined through the use of the I-structure memory as shown in Fig. 5.9. The presence bit of a ray in subshell (1,0) remains empty until subshell (1,0) is completed. The ray is stored in *Deferred Ray Memory* and will be reactivated and tested against the accompanying patch when the presence bit becomes full.

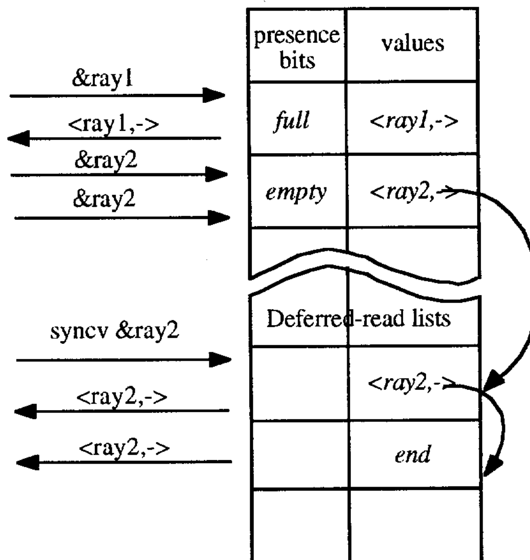


Figure 5.9: I-structure memory.

---

[8] I-structure means a structure with Incremental nature of production and consumption.

[9] In Task Precendence Graph, a parallel program is modeled as a collection of tasks with explicit execution dependences expressed in the form of precedence relations.

5.2.4.2. Synchronization for Cell Traversal

By convention, a signal *a* in iteration *i* of a loop can be referred to as *a*[*i*] where *i* is the index of variable *a*. Similarly, a cell-traversal ray or an intersection-computation ray can be named by different versions in the course of its lifetime in different shells. For this purpose, a ray *r* is associated with a *shell_id* flag representing its current version, and can be referred to as *r->shell_id*. As stated previously, a subshell is a local process for synchronization. Its process identifier is denoted as *subshell->subshell_id*. Similarly, a subshell is associated with a *shell_id* flag representing in which shell it resides, and can be referred to as *subshell->shell_id*. We now discuss the synchronization for cell traversal. Our design requirements are the following:

1. At any time instant, only one copy of a cell-traversal ray is allowed.
2. For each version of a cell-traversal ray, we can only start with cell traversal once.
3. A cell-traversal ray traverses the space in depth order.
4. Only remaining rays are allowed to continue traversing.

```
if ray->shell_id < subshell->shell_id then
    begin
        if ray->presence_bit = TRUE then
            begin
                ray->presence_bit := FALSE;
                ray->shell_id := subshell->shell_id;
                copy(ray)
            end;
        else defer(ray)
    end;
```

Figure 5.10: Synchronization for cell traversal.

The synchronization for cell traversal is shown in Fig. 5.10. When defining a subshell, we first compare the version of each *ctray* (i.e., cell-traversal ray) within the subshell with the version of current subshell. In case *ctray->shell_id* is larger than *subshell->shell_id*, we ignore *ctray*

because it already traversed to a higher numbered shell. Otherwise, we copy or defer *ctray* depending on its *presence_bit*. If *ctray->presence_bit* is set, we reset *ctray->presence_bit*, upgrade *ctray->shell_id* to be *subshell ->shell_id* and copy *ctray* into a queue waiting for cell traversal. Otherwise, we defer the decision of copying *ctray* to be made at a later time. The concept of using *presence_bit* for synchronization is inspired by the concept of I-structure memory. However, the deferred memory in our case is much simpler than that used in I-structure memory. When deferring a *ctray*, it suffices to only store *subshell->subshell_id* and *subshell->shell_id*.

Finally, we can mention that our approach satisfies the above-mentioned requirements: on the one hand *ctray* can be kept from duplication in the current version by upgrading *ctray->shell_id* to be *subshell->shell_id*, on the other hand it allows to continue traversing only when its *icray* counterpart is not hit by checking *ray->shell_id* against *subshell->shell_id* when *ctray->presence_bit* is set.

5.2.4.3. Synchronization for Intersection Computation

The requirements for the synchronization of intersection computation are the following:

1. Multiple copies of an intersection-computation ray are enforced.
2. Only remaining rays are used for intersection computation.

```
if ray->shell_id <= patch->shell_id then
    begin
        if ray->presence_bit = TRUE then
            copy(ray);
        else defer (ray)
    end;
else copy (ray);
```

Figure 5.11: Synchronization for intersection computation.

Similar to cell traversal, a synchronization algorithm for intersection computation is given in Fig. 5.11.

1. As stated previously, subshells are defined by patches found in the low-density ray casting. For each subshell, if the *presence_bit* of an *icray* within the subshell is set, we reset *icray->presence_bit*, and upgrade *icray->shell_id* to be *subshell ->shell_id*. If this happens, we say the *icray* has been defined by the subshell. This allows *icray* to have multiple copies for intersection computations, which is essential to the pipeline efficiency of an *ICU*.
2. The speculative strategy is elegantly implemented in the algorithm. If an *icray* has been defined by a subshell, its *shell_id* should be upgraded to be *subshell->shell_id*. Otherwise, its *shell_id* is kept unchanged. When determining intersection computation rays for a *patch*, we first compare the *shell_id* of an *icray* within the *SBB* of *patch* with *patch->shell_id*. As is apparent, if *icray* has never been defined by a subshell before, it will be copied directly.

Notice that the overhead of deferring intersection-computation rays are quite high. When deferring a *ray*, its accompanying *patch* must be stored together with the *ray* in the deferred ray memory. This implies that the *patch* must be referenced many times if many *rays* defined by its *SBB* have been deferred. In current implementation, we suggest to use a lazy isolation as given in Fig. 5.12 to filter out unnecessary rays, where *ray->intd* represents the current intersected distance for *ray* and *patch->r_min* represents the minimum distance from the sample point to *patch*.

if *ray->intd* > *patch->r_min* **then** *copy*(*ray*);

Figure 5.12: Lazy isolation for intersection computation.

5.3 Outlining Functionality

5.3.1. Introduction

Performance evaluation and trade-off analysis are the central issues in the design of a computing system. For this purpose, we often build a *model* that is used to try to gain some understanding of how the corresponding *system* behaves. If the relationships that constitutes the model are simple enough, it may be possible to use mathematical methods (such as algebra, calculus, or probability theory) to obtain *exact* information on questions of interest; that is called an *analytic* solution. However, most real-world systems like the one we have are too complex

to allow realistic models to be evaluated analytically, and these models must be studied by means of simulation. A mixed-level simulator called the Block Oriented Network Simulator (BONeS®) is chosen for simulation-based analysis and design of the target system.

BONeS® uses a hierarchical data flow block diagram as the modelling paradigm. You construct a model by specifying the data structures flowing in the model, and the data flow firing rule governs the movement of the data structures. This allows the data-driven execution of the radiosity engine to be modelled easily. Another important reason to use BONeS® is that it provides a mixed-level simulation, that is, the level of modelling for blocks can be detailed or abstract. Detailed models provide accurate results but may take a long time to develop and consume large amounts of computer resources to generate results. Abstract models typically generate results much more quickly, but their results may not be useful if too many simplifying assumptions have been made in the abstraction. The radiosity engine has been completely modelled by using BONeS®. As we shall see, some key components of the system, e.g.., Geometry Transformation, Cell Traversal and Intersection Computation etc. are modelled in detail, while the Memory System and the Interconnection Network etc. are build in abstract. In summary, the integrated BONeS® environment allows you to:

1. Describe data structures, functions and connections in a hierarchical fashion.
2. Translate the model into a C program, and execute an event-driven simulation of the model.
3. Perform statistical analysis of the simulated data, extract performance measures and display the analysis results.
4. Perform design iterations and trade-off analysis.
5. Document models and results.

For modelling the complete system, we have built over 160 modules including some primitives. Certainly, we cannot give a detailed description for all the modules we built. In this section, we describe the functional behaviour of some major modules that allows the system to be operated without knowing the detailed implementation. To construct a computing system according to a hierarchical design principle, at each level of description, the details of some modules may be irrelevant at that level and can be deferred to a lower level. If we manage to choose some critical modules which are functionally defined at some level and implement their details only when they are meaningful at that level. This leads to the idea that different computing models can be chosen for different levels so that each model may best suit the level it serves. This recognition of functional behaviour is very helpful for the practice of system design. Fortunately, the labelling of a module most of time can self explain its functionality. Potential users can just open down to the lowest level of a module, where the detailed description can be obtained through the commands for Help. For instance, (Help) Describe command produces a read-only text dialogue for each selected object, with a description of the object in the dialogue. In addition, (Help) Show Primitive Code command displays a read-only dialogue containing the C

source code for the primitive module being edited.

5.3.2. Data Structures

The first thing to be considered in designing a BONeS® model is the data structure. Although the data structures created are sometimes used as an aid to the simulation, they should conform to the physical implementation of the system.

In the following, the data structures used for the radiosity engine model are described.

1. *Ray Frustum*

   In radiosity algorithms, the hemisphere rays used for form-factor computations form a ray frustum with the sample point as origin. The primary rays in ray tracing form a ray frustum with the viewpoint as origin. Ray frustums are further used for shadow rays and specular rays with an intersection point as origin, in which a subset of hemisphere rays is defined by the *SBB* of light sources and the specular reflectivity of surfaces, respectively. The ***Ray Frustum*** data structure is used to characterize a ray frustum representing hemisphere rays, primary rays, shadow rays or specular rays. This data structure contains the following fields:

   or — The origin of a ray frustum. It can be a sample point, a viewpoint or an intersection point.

   normal — The normal of a ray frustum. It can be the normal of a hemisphere or the viewing direction from a viewpoint.

   noc — The number of constant-θ circles on the surface of a hemisphere or the number of pixels on each column of the display screen.

   nor — The number of constant-φ circles on the surface of a hemisphere or the number of pixels on each row of the display screen..

2. *Section*

   For concurrent processing, the problem data domain is partitioned into data segments which are distributed among the processors. Accordingly, the set of hemisphere rays or primary rays is partitioned and distributed over a network of processors. The *Section* data structure defines the subset of hemisphere rays or primary rays assigned to an *ICU*. As described previously, a hemisphere ray or a primary ray can be addressed by a two-dimensional index (A_φ, A_θ) or (A_x, A_y), respectively. So the fields necessary for this data structure are:

   $A_{\varphi min}/A_{xmin}$ — The minimal φ or x index for addressing the subset of hemisphere rays or primary rays, respectively.

$A_{\varphi max}/A_{xmax}$	The maximal φ or x index for addressing the subset of hemisphere rays or primary rays, respectively.
$A_{\theta min}/A_{ymin}$	The minimal θ or y index for addressing the subset of hemisphere rays or primary rays, respectively.
$A_{\theta max}/A_{ymax}$	The maximal θ or y index for addressing the subset of hemisphere rays or primary rays, respectively.

3. *Sector*

   The *Sector* data structure defines the subset of hemisphere rays or primary rays assigned to a *CTU*.

4. *Subshell*

   A subshell is a local process that determines an appropriate way of searching the space. Instead of searching the space blindly, it defines a possibly non-empty space for cell traversal. Hence, many useful intersection computations may fill up the intersection-computation pipe quickly instead of introducing bubbles. The data structure which represents subshell is *Subshell*. It contains the following fields:

SBB/IPBB	*SBB/IPBB* used for defining cell-traversal rays.
shell_id	Shell identifier that denotes the Shell to which cell-traversal rays should traverse.
subshell_id	Unique identification number for a subshell (i.e., process ID) assigned in order of creation.

5. *CT Ray*

   The data structure which represents cell traversal ray (*ctray*) is *CT Ray*. The fields of the data structure are:

ctray_id	Identifier for the *ctray*.
or	The origin of the *ctray*, which is represented in the G-coordinate system.
dir	The direction vector of the *ctray*, which is represented in the G-coordinate system.
lambda	The largest intersected distance to a cell.
shell_id	Shell identifier that identifies the Shell on which the *ctray* is currently working.
subshell_id	Subshell identifier that identifies the subshell on which the *ctray* is currently working.

6. *IC Ray*

The data structure which represents intersection-computation ray (*icray*) is *IC Ray*. The fields of the data structure are:

icray_id	Identifier for the *icray*.
dir	The direction vector of the *icray*, which is represented in the R-coordinate system.
intp_id	Identifier for the patch intersected by the *icray*.
intd	The distance between the origin and the intersection point of the *icray*.
u, v	The local coordinate for the intersection point of the *icray*.
shell_id	Shell identifier that identifies the Shell on which the *ctray* is currently working. It will be handed over to the last subshell so that cell traversal can continue.
subshell_id	Subshell identifier that identifies the subshell on which the *icray* is currently working.

7. *Cell ID*

The fields of the data structure are:

cell_id	Identifier for the *cell*. It is used to address the contents of the *cell*.
destination_id	Cluster identifier that identifies in which processor (including the host) the patch address list of the *cell* is stored.
count	A number that denotes the total number of patches stored in the *cell*.
pointer	A pointer indicating the start of a patch address list stored in the Patch Address Table.
subshell_id	Subshell identifier that identifies the subshell where the *cell* was found.

8. *Cell Request*

In case a cell traversed has not been stored in the radiosity engine, a request must be sent out to the host to ask for the contents (i.e., a patch address list) of the cell. The *Cell Request* data structure is used for this purpose. This data structure contains the following fields:

cell_id	Identifier for the *cell*. It is used to address the contents of the *cell*.
source_id	Identifier that indicates which processor sending out the request.
memory_type	Flag indicating which kind of memory the requested *cell* is found. It is set to *0/1* if the requested cell is found in the global/cache memory.

Otherwise, it is set to 2, indicating the local memory. This field is used to model the memory access time.

9. *Patch ID*

The fields of the data structure are:

patch_id	Identifier for the *patch*. It is used to address the contents of the *patch*.
destination_id	Cluster identifier that identifies in which processor (including the host) the surface property and geometry information of the *patch* is stored.
subshell_id	Subshell identifier that identifies the subshell where the *patch* was found.

10. *Patch Type*

The *Patch Type* data structure represents the patch-type and the surface-type of a patch. The patch-type indicates whether a patch is a polygon or a bezier. The surface-type indicates the surface property of a patch. It can be diffuse-only, specular-only or both. It contains the following fields:

patch_type	Flag indicating whether a patch is a polygon or a bezier.
surface_type	Flag indicating whether the surface property of a patch is diffuse-only, specular-only or both.

11. *Patch Request*

In case a patch found by cell traversal has not been stored in the radiosity engine, a request must be sent out to the host to ask for the surface property and geometry information of the patch.

12. *Polygon*

This data structure represents the patch geometry information of a polygonal patch. It contains the following fields:

vertex[*0*] - *vertex*[*3*]	The four vertices of the polygonal patch in the G-coordinate system.
tr_vertex[*0*] - *tr_vertex*[*3*]	The four vertices of the polygonal patch in the R-coordinate system.
SBB/IPBB	The *SBB/IPBB* of the polygonal patch.
patch_id	Identifier for the polygonal patch.
subshell_id	Subshell identifier that identifies the subshell where the polygonal patch was found.

13. *Bezier*

    This data structure represents the patch geometry information of a bezier patch. It is similar to the *Polygon* data structure. The only difference is that 16 control points are used instead of 4 vertices.

5.3.3. Radiosity Engine System

The top level in the hierarchical model of the radiosity engine is shown in Fig. 5.13. To simplify the discussion, consider only one instance of the ray-casting procedure to compute form-factors. First, the radiosity engine is supplied with required information from the host by reading some files on *Init System Memory* module. These include:

1. Ray Frustum

   Ray Frustum gives the definition of a ray frustum in terms of the origin, the normal and some resolution-related settings. It is stored as a *Ray Frustum* data structure in the Ray Frustum Memory. The radiosity engine makes use of the *Ray Frustum* to define a set of rays conforming to a certain distribution function.

2. Section

   Section gives the definition of the portion of a ray frustum assigned to an *ICU*. It is stored as a *Section* data structure in the Section Memory.

3. Sector

   Sector gives the definition of the portion of a ray frustum assigned to a *CTU*. It is stored as a *Sector* data structure in the Sector Memory.

4. Subshell List

   Subshell List is a patch address list. Each entry in the patch address list will define a *Subshell* after reading a *Polygon* from the Patch Geometry Memory.

5. Cell Address Table

   The Cell Address Table stores the location of the patch address list, describing the contents of a cell. It is built up of: (1) a cluster identifier indicating in which cluster (or the host) the patch address list of a cell is stored, (2) a count indicating the total number of patches in a cell, and (3) a pointer indicating the start of the patch address list.

6. Patch Address Table

   The Patch Address Table stores the cell contents in the form of patch address list. One entry in the patch address list is built up of: (1) a cluster identifier indicating in which cluster (or the host) the patch address list is stored, (2) a patch identifier to identify the patch and (3) a pointer indicating the start of the patch type and patch geometry

information, that is stored in the Patch Type Memory and the Patch Geometry Memory, respectively.

7. Patch Type

   Patch Type identifies the patch-type and the surface-type of a patch. The patch-type indicates whether a patch is a polygon or a bezier. The surface-type indicates the surface property of a patch. It can be diffuse-only, specular-only or both. It is stored as a *Patch Type* data structure in the Patch Type Memory.

8. Patch Geometry

   For polygons, this means four vertices. For Beziers, this is a list of 16 control points. It is stored as a *Polygon/Bezier* data structure in the Patch Geometry Memory.

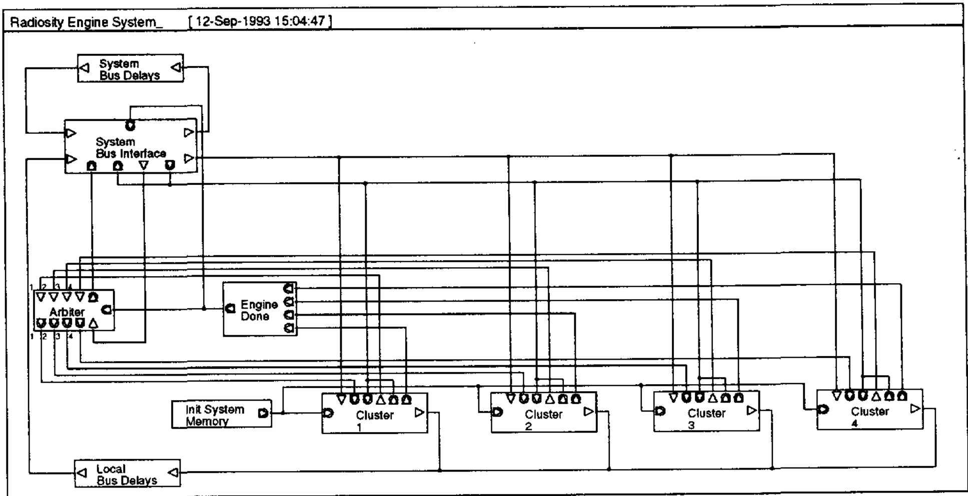


Figure 5.13: Radiosity Engine System.

When the *Init System Memory* module completes it will initiate 4 *Clusters* to start with form-factor computations. The *Local Bus Interface* module receives global cell/patch requests from the 4 *Clusters*. Upon receiving, they are packed into *Bus Requests* and queued in the module waiting to be sent. The *Local Bus Interface* module interfaces with the *Arbiter* module for the granting usage of the local bus. When the local bus is free, a Bus Grant signal is received immediately. The *Bus Request* will then be sent out the *Local Bus Interface* module, on the Local Bus, and enter the *System Bus Interface* module. When the local bus is busy, requests only come into the *Arbiter* when the local bus has been released. The *System Bus Interface* module first checks to see whether the *Bus Request* is in the cache memory. If so, this module

continues granting the usage of the local bus and sends out the *Bus Reply* by delaying a certain amount of time to model the communication cost of the local bus and the cache memory access time, goes back to the source *cluster* where the request was sent. Otherwise, the *Bus Request* is queued in the *System Bus Interface* module waiting to be transmitted to the *Host* module and the local bus can be released. When the system bus is free, the *Bus Request* is instantly granted use of the system bus, and flows to the *Host* module via the *System Bus Delays* module where the communication cost of the system bus is modelled. The *Bus Reply* finally goes back the *System Bus Interface* module after delaying the global memory access time and the communication cost of the system bus. Again, it will be queued in the *System Bus Interface* module waiting for the granting usage of the local bus. When the local bus is free, the *Bus Reply* will then be sent out the *System Bus Interface* module, on the Local Bus, and goes back to the source *cluster* where the request was sent.

5.3.4. Cell Traversal Unit

The *CTU* module traverses the space shellwise by using cell-traversal rays and outputs a set of *Cell IDs*. Fig. 5.14 shows an expansion of this module. The right output, Cell ID, outputs the *Cell ID* of each cell found. All the other outputs are used for synchronization purpose.

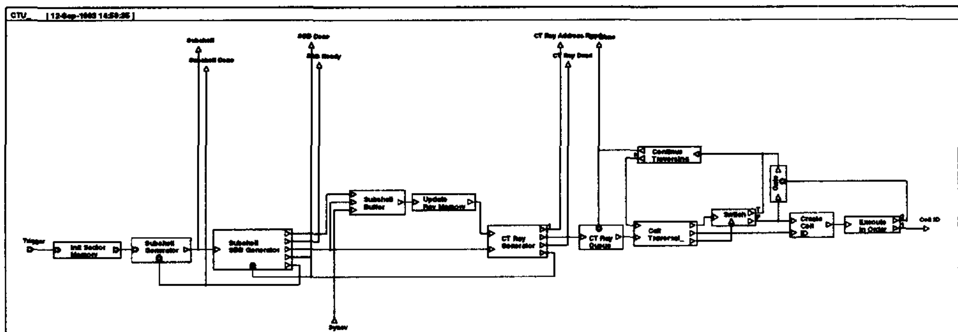


Figure 5.14: The *CTU* module.

The *CTU* module operates as follows. After filling up some required information on the *Init Sector memory*, the *Subshell Generator* module generates a set of *Subshells* by reading a patch address list from the memory named Subshell List Memory. Each entry in the patch address list will define a *Subshell* by addressing the memory named the Polygon Memory. The *Subshell SBB Generator* module computes the *SBB* field of the *Subshell*. Notice that the *SBB* should be clipped against the *Sector* of the *CTU* module stored in the memory named Sector Memory.

The *CT Ray Generator* module searches the bundle of cell-traversal rays addressed by an address window computed from the *SBB*. A *CT Ray* will be pushed into the *CT Ray Queue* module if it has not been traversed yet. Each *CT Ray* popped out of the *CT Ray Queue* module starts with cell traversal on the *Cell Traversal* module. The *Continue Traversing* module decides to continue cell traversal or pop out the next *CT Ray*, depending on whether the Shell boundary is reached or not. The Shell boundary is defined by the *r_max* field of the *SBB*. The *Cell ID* of each cell found goes out the right output to the *MEM* module.

5.3.4.1. Init Sector memory

The host supplies the required information for the *CTU* module by reading two files on *Init Sector Memory* module. The first is the Sector definition of the *CTU* module, which is stored in the memory named the Sector memory. The second is the patch address list, which is stored in the memory named the Subshell List memory.

5.3.4.2. Subshell Generator

The *Subshell Generator* module generates a set of normal *Subshells* based on the patches found in the low-density ray casting. Besides, an extra one called the last *Subshell* is appended to the normal *Subshells* in order to cover the whole space. This is because the space defined by the normal *Subshells* are possibly incomplete.

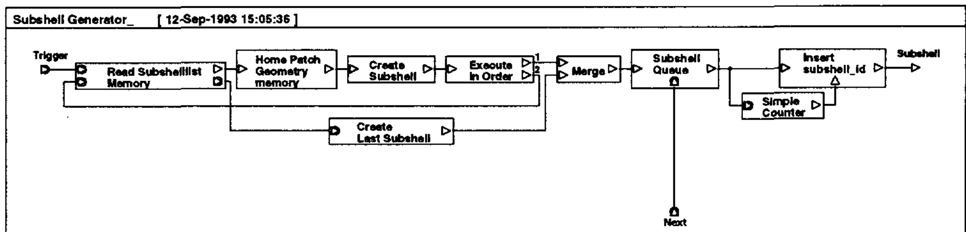


Figure 5.15: The *Subshell Generator* module.

This module is expanded in Fig. 5.15. The *Read Subshell List Memory* module reads a patch address list from the memory named the Subshell List Memory. Each entry of the patch address list creates a normal *Subshell* by reading a *Polygon* from the *Home Patch Geometry Memory* module. The *Create Last Subshell* creates and appends the last *Subshell* defined by the Sector definition of the *CTU* module to the normal *Subshells*. Note that the last *Subshell* is always the last to be processed. This is because not too much coherence can be exploited there as it is composed of the fragments of space. For synchronization purpose, an unique process identifier

subshell_id is assigned to each *Subshell* popped out of the *Subshell Queue* module before sending to the Subshell output.

5.3.4.3. Subshell SBB Generator

The *Subshell SBB Generator* module takes the 4 transformed vertices (i.e., represented in R-coordinate system) of a polygon, computes the *SBB*, and clips the *SBB* against the Sector definition of the *CTU* module. The expansion of this module is shown in Fig. 5.16.

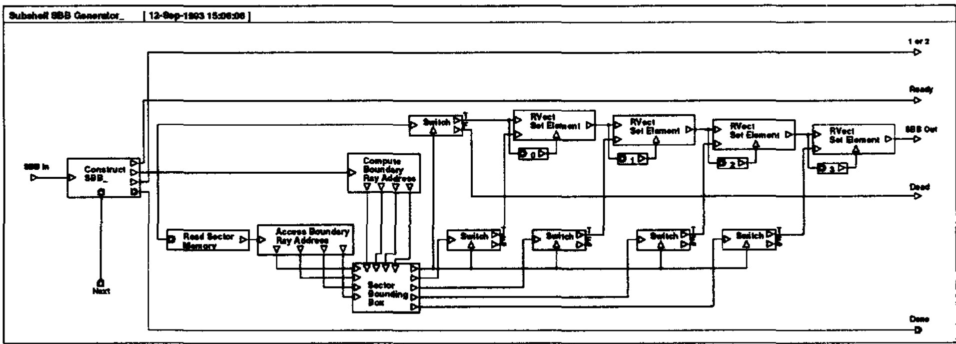


Figure 5.16: The *Subshell SBB Generator* module.

The *Subshell SBB Generator* module operates as follows. When a *Polygon* enters the left input from the *Read Subshell List* module, the *Construct SBB* module computes the *SBB* of the *Polygon* based on the method described in section 3.4.1.3. Basically, it computes the θ_{min}, θ_{max}, φ_{min}, and φ_{max} from the 4 transformed vertices of the *Polygon*. Based on the $\Delta\varphi = \varphi_{max} - \varphi_{min}$, the θ_{min} is adjusted by subtracting an error term $\varepsilon_\theta(\Delta\varphi)$ (refer to Eq. 3.6) from it, that is, $\theta_{min} = \theta_{min} - \varepsilon(\Delta\varphi)$. This is because the minimal θ angle doesn't happen at the vertices. The *Section Bounding Box* module clips the computed *SBB* against the Section definition of the *CTU* module. The Sector definition is defined in the *Sector* data structure stored in the memory named the Sector Memory.

5.3.4.4. CT Ray Generator

The *CT Ray Generator* module searches the bundle of cell-traversal rays addressed by an address window computed from the *SBB* of a Subshell. It decides whether a cell-traversal ray can be sent out or if it must be discarded or deferred.

An expansion of this module is shown in Fig. 5.17. The bottom left input accepts a *Subshell* data structure from the *Subshell Generator* module. Because cell-traversal rays are stored on the basis of local addresses. So the above-mentioned address window is computed based on the *SBB* field of the *Subshell* data structure and the Sector definition of the *CTU* module. More precisely, we have to offset the boundary addresses computed by the *SBB* by subtracting the base addresses computed by the θ_{min} and φ_{min} of the Sector from them. All the addresses within the address window are generated in the *CT Ray Address Generator* module by simply using two *Int Do* modules. Each address generated goes to the *CT Ray Memory* module.

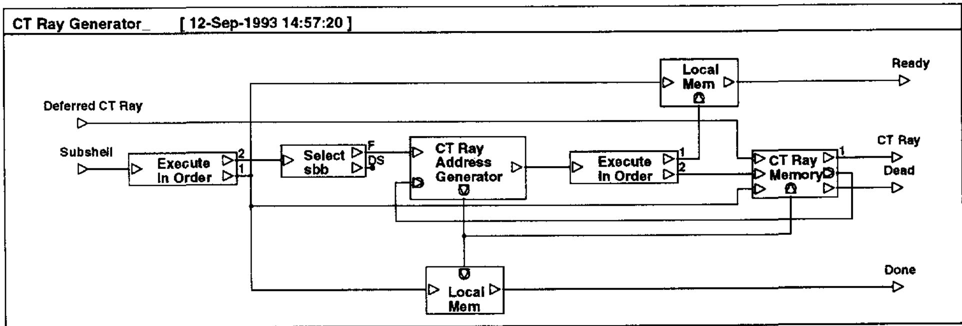


Figure 5.17: The *CT Ray Generator* module.

In section 5.2.4.2, we discussed the synchronization for cell traversal subprocess and proposed a synchronization algorithm (refer to Fig. 5.10) to meet the following requirements:

1. At any time instant, only one copy of a cell-traversal ray is allowed.
2. For each version of a cell-traversal ray, we can only start with cell traversal once.
3. A cell-traversal ray traverses the space in depth order.
4. Only remaining rays are allowed to continue traversing.

Actually, the *CT Ray Memory* module implements the synchronization algorithm completely. This module is expanded in Fig. 5.18. A *Ray Address* and a *Subshell* data structure enter the two bottom left inputs. Three different situations can happen to the cell-traversal ray addressed by the *Ray Address*:

1. This module first checks the *shell_id* of the cell-traversal ray stored in the memory named the Ray shell_id Mask against the *shell_id* field of the *Subshell* data structure. If the latter is larger than the former, then the cell-traversal ray can be discarded because it has already been traversed before.

2. In case the cell-traversal ray has not been traversed yet, this module continues to check its *presence_bit* stored in the memory named the Ray presence_bit Mask. If the *presence_bit* is set, then the cell-traversal ray must be deferred as it is still computing intersection in the *ICU* module. Then the *subshell_id* together with the *shell_id* of the deferred cell-traversal ray must be stored in the *CT Deferred Ray Memory* module.

3. Otherwise, a *CT Ray* data structure is read from the memory named the CT Ray Memory and goes out the upper right output.

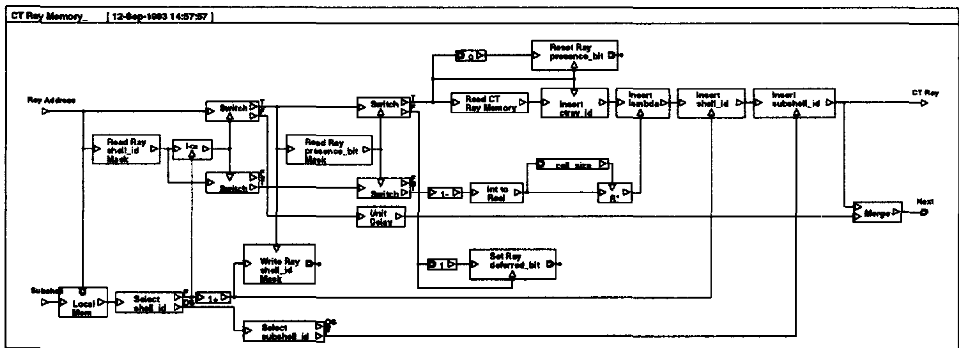


Figure 5.18: The *CT Ray Memory* module.

5.3.4.5. Cell Traversal

The *Cell Traversal* module is the core of the *CTU* module. It is used to open relevant cells by traversing the space in a certain way. What we built is a flexible module that can serve different cell structures like octree, macro region and voxel. It can also support the grid-ray approach.

The expansion of this module is shown in Fig. 5.19. It consists of two primitives labelled *_Generate Starting Cells* and *_Generate Cells*. For each new *CT Ray* enters the bottom left input, the *_Generate Starting Cells* primitive generates its starting cell. This primitive determines the starting cell for cell traversal. For this purpose, we first determine the starting point for cell traversal as follows. The ray is defined by an origin $\vec{O}$ and a unit direction vector $\vec{d}$ as $\vec{O} + \lambda \vec{d}$. So the starting point is given by multiplying the *lambda* field of the *CT Ray* with the *dir* field of the *CT Ray*, then adding the *or* field of the *Ray Frustum* data structure to the result. For uniform grid data structure, the *Cell ID* of the starting cell can be obtained by dividing each coordinate of the starting point with the cell size, then taking the *ceiling* function of the result. In order to find the next cell, it requires to update the *lambda* and the *direction*

fields of the *CT Ray*. To find the new *lambda*, we first intersect the *CT Ray* with the three pairs of parallel bounding planes of the starting cell. For each pair of parallel bounding planes, there is one maximal intersected distance. Then the new *lambda* is the minimal of the three maximal intersected distances. Normally, the new *direction* is just a unit vector perpendicular to the bounding plane where the new *lambda* happens. If the new *lambda* happens at an edge sharing by two bounding planes or a point sharing by three bounding planes, then the new *direction* is a unit vector determined by adding the unit vectors perpendicular to those bounding planes.

After updating, the *CT Ray* goes to the *Continue Traversing* module. The *Continue Traversing* module checks to see if the *CT Ray* reached the Shell boundary or the environment boundary. If this is the case, the *CT Ray* is done after updating its *lambda*. Otherwise, it goes to the *_Generate Cells* primitive and continues to do cell traversal. This primitive is exactly the same as the *_Generate Starting Cells* primitive. The reason of doing this is that we can assign different delay time for each primitive. This is necessary because the *_Generate Starting Cells* primitive generally takes more time than the *_Generate Cells* primitive due to some initialization steps [TI86].

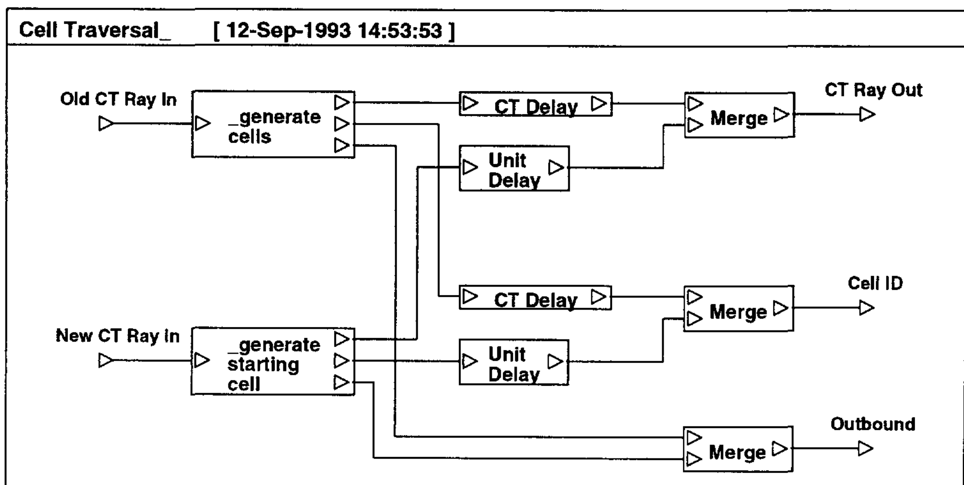


Figure 5.19: The *Cell Traversal* module.

5.3.5. Intersection Computation Unit

The *ICU* module computes the intersection points between a polygon and the bundle of intersection-computation rays determined by the SBB of the polygon.

The expansion of the *ICU* module is shown in Fig. 5.20. This module operates as follows. When a *Polygon* enters the left input of the *ICU* module from the *MEM* module, it is queued in the *Polygon Queue* module due to the changing of data rate. Because the intersection computation is done in the R-coordinate system, the vertices of a *Polygon* popped out of the *Polygon Queue* module undergoes a G to R coordinate transformation in the *G2R Geometry Transformation* module. The *SBB* of the *Polygon* is then computed by using the transformed vertices (i.e., in the R-coordinate system) on the *Polygon SBB Generator* module. The *IC Ray Generator* module searches the bundle of intersection-computation rays addressed by an address window computed from the *SBB*. An *IC Ray* will be pushed into the *IC Ray Queue* module if its *intd* field is larger than the *r_min* field of the *SBB*. Each *IC Ray* popped out of the *IC Ray Queue* module together with its defining *Polygon* flow into a pipelined *Intersection Computation* module to compute intersection point. When all the *IC Rays* defined by the *SBB* have been consumed, the *Count Match* module generates a signal to pop out the next *Polygon* from the *Polygon Buffer* module.

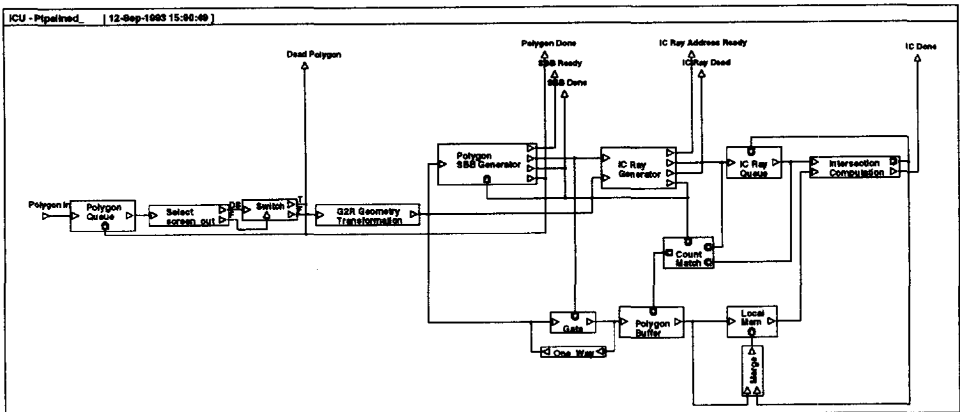


Figure 5.20: The *ICU* module.

5.3.5.1. G2R Geometry Transformation

The *G2R Geometry Transformation* module transforms the four vertices of a *Polygon* from the G-coordinate system (i.e., *vertex*[*0*] - *vertex*[*3*]) to the R-coordinate system (i.e., *tr_vertex*[*0*] - *tr_vertex*[*3*]). The top level of this module is shown in Fig. 5.21a. When a *Polygon* enters the left input, it triggers the *Read Ray Frustum* module to read the *or* and the *normal* fields of the *Ray Frustum* data structure stored in the memory named Ray Frustum Memory. They

altogether go into the *TransformPatch2R* module to perform the geometry transformation.

(a) The *G2R Geometry Transformation* module.

(b) The *TransformPatch2R* module.

(c) The *TransformPoint2R* module.

Figure 5.21: Geometry transformation modules.

The *TransformPatch2R* module is expanded in Fig. 5.21b. It operates as follows. The *normal* comes in the *DetermineAnglesR* module where its θ and φ angles in the G-coordinate system are computed. The computed θ and φ angles together with the *or* and each vertex of the *Polygon* go into the *TransformPoint2R* module.

The *TransformPoint2R* module is responsible for translating and rotating each vertex of the *Polygon* to the R-coordinate system. The expansion of this module is shown in Fig. 5.21c. It translates each vertex of the *Polygon* according to the *or* by using the *VecSub* module, rotates the translated vertex along the z-axis through an angle $-\varphi$ and then rotates along the y-axis through an angle $-\theta$. The latter two rotations are performed in the *RotateZ* and the *RotateY* modules, respectively. The transformed vertex goes out the right output and is used to insert the transformed vertex field (i.e., *tr_vertex*[*0*] - *tr_vertex*[*3*]) of the *Polygon.*

5.3.5.2. Polygon SBB Generator

The *Polygon SBB Generator* module, as shown in Fig. 5.22, is much like the previously described *Subshell SBB Generator* module. The only difference is that the computed *SBB* is now clipped against the Section definition of the *ICU* module instead of the Sector definition of the *CTU* module.

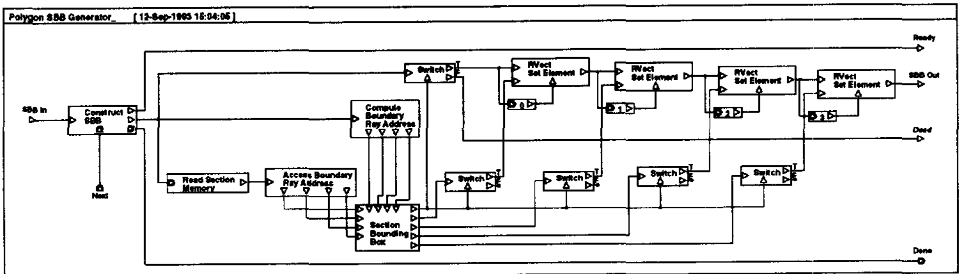


Figure 5.22: The *Polygon SBB Generator* module.

5.3.5.3. IC Ray Generator

The *IC Ray Generator* module searches the bundle of intersection-computation rays addressed by an address window computed from the *SBB* of a polygon. It decides whether an intersection-computation ray has to be taken or if it can be discarded.

This module's expansion can be seen in Fig. 5.23. It consists of the *IC Ray Address Generator*

module and the *IC Ray Memory* module. It is much like the previously described *CT Ray Generator* module. The only major difference is that a cheap lazy isolation (refer to Fig. 5.12) is used instead of deferring intersection-computation rays. The *IC Ray Memory* module is expanded in Fig. 5.24. This module accepts a *Ray Address* and a *Polygon* from the left two inputs. As you can see, the lazy isolation is simply done by comparing the current *intd* field of the *IC Ray* addressed by the *Ray Address* with the *r_min* of the *Polygon*. If the latter is larger than the former, then the *IC Ray* can be discarded. Otherwise, the *IC Ray* data structure is read from the memory named IC Ray Memory, and goes out the upper right output.

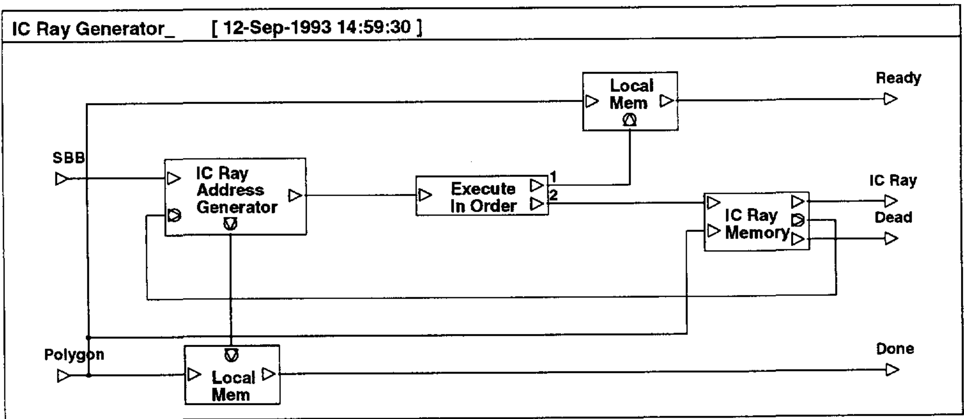


Figure 5.23: The *IC Ray Generator* module.

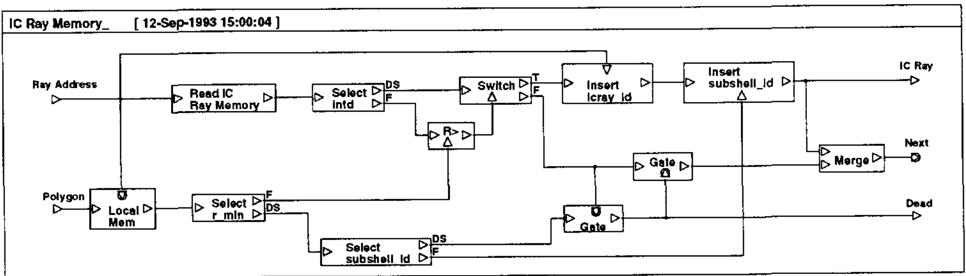


Figure 5.24: The *IC Ray Memory* module.

5.3.5.4. Intersection Computation

The *Intersection Computation* module is the core of the *ICU* module. It is used to compute intersection points between a bundle of intersection-computation rays and a polygon in a pipelined fashion. There are many different ways of building this module. The one shown in Fig. 5.25 is our current implementation. It consists of two primitives labelled *_Screen* and *_Subdivide*, and three modules labelled *Xproduct, PreDistance* and *Update IC Ray Memory.*

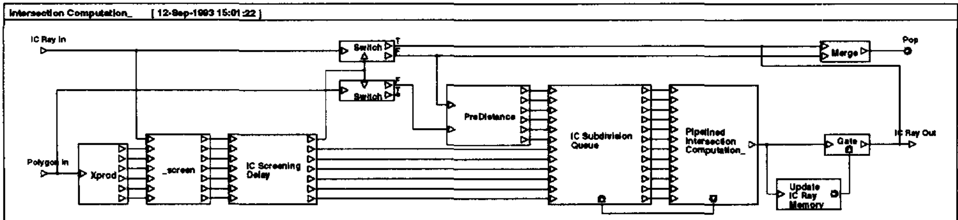


Figure 5.25: The *Intersection Computation* module.

The *Intersection Computation* module operates as follows. When a *Polygon* enters this module from the lower left input, the *Xproduct* module computes the normal vectors of the four bounding planes[10] (see Fig. 5.26) of the *Polygon* by taking the cross product of two vectors $\overrightarrow{OV_i}$ and $\overrightarrow{OV_j}$, where $i = 0, ..., 3$ and $j = (i + 1)$ modulo 4. This can be seen by expanding the *Xproduct* module in Fig. 5.27. Each *VecCrossProd* module takes two contiguous transformed vertices of the *Polygon* and computes their cross product.

We use the pseudo-distance from an *icray* to a bounding plane to decide on which side of the bounding plane the intersection point lies. The pseudo-distance $delta_{ij}$ from the *icray* to the bounding plane OV_0V_1 is defined as the inner product of the direction vector of the *icray* and the normal vector of the bounding plane OV_iV_j. The *_Screen* primitive computes the pseudo-distances $delta_{01}$, $delta_{12}$, $delta_{23}$ and $delta_{30}$ by taking the inner product of the *dir* field of the *IC Ray* and the four normal vectors of the bounding planes computed in the *Xproduct* module. It checks to see if any pseudo-distance is less than zero. If this the case, the *IC Ray* will not intersect the *Polygon*, and can be screened out. Otherwise, the *IC Ray* cannot be screened out and two more auxiliary pseudo-distances $delta_{02}$ and $delta_{13}$ are computed. This screening is

[10] The bounding plane for two contiguous vertices, say V_i and V_j, of a polyogn is defined as the plane containing the origine O and the two vertices V_i and V_j, and is denoted as OV_iV_j.

quite effective because the *SBB* of a polygon generally overestimates 50% of the number of rays hit the polygon. Note that the scan conversion used in the Pixel-Planes [Fuc85] [Pou85] is a degenerate case (i.e., a two-dimensional case) of this method. This can be understood as follows. After transforming a polygon into the eye coordinate system, the polygon's edges are encoded in linear equations of the form $Ax + By + C = 0$. The Pixel-Planes scan-converts the polygon by checking the sign of the linear expression $F(x, y) = Ax + By + C$, where x, y is the pixel's location on the screen. All the pixels outside the bounding line (i.e., the sign of F is negative) will be disabled. Only those inside participate the further visibility and shading calculations.

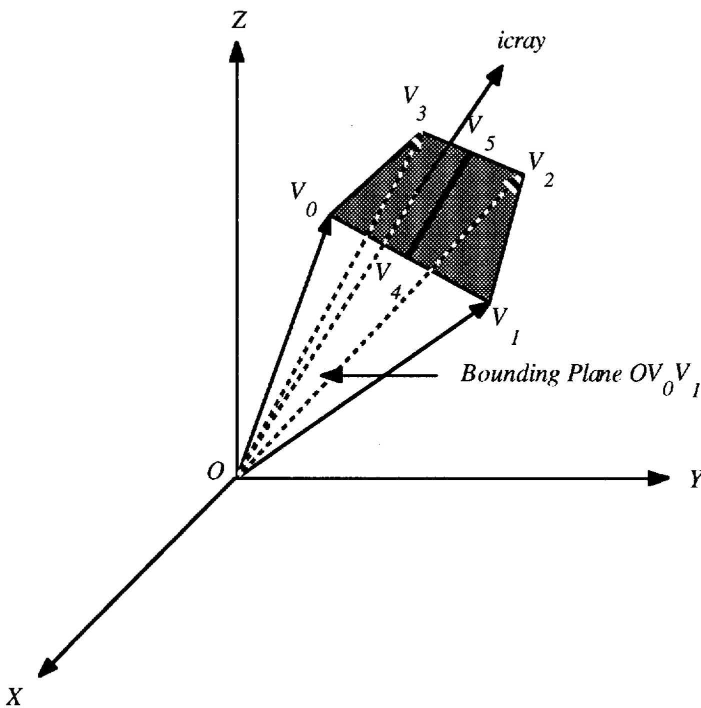


Figure 5.26: The bounding plane OV_0V_1 and one subdivision step in u direction.

For determining the (u, v) coordinates of the intersection point, the intersection point and the distance to the intersection point, we rely on a subdivision procedure. The *_Subdivide* primitive is responsible for the subdivision of a polygon. Basically, the subdivision of a polygon is to bipartite the original polygon into two new polygons by taking alternatively u and v direction. The two new polygons are then tested to decide in which one the intersection point lies. The

one intersected will be chosen for the next subdivision procedure. This is repeated until a certain level is reached such that the resultant polygon is accurate enough to be treated as the intersection point. The (u, v) coordinates of the intersection point can be derived directly by the decision made in each subdivision step. To clarity this, we illustrate with one subdivision step in u direction. As shown in Fig. 5.26, we bipartite a polygon $V_0V_1V_2V_3$ into two polygons, $V_0V_4V_5V_3$ and $V_4V_1V_2V_5$, where V_4 and V_5 are the midpoints of the edges V_0V_1 and V_2V_3, respectively. Based on the pseudo-distances $delta_{01}$, $delta_{12}$, $delta_{23}$, $delta_{30}$, $delta_{02}$ and $delta_{13}$ computed in the *_Screen* module, the pseudo-distance $delta_{45}$ can be derived as follows:

$$delta_{45} = (-delta_{30} + delta_{13} + delta_{02} + delta_{12})/4.$$

If $delta_{45} \geq 0$, then the left polygon $V_0V_4V_5V_3$ is selected. Otherwise, the right polygon $V_4V_1V_2V_5$ is selected. Upon deciding which polygon is taken, the new pseudo-distances must be updated accordingly. Then we take another subdivision step in v direction. After a number of steps, the algorithm converges such that the resultant vertices approach the intersection point. The distance to the intersection point can be derived in a similar way as the intersection point but the distance from the origin O to each vertex is used instead of the vertex. The *PreDistance* module computes the distance from the origin O to each vertex of the original polygon. For more details about the subdivision procedure, we refer to [Hek93*a*].

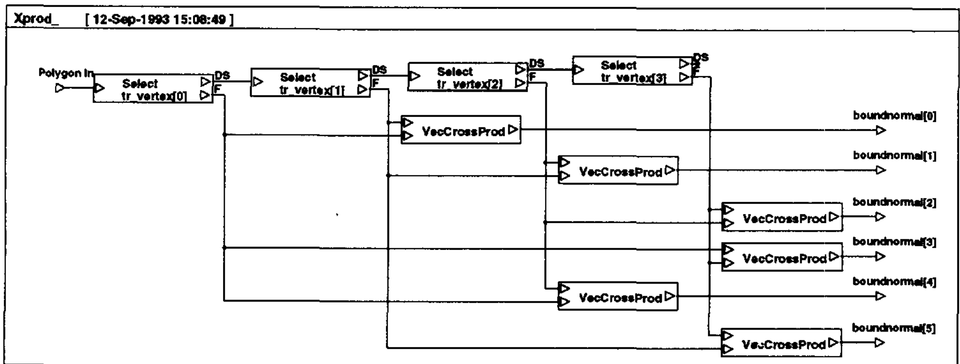


Figure 5.27: The *Xproduct* module.

5.3.6. Memory System

The expansion of the *MEM* module is shown in Fig. 5.28. The left input accepts *Cell ID* from

the *CTU* module. The right bottom four signal ports are used to interface with the *System Bus Interface* module for sending or receiving global requests or replies. The right output sends *Polygon* to the *ICU* module.

The *MEM* module operates as follows. When a *Cell ID* enters the *MEM* module from the *CTU* module, it is queued in the *Cell ID Queue* module due to the changing of data rate. If a cell has already been opened, it is masked in the *Cell Filter* module. Otherwise, it enters the *Cell Address Table* module where a *Cell Request* is created after reading the Local Cell Address Table. Based on the *destination_id* field of the *Cell Request*, the *Local/Global Cell Request ?* module decides whether it is a local or global request. If the *Cell Request* is global, it is packed into *Request* message in the *Local Bus Interface* module and sent out to the *System Bus Interface* module. Otherwise, it goes to the *Local Patch Address Table* module where a *Patch ID* is created by reading the Patch Address Table. The *Patch ID* is queued in the *Patch ID Queue* module, and then checked to see if it has already been found or not in the *Patch Filter* module. If it is a new *Patch ID*, a *Patch Request* is created after reading the Patch Type Memory in the *Patch Type Memory* module. A global *Patch Request* will be packed into *Request* message in the *Local Bus Interface* module and sent to the *System Bus Interface* module. A local *Patch Request* goes to the *Local Patch Geometry* module, where a *Polygon* is created and sent to the *ICU* module. A global *Cell Reply* or *Patch Reply* enters the *Local Bus Interface* module, where a *Patch ID* or a *Polygon* is unpacked and sent to *Patch ID Queue* module or the *ICU* module, respectively.

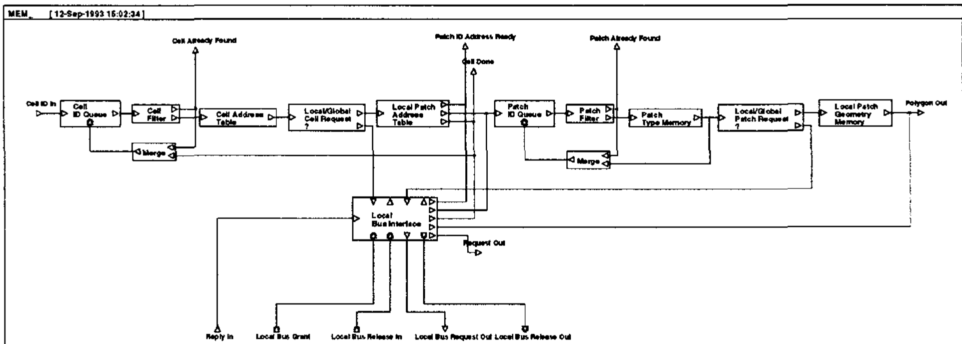


Figure 5.28: The *MEM* module.

5.3.6.1. Local Bus Interface

The *Local Bus Interface* module is used to interface a processor on the radiosity engine to the Local Bus. It packs global requests to be transmitted to the Local Bus and unpacks global

replies received from the Local Bus. Furthermore, it interfaces with the *Arbiter* module for the granting usage of the Local bus.

This module's expansion can be seen in Fig. 5.29. It consists of the *Cell Bus Interface* module, the *Patch Bus Interface* module and the *Request Queue* module. The top two inputs accept global *Cell Request* and global *Patch Request*, respectively. The right bottom input receives *Reply* from the *System Bus Interface* module. The left bottom four signal ports are used for bus arbitration.

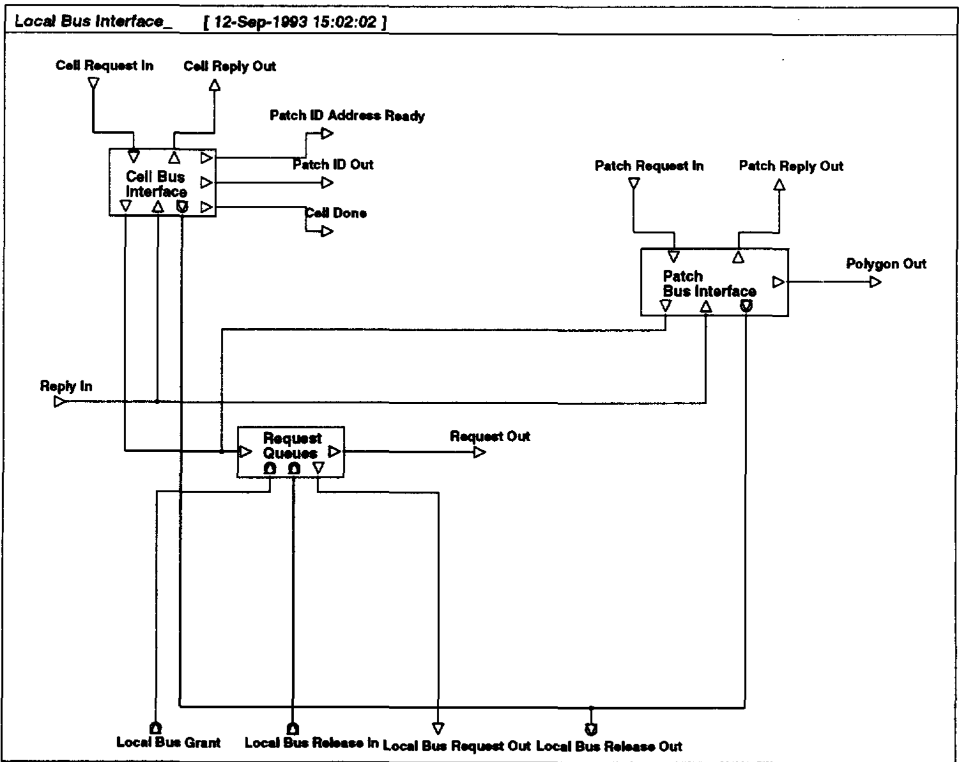


Figure 5.29: The *Local Bus Interface* module.

The *Local Bus Interface* module operates as follows. When a global request enters the *Cell Bus Interface* or the *Patch Bus Interface* module, it is packed into a *Request* message by inserting the *request_type* and the *source_id* fields. The *Request* is then queued in the *Request Queues* module. Each time a new request comes into the *Request Queues* module, the current time also

goes out the Local Bus Request output if the Local Bus is not busy. The arbitration scheme guarantees that if the Local Bus is not busy, a Local Bus Grant signal is received immediately. The *Request* queued will then be released from the *Request Queues* module, sent out the *Local Bus Interface* module, and on the Local Bus, goes to the *System Bus Interface* module. If the Local Bus is busy when the *Request* comes into the *Request Queues* module, it is placed in the queue and will only be released when the Local Bus is granted to it.

The *Local Bus Interface* module also receives *Reply* message coming from the *System Bus Interface* module via the Local Bus. The *Reply* only comes into the *Local Bus Interface* module of a processor whose *processor_id* is matched with the *source_id* field of the *Reply*. The *Local Bus Interface* module unpacks the *Reply* into *Cell Reply* or *Patch Reply* according to the *reply_type* field of the *Reply*. The *Cell Reply* or *Patch Reply* goes into the *Cell Bus Interface* or the *Patch Bus Interface* module, respectively, to create the *Patch ID* or *Polygon*.

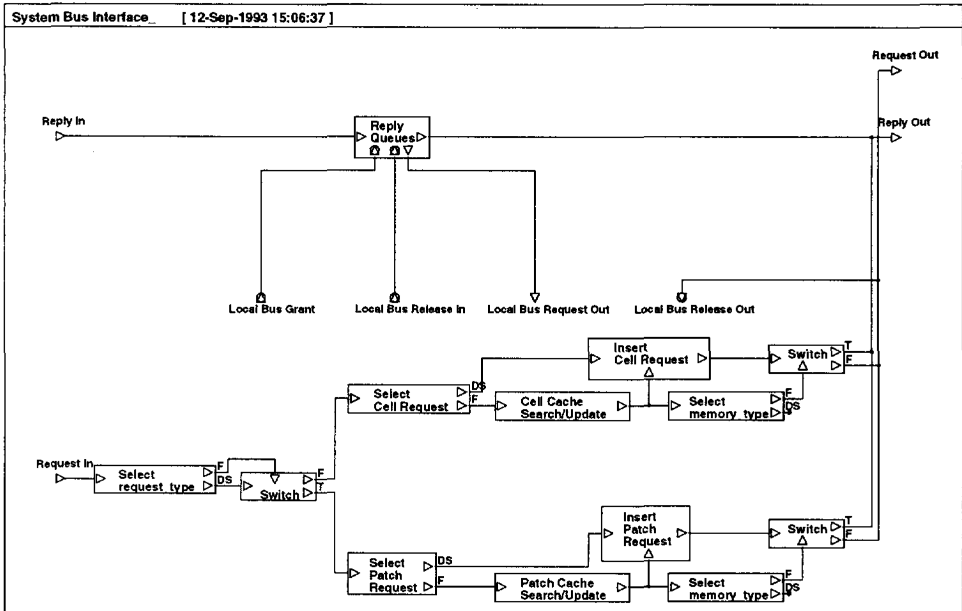


Figure 5.30: The *System Bus Interface* module.

5.3.7. System Bus Interface

The radiosity engine is connected to the host via the *System Bus Interface* module. The *System Bus Interface* module provides the communication protocols to receive and transmit data and

commands. Besides, a global cache memory is built in this module to reduce the effective memory access time of reading patch address list, patch type and patch geometry.

The expansion of the *System Bus Interface* module is shown in Fig. 5.30. The left input accepts *Request* from the *Cell Bus Interface* module or the *Patch Bus Interface* module. The bottom three signal ports are used for system bus arbitration. The two signals on the right side are used to send *Cell Reply* or *Patch Reply* back to the *Cell Bus Interface* module or the *Patch Bus Interface* module requesting the system bus.

The *System Bus Interface* module operates as follows. When a *Request* enters the Request In input, it is either sent to the *Cell Cache Search/Update* module or to the *Patch Cache Search/Update* module to check if it is in the cache memory, depending on the *request_type* field of the *Request*. If it hits, then the *Reply* goes back to the *Cluster* module where the *Request* was sent. Otherwise, the *Request* goes to the Request Out output and the Local Bus can be released immediately. The *System Bus Interface* module also receives the *Reply* coming from the Host via the System Bus. When a *Reply* enters the Reply In input, it is queued in the *Reply Queues* module. Each time a *Reply* comes into the *Reply Queues* module, the current time also goes out the Local Bus Request Out output if the Local Bus is not busy. The arbitration scheme guarantees that if the Local Bus is not busy, a Local Bus Grant signal is received immediately. The *Reply* queued will then be released from the *Reply Queues* module, sent out the Reply Out output, and on the Local Bus, goes to the appropriate component. If the Local Bus is busy when the *Reply* comes into the *Reply Queues* module, it is placed in the queue and will only be released when the Local Bus is granted to it.

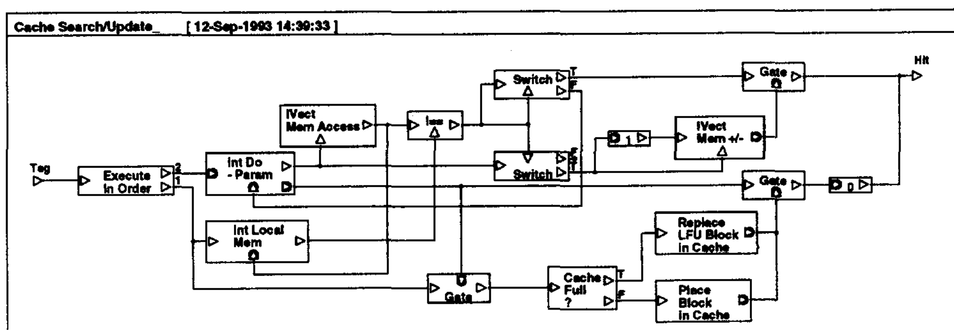


Figure 5.31: The *Cache Search/Update* module.

5.3.7.1. Cache Search/Update

This module checks to see if a request is in the cache memory and updates the cache by including the new request and removing the one that has been referenced the least number of times.

The expansion of the *Cache Search/Update* module is shown in Fig. 5.31. A tag (*cell_id* or *patch_id*) of the request requiring a cache status check enters the left input. The left *8* modules determine if the request is in the cache by comparing the tag with the contents of the memory named Cache Tag Memory on the *I==* module. If the tag is found to be equal to any content of the Cache Tag Memory, it means the request is in the cache. In this case, we increment the number of times the request has been referenced that is stored in the memory named Cache Count Memory and set the Hit output to 1. Otherwise, we update the cache by using the *Cache Full?* module, the *Replace LFU Block in Cache* module and the *Place Block in Cache* module, and set the Hit output to 0.

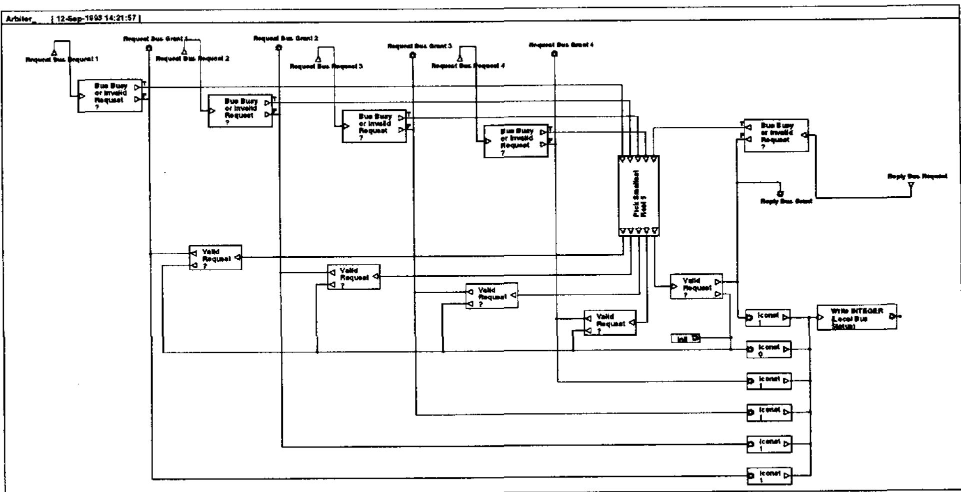


Figure 5.32: The *Arbiter* module.

5.3.8. Arbiter

An expansion of the *Arbiter* module is shown in Fig. 5.32. This module receives *Request* bus requests from the 4 *Cluster* modules and *Reply* bus requests from the *System Bus Interface* module. If a request is valid and the Local Bus is not busy, a bus grant signal is immediately

sent back to the *Cluster* or the *System Bus Interface* module where the request was sent and the memory named Local Bus Status is set to 1 by the *IConst 1* module and the *Write Integer* (*System Bus Status*) module. Otherwise, the *Pick Smallest Real* module picks the request which has been in the queue the longest and issues a bus grant signal to the appropriate component. If none of the requests is valid, the Local Bus is returned to the free state by sending a signal to the *IConst 0* module. A request is invalid if no request is in the queue that can be transmitted when the bus is being released.

Chapter 6

Performance Measurements

The previous chapter described the system configuration and outlined the functionality of the target system modelled in BONeS®. This chapter quantifies the performance of such a system by using Monte Carlo simulation. The performance measurements in this chapter are broken into three major sections. The first section examines the effects of different network topologies on the system performance. The second section summarizes the performance metrics of the memory system. The final section discusses the overall system performance.

6.1. Network Performance

In section 5.2.3.1, we compared different network topologies in terms of the number of hops required. In this section, we shall examine the effects of different network topologies on the system performance. We first describe how to model the communication time over the interconnection network.

Let $\sigma_s = \sigma_{ss} + \sigma_{sd}$ be the node-to-node communication latency, where σ_{ss} is the time spend at source to initiate a communication and σ_{sd} is the time spend at destination to terminate a communication. Furthermore, let τ_t be the channel transmission time per byte, and let τ_r and $\tau_{r'}$ be the time required to make a routing decision for store-and-forward and wormhole routings, respectively. Consider an s-byte message passing from a source node n_s to a destination node n_d with the distance h hops. The time required for store-and-forward routing is:

$$t = \sigma_s + (h - 1)\ \tau_r + h\ \tau_t\ s$$
$$= h\ (\tau_r + \tau_t\ s) + \Delta,$$

where $\Delta = \sigma_S - \tau_r$ does not depend on the number of hops, h, in the communication. For the case of wormhole routing, the time required is:

$$t = \sigma_s + (h - 1)\ \tau_r' + \tau_t\ s + (h - 1)\ \tau_t\ s'$$
$$= h\ (\tau_r' + \tau_t\ s') + \Delta',$$

where s' is the size in byte of the header flit (i.e., address field) of the message and $\Delta' = \tau_t\ (s - s') + (\sigma_S - \tau_r')$ does not dependent on the number of hops, h, in the communication. Note that $\tau_r' << \tau_r$ and $s' << s$. Thus, $\tau_r' + \tau_t\ s' << \tau_r + \tau_t\ s$, which implies that the wormhole routing is much faster than the store-and-forward routing.

The *Compute LM Delay* module, shown in Fig. 6.1, is used to model the communication time discussed in the above. As you can see, the *Routing Delay* module is used to model τ_r or τ_r', the *Network Delay* module is used to model $\tau_t\ s$ or $\tau_t\ s'$ and the *Network Bus Set_up Delay* module is used to model Δ or Δ'.

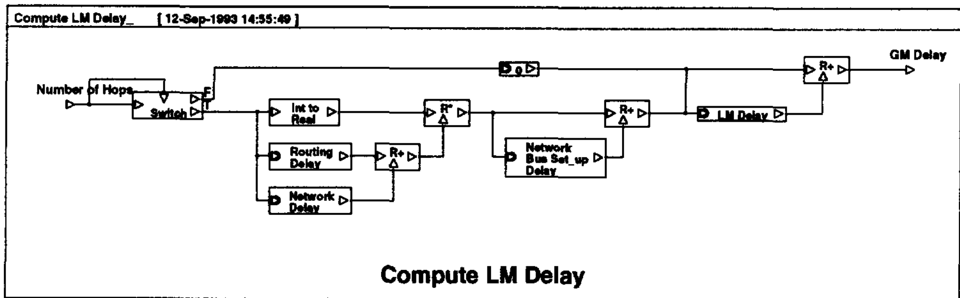


Figure 6.1: The *Compute LM Delay* module.

Two test scenes, *lobby* and *room16.27* (see Plate 1 and Plate 2 in Color Section), are chosen for evaluating the performance of different network topologies. The scene *lobby* contains a number of large patches, while the scene *room16.27* constitutes of the scene *lobby* and many more small patches. Test conditions are listed in Table 6.1. We choose a large cell size (0.5) so

that most cell/patch data can either be referenced from local clusters[11] directly or from remote clusters via the interconnection network. Tables 6.2 and 6.3 show the overall computation time for the four network candidates: Illiac, Torus, Hypercube and Ring. It turns out that the two-dimensional mesh (i.e., Illiac or Torus) provides a good performance for both 8-processor and 16-processor configurations. The Ring network is only acceptable for small configurations. When using 16 or more ICUs, the overall computation time for the Ring network is almost twice as other networks. For a 16-processor configuration, the Hypercube network becomes a torus but with different wrap-around as the Torus network. The performance of the Torus network appears to be better than the Hypercube network because the wrap-around at boundary links meets the traffic requirement.

Table 6.1: Test Conditions for Network Performance Measurement.

Scene	# of patches	# of ic rays	cell size
lobby	227	16384	0.5
room16.27	1297	16384	0.5

Table 6.2: The overall computation time for scene *lobby*.

# of ICUs	Illiac	Torus	Hypercube	Ring
8	7390.2	7390.2	-	7444.1
16	4425.7	4425.7	4425.7	6178.2

Table 6.3: The overall computation time for scene *room16.27*.

# of ICUs	Illiac	Torus	Hypercube	Ring
8	10937.2	10937.2	-	11083.6
16	16938	14201.2	15661.4	26163.4

[11] A local cluster is the cluster that contains the processor which issued the memory request. Otherwise, it is called a remote cluster.

6.2. Memory Performance

Memory performance is measured in terms of the *effectiveness* and the *cumulative usage frequency* of resident set, and the number of patch requests issued to different hierarchies of the memory system in the course of a ray-casting procedure. Five test scenes, *lobby*, *room16.27*, *array of cubes*, *man*, and *phone*, as depicted in Color Section (Plate 1 to Plate 5), are chosen for this purpose. In order to investigate the effects of different sample points on the memory performance, we partition the ceiling of a scene into 16 elements by using binary partition, then define 16 sample points (see Fig. 6.2) at the center of each element. Sample Point 1 is chosen as the reference point where we proceed with the scheduling task. Once a partitioning is determined by the scheduler, it is kept unchanged for all other sample points in the same phase. The test conditions for memory performance measurement are listed in Table 6.4.

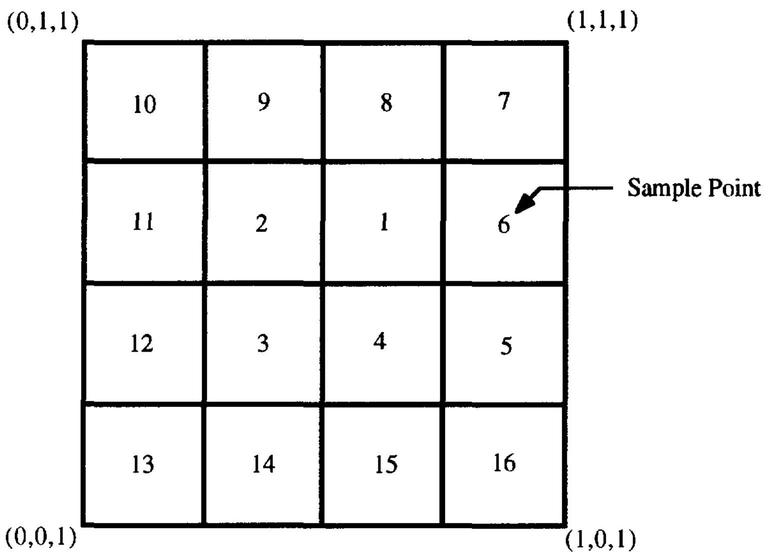


Figure 6.2: The 16 sample points defined on the ceiling of a scene.

Table 6.4: Test Conditions for Memory Performance Measurement.

Scene	# of patches	# of ic rays	cell size	Hi_Low_Ratio
lobby	227	16384	0.0625	16, 64, 256
room16.27	1297	16384	0.0625	16, 64, 256
array of cubes	3079	16384	0.0625	16, 64, 256
man	3315	16384	0.0625	16, 64, 256
phone	4103	16384	0.0625	16, 64, 256

1. The Effectiveness of Resident Set (refer to Figs. 6.3 and 6.4)

   We only show the results of *lobby* and *room16.27* for 4 sample points as labelled 4, 7,10 and 13 in Fig. 6.2. Other scenes actually show similar results. In general, the effectiveness of resident set decreases with the increased Hi_Low_Ratio, in particular when a scene contains many small patches like *room16.27*. Fig. 6.3 indicates that E(k) decreases from 0.8 to 0.6 when Hi_Low_Ratio increases from 16 to 256. As for scene *lobby*, the influence of Hi_Low_Ratio on E(k) is much less than the case of *room16.27* because patches in the scene are generally large. In both cases, E(k) is going off when a sample point is moving away from the reference point. However, the effectiveness of resident set can be as high as 0.7 for all sample points when Hi_Low_Ratio is chosen as 64 (i.e., 256 low-density rays). As compared with the high-density rays (1M or more for antialiasing), we conclude that a highly effective resident set can be selected by low-density ray casting with a small overhead relative to the cost of high-density ray casting.

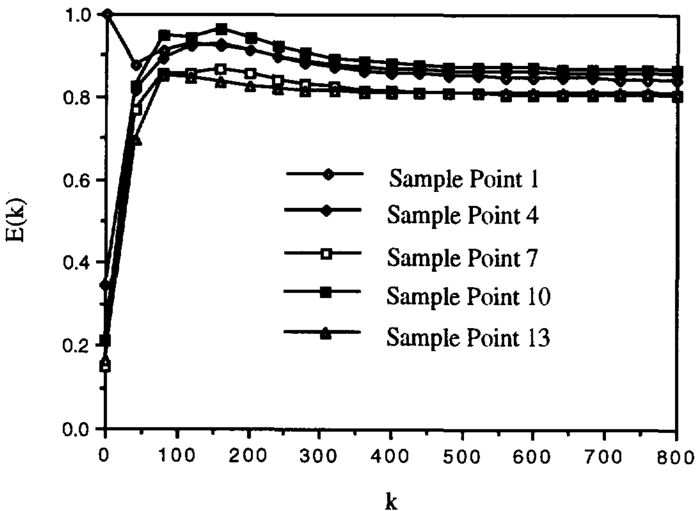
2. The Cumulative Patch Usage Frequency (refer to Tables. 6.5 and 6.6)

   The cumulative usage frequency of resident set is defined by replacing the k in the denominator of Eq. 5.1 with the total number of patches in a scene. It indicates the percentage of references issued to processor memories constituting of local memories and working registers (see Fig. 5.3) in the course of a ray-casting procedure. Due to the medium-sized cells, cell requests will incur much less communication overhead as compared with patch requests. From now on, our emphasis will be on patch requests. We assume that all the patches found in the low-density ray casting can be stored in local memories. When Hi_Low_Ratio is chosen as 64, about 70% - 80% of references goes to processor memories by storing only 4% and 25% of the model databases for *lobby* and *room16.27*, respectively. This implies that relatively small subsets of model databases will account for a large proportion of the references made during a ray-casting procedure and a highly effective resident set can be selected by the low-density ray casting. Note that N' is the total number of patches found in the low-density ray casting and CUF_i is the

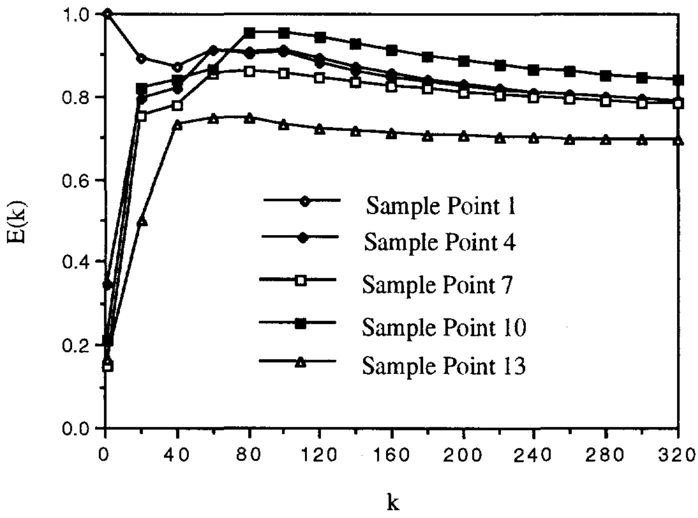
cumulative patch usage frequency referring to Sample Point i.

3. Number of GM/CM/LM Patch Requests (refer to Figs. 6.5 and 6.6)

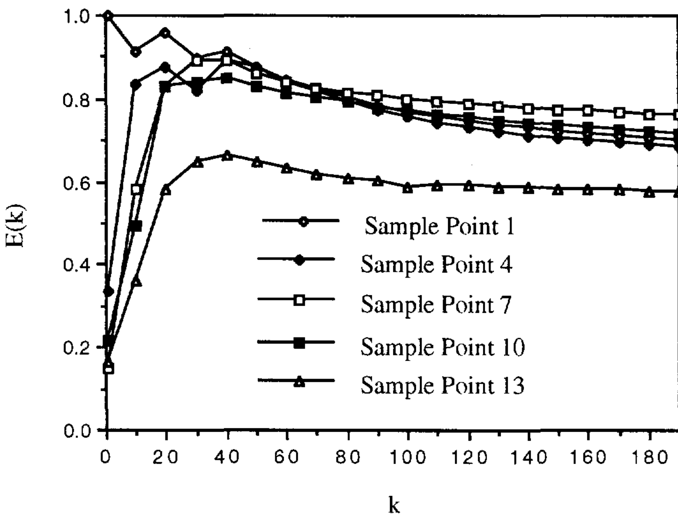
   For complex scenes containing hundreds of thousands or even a million patches, it is not practical to store all patch data in local memories. Moreover, patches stored in the local memories are determined by casting low-density rays from a reference point. Only some percentage of relevant cells/patches will be captured during the low-density ray casting and pre-loaded to local memories. Consequently, patch requests may be issued to global memory, cache memory or local memories for a ray-casting procedure referring to the reference point. Fig. 6.5 shows the number of patch requests issued to those memories. It turns out that the number of patch requests issued to global memory remains to be the same as the system grows by adding more clusters, which may become a fundamental bottleneck to the system scalability. To avoid this bottleneck, a cache memory is introduced by caching most frequently referenced patch data during a phase. The cache memory takes effects on other sample points in the same phase. Instead of retrieving patch data from the low bandwidth global memory, most of them can be referenced from the cache memory due to the data-coherence property. As can be seen from Fig. 6.6, most global memory requests in Fig. 6.5 become to be cache memory requests. Since the cache memory is faster than the global memory, some improvement in system performance can be expected.



(a) *Hi_Low_Ratio* = 16

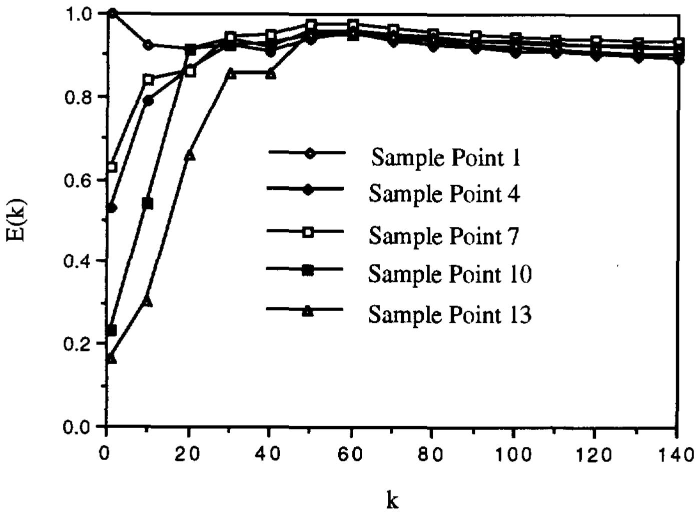


(b) *Hi_Low_Ratio* = 64

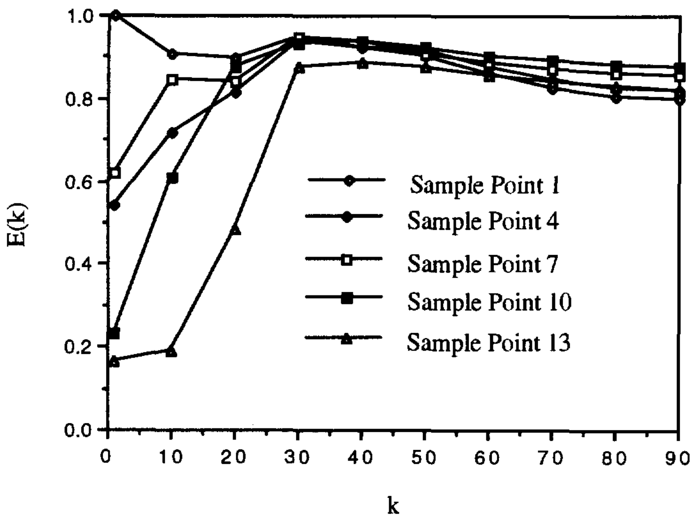


(c) *Hi_Low_Ratio* = 256

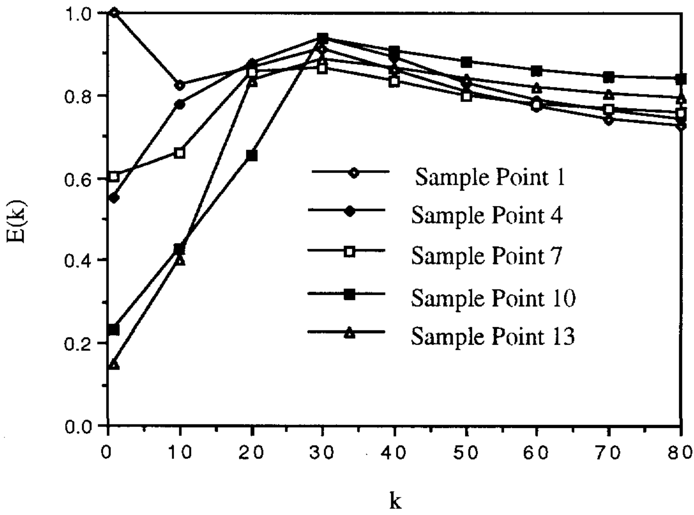
Figure 6.3: The Effectiveness E(k) vs k for scene *room16.27.*



(a) *Hi_Low_Ratio* = 16



(b) *Hi_Low_Ratio* = 64



(c) *Hi_Low_Ratio* = 256

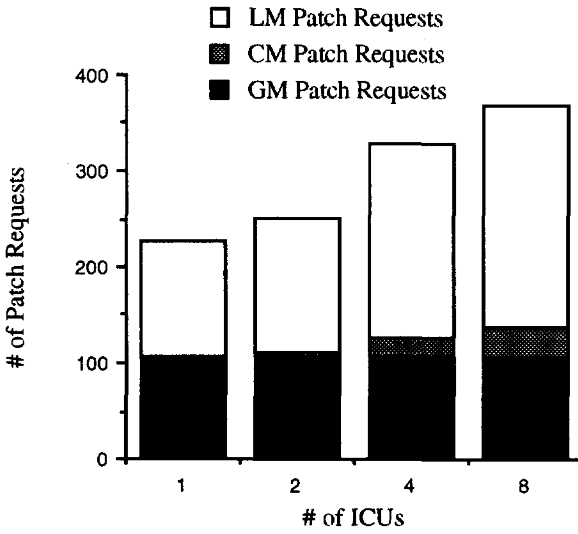
Figure 6.4: The Effectiveness E(k) vs k for scene *lobby*.

Table 6.5: Cumulative Patch Usage Frequency for scene *lobby*.

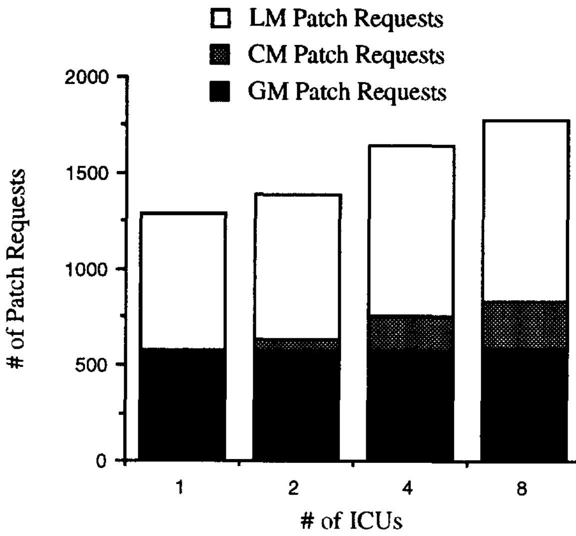
Hi_Low_Ratio	N'	CUF_4	CUF_7	CUF_{10}	CUF_{10}
16	142	0.8934	0.9281	0.913	0.9105
64	97	0.7919	0.836	0.8574	0.7985
256	80	0.7122	0.7367	0.816	0.7367

Table 6.6: Cumulative Patch Usage Frequency for scene *room16.27*.

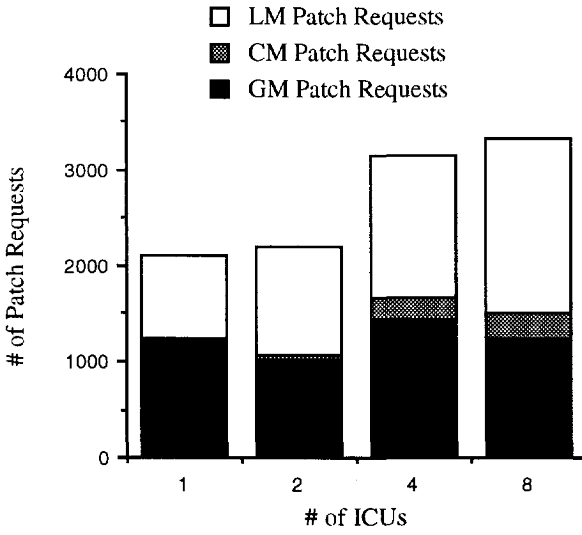
Hi_Low_Ratio	N'	CUF_4	CUF_7	CUF_{10}	CUF_{10}
16	855	0.8549	0.8036	0.8672	0.807
64	325	0.7671	0.7628	0.8163	0.6882
256	190	0.6363	0.713	0.6563	0.5632



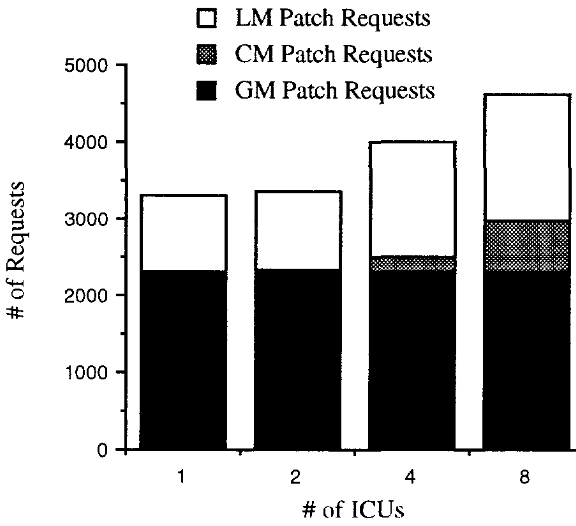
(a) *lobby*



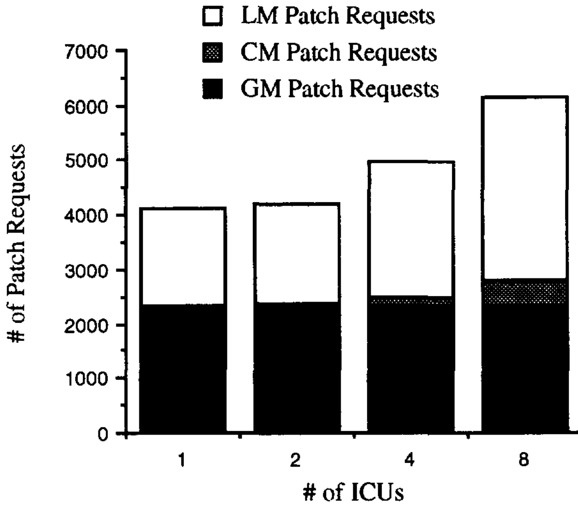
(b) *room16.27*



(c) *array of cubes*

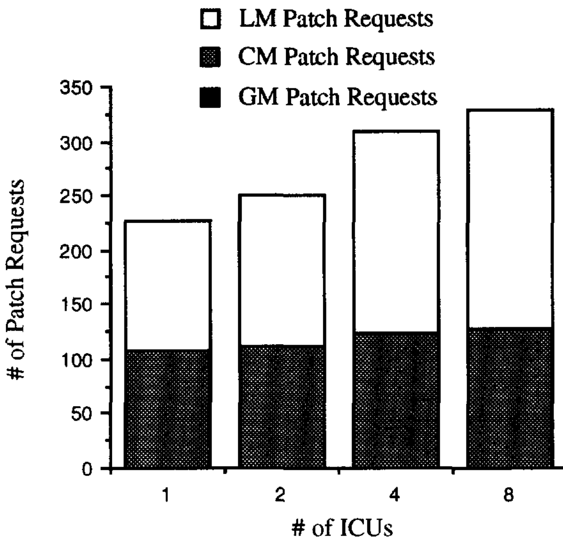


(d) *man*

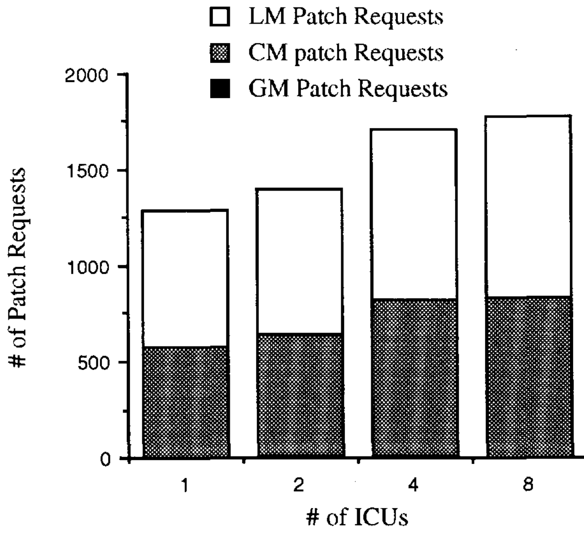


(e) *phone*

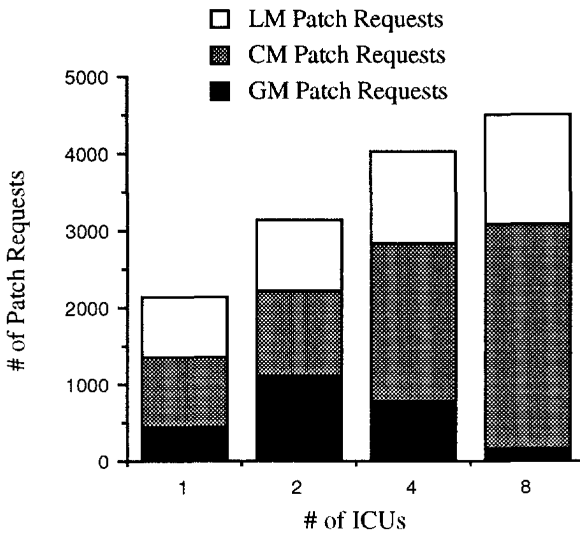
Figure 6.5: Number of patch requests issued to different memories (the reference point).



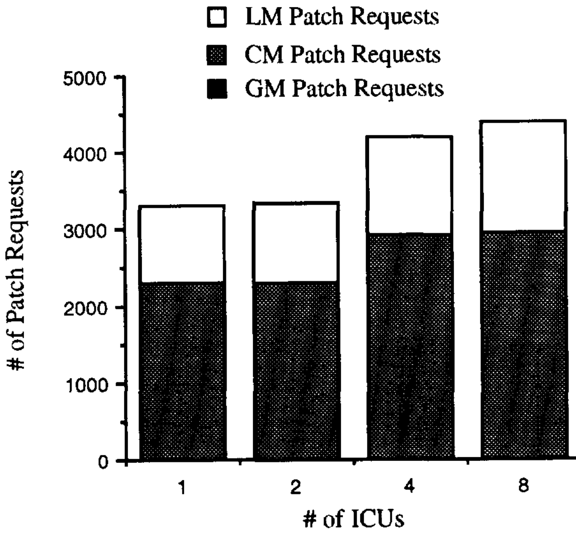
(a) *lobby*



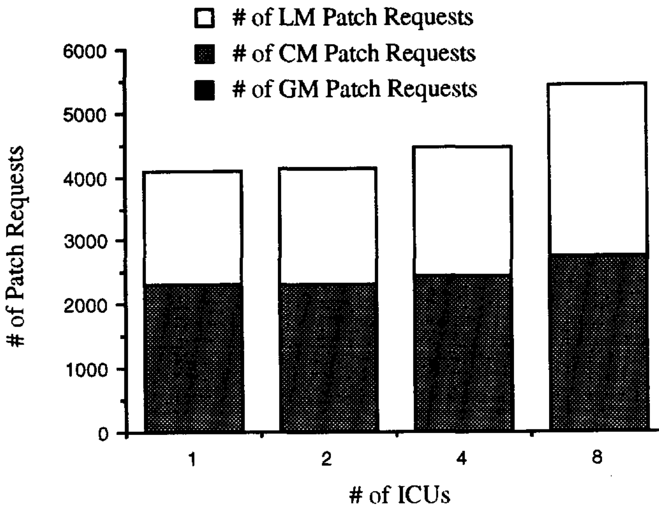
(b) *room16.27*



(c) *array of cubes*



(d) *man*



(e) *phone*

Figure 6.6: Number of patch requests issued to different memories (Sample Point 2).

6.3. System Performance

The system performance is measured in terms of *overall computation time*, *speedup* and *pipeline efficiency*. The test conditions for system performance measurement are listed in Table 6.7.

Table 6.7: Test conditions for system performance measurement.

Scene	# of patches	# of ic rays	cell size
lobby	227	16384, 196608	0.5, 0.0625
room16.27	1297	16384, 196608	0.5, 0.0625
array of cubes	3079	196608	0.0625
man	3315	196608	0.0625
phone	4103	196608	0.0625

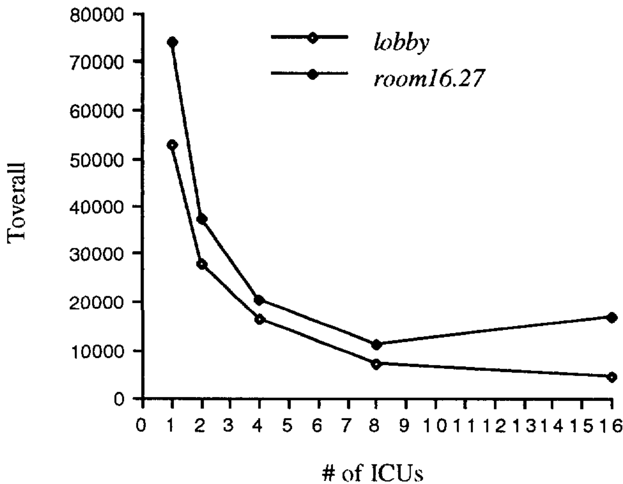


Figure 6.7: Toverall vs # of ICUs (cell size = 0.5; number of rays = 16384).

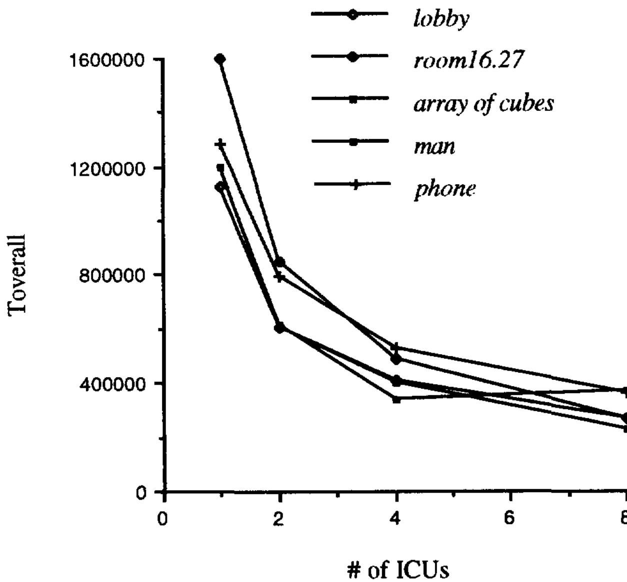


Figure 6.8: Toverall vs # of ICUs (cell size = 0.0625; number of rays = 196608).

6.3.1. Overall Computation Time

The time interval between starting the first computation and finishing the last computation of a problem is called the overall computation time, denoted by $T_{overall}$. Figs. 6.7 and 6.8 show the overall computation times for the five test scenes. Although scene *rooml6.27* has 6 times the number of patches in scene *lobby*, the overall computation time is only 20% - 50% more than that of scene *lobby*. The same situation can be observed from other scenes. This is very promising because the performance of the shelling technique is a weak function of the scene complexity.

6.3.2. Speedup

The speedup is the ratio of the overall computation time of one processor to that with *p* identical processors. It represents, in fact, the number of processors effectively used during the parallel processing of an algorithm. The ideal speedup is never achievable, since some processors are idle at a given time because of conflicts over memory access or communication links, inefficient utilization of concurrency in the underlying hardware. We shall analyze the speedup from the

following three different perspectives.

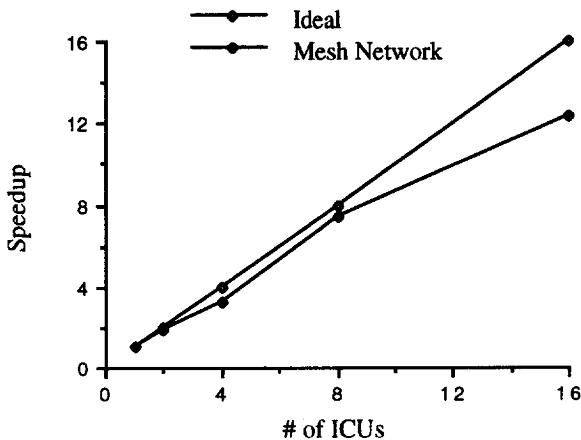
1. First of all, we examine the effects of intercluster communication and local memory latency on the system performance. In order to do this, we choose a large cell size (0.5) so that most patch requests are issued to local clusters directly or to remote clusters via the mesh network. Figs. 6.9a and 6.9b show the speedups of two test scenes *lobby* and *room16.27* when using 1 to 16 ICUs, respectively. As can be seen from the figures, scene *lobby* achieves a reasonable speedup up to 16 ICUs, while there is a fall-off in speedup for scene *room16.27* as the number of ICUs is increased. This fall-off is due to the fact that the latency for local patch requests dominates the useful computation. It is shown in Table 6.9 that the Estimated Degree of Ray Coherence (EDRC[12]) for scene *room16.27* drops to 3 - 5 when using 16 ICUs, or equivalently there are on average 3 - 5 rays tested against a patch. Since each local patch request will take 10 intersection-computation time units, this certainly degrades the system performance. In contrast, the EDRC of scene *lobby* is still larger than 10, that's the reason why it continuously offers a reasonable speedup up to 16 ICUs. In order to guarantee high quality and high resolution rendering, the number of intersection-computation rays required is generally much larger than what has been used for simulation. From this sense, higher scalability of the system can be expected. What is worth noting is that a satisfactory speedup has been observed while preserving a highly effective computation. This can be understood as follows. The total number of intersection computations is $k \times R$ (k is equal to 2.6 and 4.1 for *lobby* and *room16.27*, respectively) as compared to $N \times R$ (N is equal to 227 and 1297 for *lobby* and *room16.27*, respectively) of the naive algorithm, where R is the total number of intersection-computation rays. This represents two-orders-of-magnitude improvement in terms of meaningful computations.

2. As stated previously, patch requests may be issued to global memory, cache memory or local memories. Due to the long latencies for global memory requests and the limited bandwidths provided by the system and local buses (see Fig. 5.1), its effect on the system performance becomes critical. In Figs. 6.10a - 6.10e, the curves labelled Reference Point indicate the speedups for a ray-casting procedure referring to the reference point. There is a clear fall-off in speedup when the number of ICUs is increased. This is mainly due to the following two factors: (1) workloads distributed over ICUs are not balanced, and (2) the long latencies for global memory requests and the limited bandwidths provided by the system and local buses. Fig. 6.11 shows the workloads in terms of the number of intersection computations distributed over ICUs when using 8 ICUs. Since the intersection-computation time still dominates over the communication time of requesting patch data for *lobby* and *room16.27* when using 8 ICUs, a higher scalability of the system can be expected. However, workload imbalancing may degrade the system performance

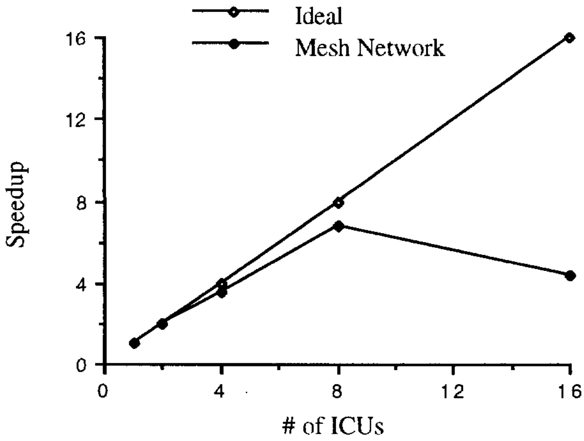
[12] The EDRC represents the average number of rays tested against a patch. It is equal to the total number of intersection computations divided by the total number of patches found in the high-density ray casting.

and it becomes even worse when adding more ICUs. All other scenes indeed suffer from the long latencies for global memory requests and the limited bandwidths provided by the system and local buses when using 8 or more ICUs (see also Fig. 6.5). This explains the fall-off in speedup (Fig. 6.10).

3. At the beginning of a phase, it is indispensable to filling up the cache and local memories through data caching and pre-loading. After data caching and pre-loading, most patch data can be referenced from the cache memory or the local memories that can avoid the long latencies for global memory requests. In Fig. 6.10, the curves labelled Sample Point indicate the speedups for the five test scenes for a ray-casting procedure referring to the Sample Point 2 (see Fig. 6.2). Since the cache memory is faster than the global memory, some improvement in speedup can be achieved. This is explained by the reduction in the number of global memory requests as shown in Fig. 6.6. What is worth noting is that a nearly linear speedup can be obtained for scene *man*. For other scenes, there might be gain or loss in speedup as shown in Fig. 6.10.



(a) *lobby*



(b) *room16.27*

Figure 6.9: The Speedup vs # of ICUs (cell size = 0.5; number of rays = 16384).

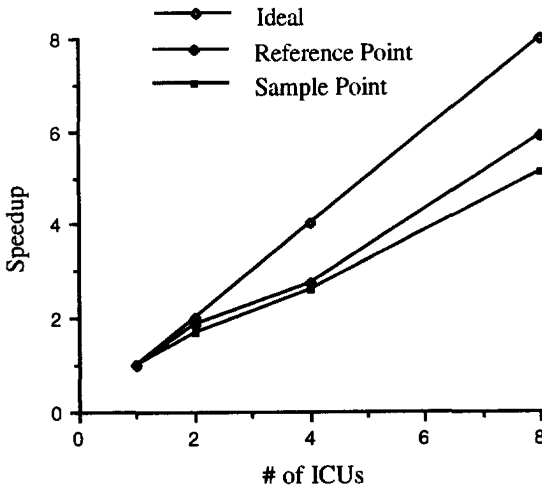
Table 6.8: EDRC for two test scenes (# of ICUs = 8).

Scene	ICU0	ICU1	ICU2	ICU3	ICU4	ICU5	ICU6	ICU7
lobby	125.32	51.7	42.2	27.34	47.69	20.8	135.24	72.94
room16.27	9.32	19.64	20.18	7.93	9.39	22.36	9.72	26.06

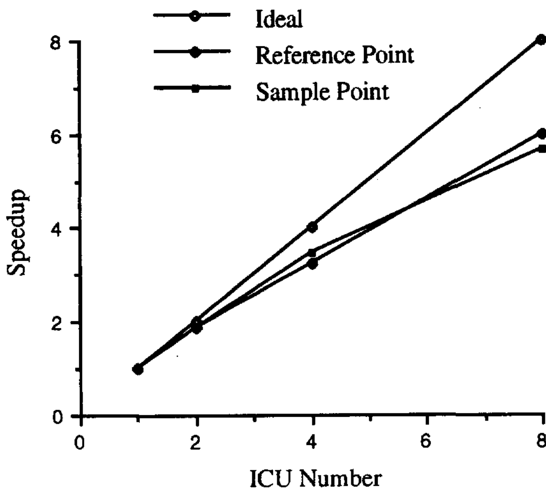
Table 6.9: EDRC for two test scenes (# of ICUs = 16).

Scene	ICU0	ICU1	ICU2	ICU3	ICU4	ICU5	ICU6	ICU7
lobby	85.78	60.16	33.62	59.16	20.59	24.21	29.96	26.83
room16.27	26.46	8.72	7.06	11.13	6.13	12.35	13.9	5.84

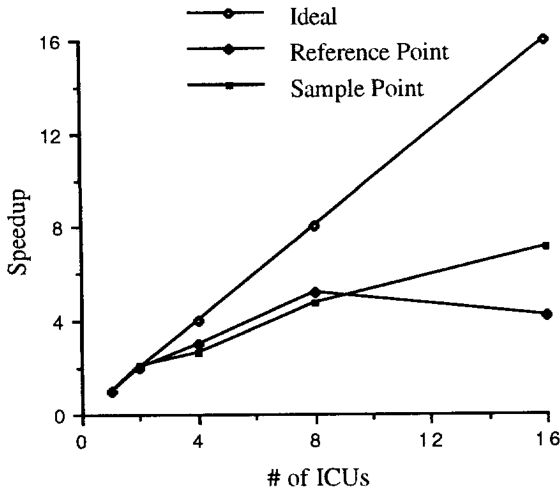
Scene	ICU8	ICU9	ICU10	ICU11	ICU12	ICU13	ICU14	ICU15
lobby	11.69	36.36	72.18	75.12	9.42	20.13	12.5	20.82
room16.27	8.88	14.14	4.7	3.35	3.84	4.97	4.31	5.01



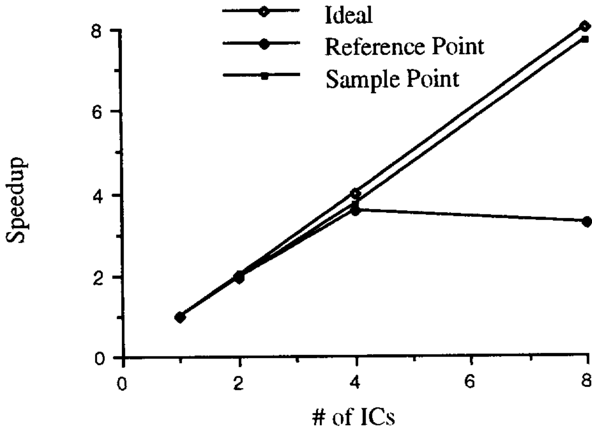
(a) *lobby*



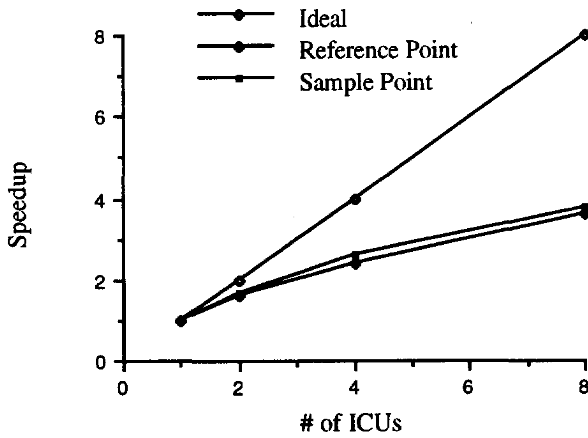
(b) *room16.27*



(c) *array of cubes*

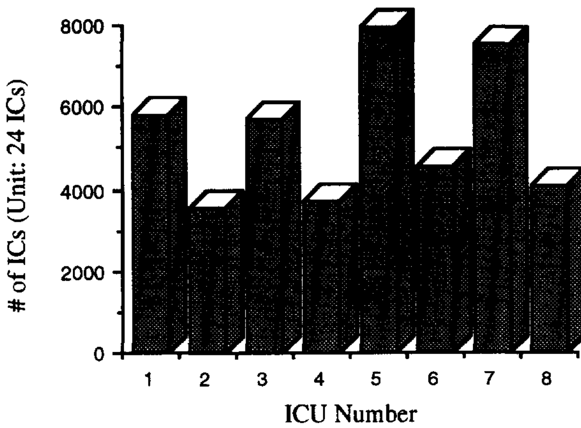


(d) *man*

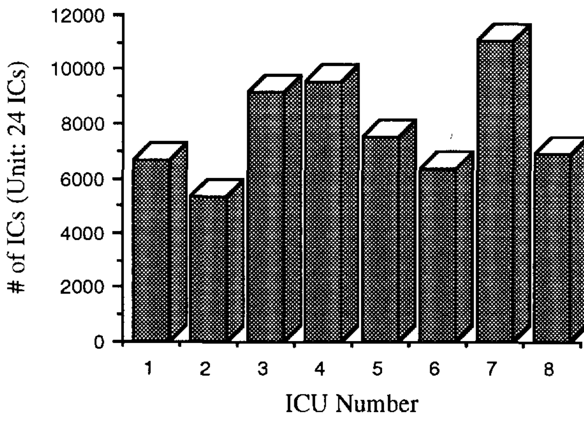


(e) *phone*

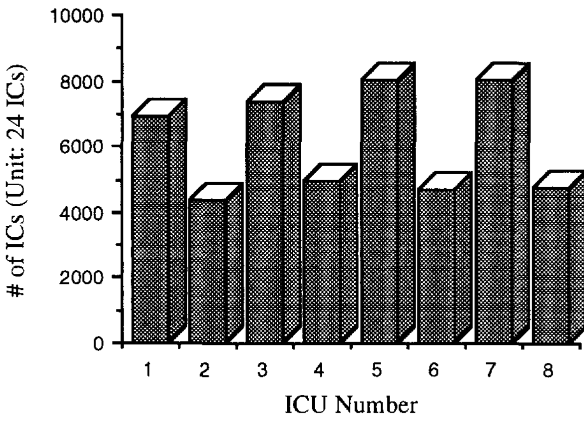
Figure 6.10: The Speedup vs # of ICUs (cell size = 0.0625; number of rays = 196608).



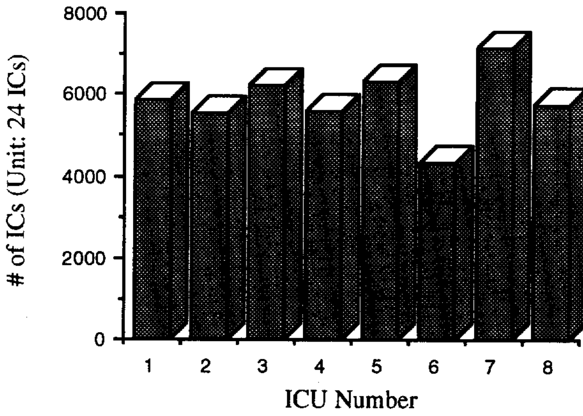
(a) *lobby*



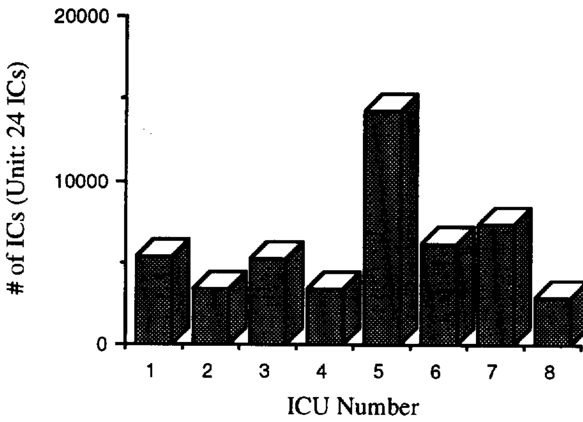
(b) *room16.27*



(c) *array of cubes*



(d) *man*

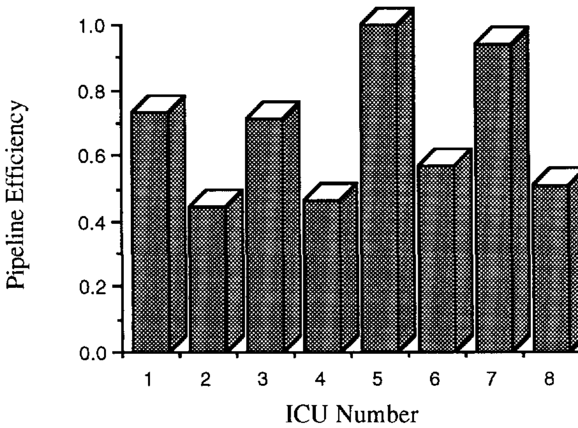


(e) *phone*

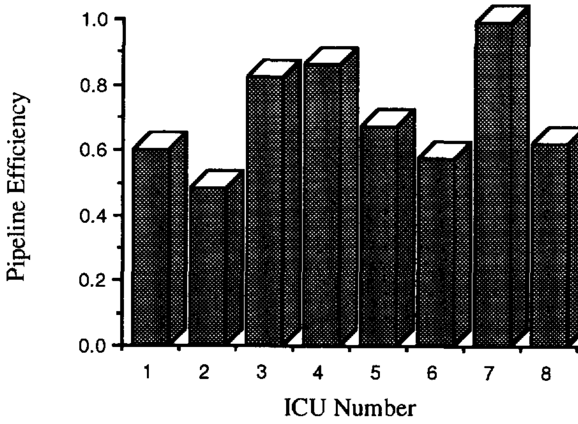
Figure 6.11: Workloads distributed over ICUs for a ray-casting procedure referring to the reference point (cell size = 0.0625; number of rays = 196608).

6.3.3. Pipeline Efficiency

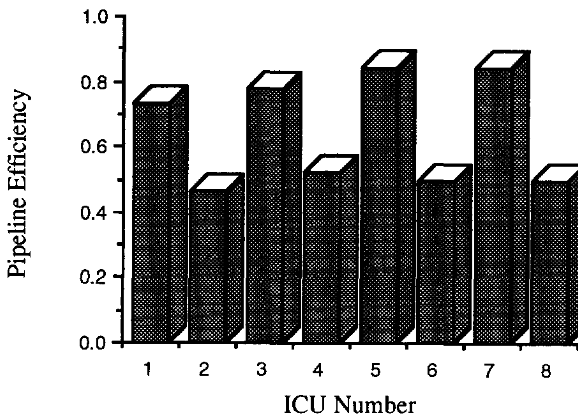
As discussed in chapter 5, a highly pipelined ICU plays an important role in the radiosity engine. We use pipeline efficiency which is defined by the percentage of busy time-space span over the total time-space span (i.e., the sum of all busy and idle time-space spans) to measure the efficiency of an ICU. Figs. 6.12a - 6.12e show the pipeline efficiencies of a ray-casting procedure referring to the reference point for the five test scenes. In fact, they are just another interpretation of those figures showing workloads distributed over ICUs (Figs. 6.11a - 6.11e). Although scene *man* is the best in terms of workload balancing, the pipeline efficiency is only 0.4 when using 8 ICUs. This is due to the long latencies for global memory requests, which is supposedly to be solved by caching most frequently referenced patch data during a phase. Fig. 6.13d shows the improvement in the pipeline efficiency (0.8) for a ray-casting procedure referring to Sample Point 2. In contrast, scene *phone* has the worst workload balancing, the low pipeline efficiency is mainly due to the heavily loaded ICU (i.e., ICU Number 5). For other scenes, the gain or loss in speedup consistently reflects on the pipeline efficiency as shown in Fig. 6.13.



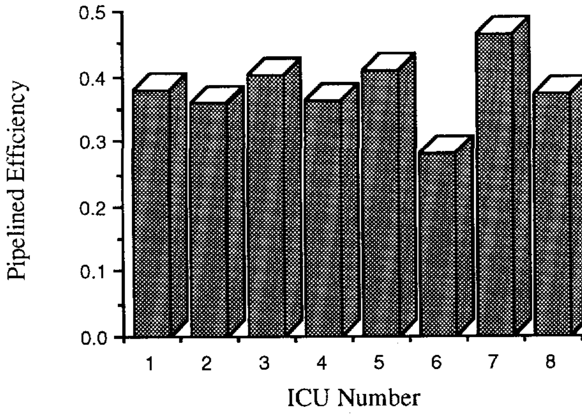
(a) *lobby*



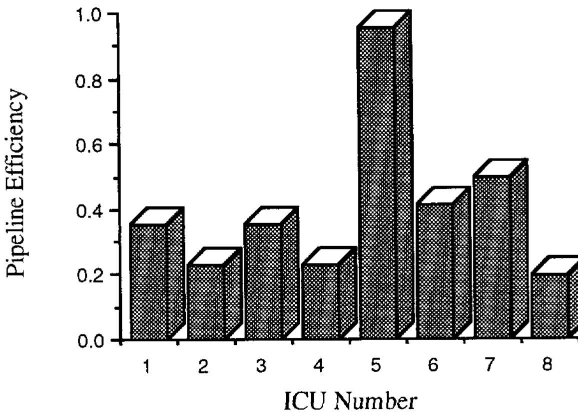
(b) *room16.27*



(c) *array of cubes*

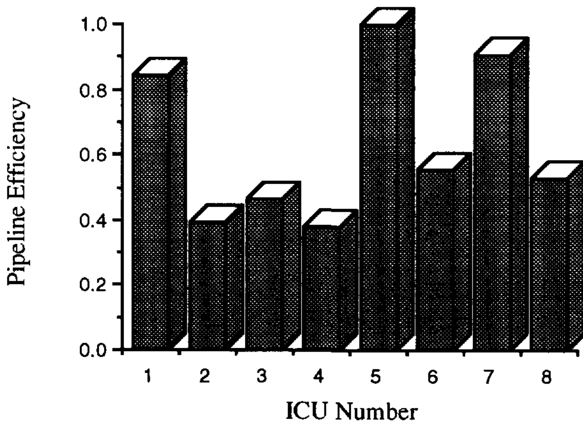


(d) *man*

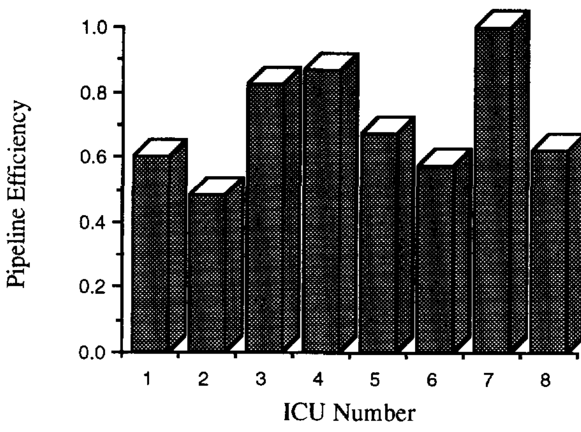


(e) *phone*

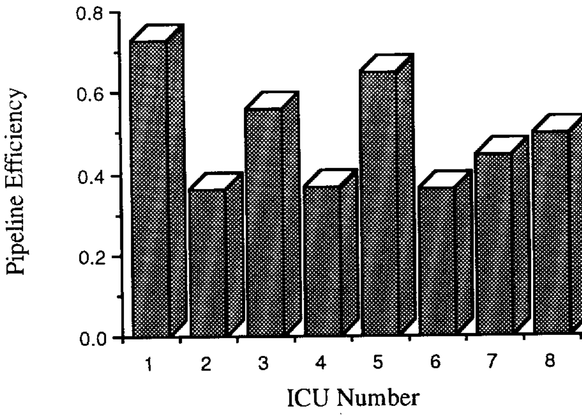
Figure 6.12: The Pipeline Efficiency for a ray-casting procedure referring to the reference point (cell size = 0.0625; number of rays = 196608).



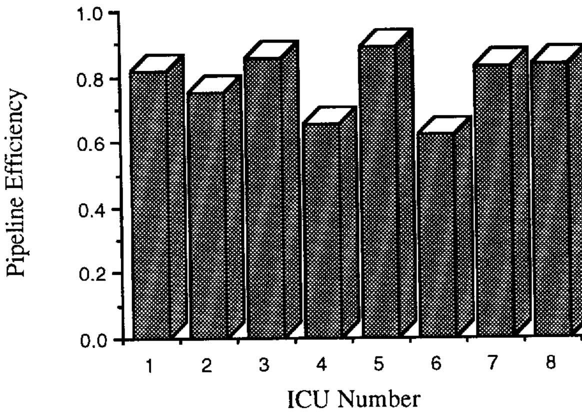
(a) *lobby*



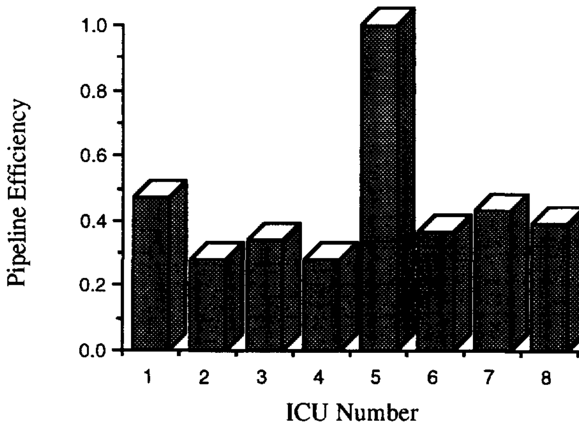
(b) *room16.27*



(c) *array of cubes*

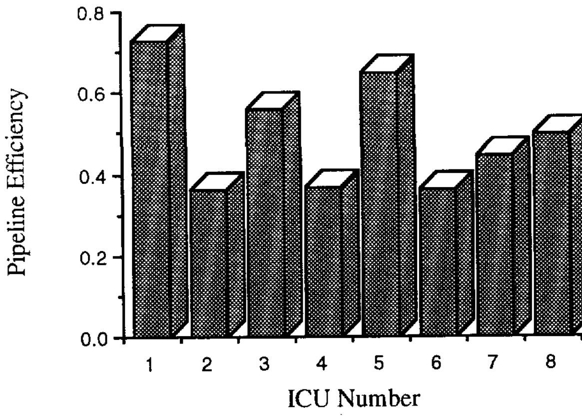


(d) *man*

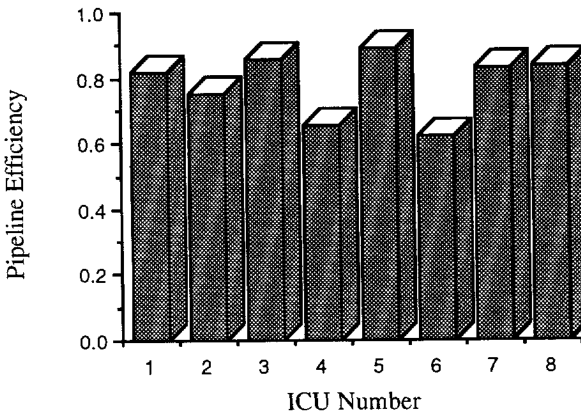


(e) *phone*

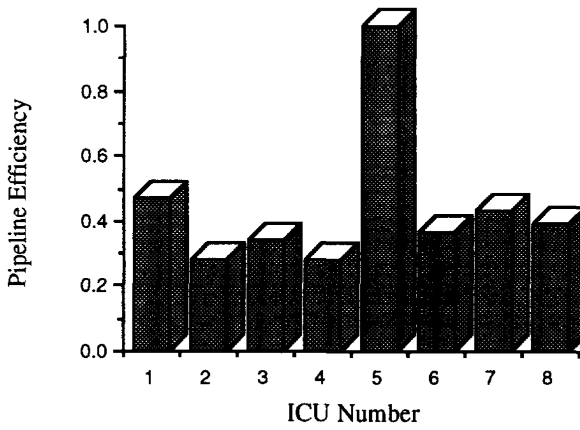
Figure 6.13: The Pipeline Efficiency for a ray-casting procedure referring to Sample Point 2 (cell size = 0.0625; number of rays = 196608).



(c) *array of cubes*



(d) *man*



(e) *phone*

Figure 6.13: The Pipeline Efficiency for a ray-casting procedure referring to Sample Point 2 (cell size = 0.0625; number of rays = 196608).

Chapter 7

Concluding Remarks

In this dissertation, we have explored ways to improving the performance of a ray-casting based approach for visualizing artificial scenes with photo realism on the screen of a workstation. The approach is simulation and the objective is high-performance and high-speed computation. One could hardly come up with something more demanding, except perhaps for an additional walk-through option. Surely, virtual reality comes to mind as being an order of magnitude more complex, but this application is still in its infancy, in the stage of brute-force super-computing. And yet, computer graphics plays a role and will play a role in the world of virtual reality, in the world of multi-media and in fact in all applications where visualization and simulation come into play. It is all a matter of large-scale computing, such is clear. The question is what the computing platform should be to achieve true high-performance and high-speed computation. Possible solutions are super-computing, distributed computing, multi-processor computings - general purpose and special purpose - and dedicated computing. A characteristic feature common to all those large-scale computing is that they come with massively parallel algorithms which seem to suggest naturally massively parallel implementations. However, massive parallelism in hardware is an illusion, except in some rare cases, where lots of trivial operations with local interaction can be smeared out of large silicon areas to achieve very high throughput and low-power implementations. The ray-tracing/radiosity rendering problem that we have tackled in this dissertation is a beautiful example of simulation technique that leads to massive parallelism in the architecture. High performance and high speed have mainly be attained through combined algorithm and architecture design in which a parameterized space partitioning on the one hand has found its counterpart in a scalable network of clusters on the other hand. The result is not a special purpose graphics computer but a semi-dedicated graphics

co-processor which is attached to a standard workstation. The complete algorithm runs on both the host and its graphics mate in a true shared/distributed manner. This mate in turn is partly programmable and partly dedicated to speed up the execution of the tremendous amount of primitive operations. To summarize, we have focused on an application-specific solution that allows us to fine tune an algorithm-architecture pair. This recognition is very helpful for the practice of system design because a general-purpose solution is not conceivable.

The overhead of latency and synchronization for a parallel machine can be mitigated by exploiting data coherence at various levels. At the algorithmic level, two solutions have been proposed as follows:

1. In chapter 2, we presented a proximity enforced algorithm based on the following strategy: (1) First of all, we should store most referenced data in fast memory, while keeping less referenced data in slow memory; (2) Secondly, the execution of the program (i.e., the ray casing based approach) should be enforced so that the data stored in fast memory allow to be referenced as often as needed.
2. In chapter 3, we proposed a new space partitioning technique called the shelling technique. By relying on ray-frustum casting, the long latencies of retrieving patch data can be amortized over many useful computations.

To support the above algorithmic demands, some architecture requirements have been necessary.

1. In chapter 5, a memory structure which is divided into working register, resident set, cache memory, and global memory has been described. This allows to store most referenced data in fast memory such as working register or resident set, while keeping less referenced data in slow memory such as cache and global memory.
2. To support the ray-frustum casting, we have assumed a highly pipelined Intersection Computation Unit. Through pipelining, communication with the external world only occurs at the start and end of the pipeline.

In chapter 4, we have presented an *application-specific runtime scheduling* (*ASRS*) for mapping the *shelling technique* onto a *pipelined parallel architecture*. Although *ASRS* was shown to be sound and useful, it is limited to this specific application. In fact, many applications exhibit dynamic and data-dependent runtime behavior, including sparse matrix problems, adaptive PDE solvers, particle simulations and many combinatorial optimization problems, etc. This work laid the foundation of a compiler-directed runtime resource management. To broaden the domain of applications, we should combine compilation techniques and runtime system techniques to handle resource management automatically. By analyzing the static syntax of an application program (e.g., a ray-casting procedure), the compiler creates special runtime routines (e.g., low-density ray casting) to gather program-specific runtime information. This runtime information is made available to the runtime resource

management system during program execution. The runtime resource management system also keeps track of the availability of resources. With all this information, the runtime resource management system manages available resources and schedules subsequent concurrent execution (e.g., clustering, assignment and local scheduling).

The radiosity engine has been described in chapter 5, including system configuration, memory design, network design, synchronizaton mechanism and its functionality. It has been recognized as a pipelined parallel architecture featuring the following:

1. It is a parallel machine made of a mesh-connected network of processing nodes (see Fig. 5.1). Each processing node consists of some interface chips, memory chips, possibly some general-purpose chips, and dedicated VLSI chips. All processing nodes work on assigned computations simultaneously and asynchronously.
2. It is a ray-frustum based machine that achieves highly efficient and effective parallel computing. Possible applications include: (1) radiosity preprocessing for projective (depth-buffer) display, (2) ray tracing with a radiosity preprocessing and (3) ray tracing without a radiosity preprocessing. In this dissertation, we have compared ray-frustum casting with single-ray casting. Let us to use this opportunity to also compare ray-frustum casting with standard projective algorithms (depth-buffer, scan-line, etc.). First of all, the SBB computation is an alternative to the screen-space projection of a patch. The former is more flexible because patches are not restricted to polygons as in the latter case. Secondly, intersection computation in ray-frustum casting can be accelerated by exploiting the coherence property [Hek93*b*] as in scan-conversion algorithms [SSS74]. The radiosity engine is ray-tracing based, while preserving most desirable features in standard projective algorithms, and so lend itself to *fast* and *realistic* image synthesis.

The radiosity engine has been completely modelled in BONeS®, and we have evaluated its performance for a set of practical scenes. Promising results have been observed, including the following:

1. The performance of the shelling technique is a weak function of the scene complexity. The computational complexity of the shelling technique is $k \times R$ (k is about 2 - 5) as compared to $N \times R$ (N is the total number of patches) of the naive algorithm, where R is the total number of intersection-computation rays.
2. A reasonable speedup has been observed up to 8 clusters. The limiting factors in speedup are workload imbalancing, the long latencies for global memory requests and the limited bandwidths supported by the system and local buses. To achieve a higher scalability of the system, further improvement in the front-end system together with the use of a dynamic workload balancing scheme would be necessary.
3. The performance of software intersection computation on HP720 is about 0.2M/sec. The radiosity engine provides two-orders-of-magnitude more processing power than this software approach per cluster.

Further investigation is required for the following issues:

1. Ray-frustum casting is mainly intended for *undirect* shooting. As pointed out in [JKV92] [WEH89], it is possible that a small patch when viewed from a sample point will not receive any contribution. Therefore, it may look black even it is viewed from a close distance. This artifact due to undersampling may be solved by casting ultrahigh-resolution rays together with jittering the ray directions. Further investigation is required in order to tune the ray-density as per patch or ray frustum.

2. In the ray-tracing pass, we proposed a patch-based ordering for shooting. By following the scan-line ordering, intersection points pertaining to the same patch can be grouped together for shadow ray casting and/or reflection/transmission ray casting. In this way, the relevant data required by an intersection point (to start with a shadow ray casting and/or a reflection/transmission ray casting) can be stored in fast memory, and most of them will be reused by other intersection points in the same group due to the data-coherence property. In general, the amount of data coherence for the primary rays will be large, but it will drop quickly for a higher-generation rays. From this point of view, it is better to take a breadth-first ordering in the ray tree instead of a depth-first. We could first shoot all the primary rays, then the secondary rays, and so on. However, this may require a large memory to store intermediate results. The question is how to exploit the data coherence to a maximum extent as long as intermediate results will not swamp the entire memory.

We have travelled the whole trajectory from algorithm, architecture and down to the electronic design. We are now developing a 4-cluster prototype system in corporation with TPD-TNO at Delft. Meanwhile, we are developing VLSI chips for some critical blocks such as *Cell Traversal* and *Intersection Computation*. Finally, we conclude that the radiosity engine is a very efficient and effective parallel machine that can serve as a rendering accelerator attached to various standard workstations, in particular for ray-casting based applications.

Bibliography

[ABM88] B. Apgar, B. Bersack and A. Mammen, A Display System for the Stellar Graphics Supercomputer Model GS1000, *ACM Computer Graphics*, **22**, 4, pages 255-262, August 1988.

[AD79] W.B. Ackerman and J.B. Dennis, VAL - A Value-Oriented Algorithmic Language, *Preliminary Reference Manual*, Laboratory for Computer ScienceTechnical Report 218, MIT, Cambridge, Mass., June 1979.

[AI87] Arvind and R.A. Iannucci, Two Fundamental Issues in Multiprocessing, *DFVLR - Conf. 1987 on Parallel Processing in Science and Engineering*, June 1987.

[AJ88] K. Akley and T. Jermoluk, High-Performance Polygon Rendering, *ACM Computer Graphics*, **22**, 4, pages 239-246, August 1988.

[Akl89] K. Akley, The Silicon Graphics 4D/240GTX Superworkstation, *IEEE Computers and Applications*, **9**, 4, pages 71-83, July 1989.

[Amd67] G. Amdahl, The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, *AFIPS Conf. Proc.*, **30**, 1967.

[App68] A. Appel, Some Techniques for Shading Machine Rending of Solids, *Proc. AFIPS JSCC*, **23**, 6, pages 37-45, 1968.

[Arv86] J. Arvo, Backward Ray Tracing, Developments in Ray Tracing *(SIGGRAPH'86 Cource Note)*, 1986.

[AT80] Arvind and R.E. Thomas, I-Structures: An Efficient Data Type for Functional Languages, Laboratory for Computer Science, MIT, September 1980.

[Bok81] S.H. Bokhri, On the Mapping Problem, *IEEE Trans. Computers*, **30**, 3, pages 207-214, 1981.

[CCWG88] M. Cohen, S. Chen, R. Wallace, and D. Greenberg, A Progressive Refinement Approach to Fast Radiosity Image Generation, *Computer Graphics (SIGGRAPH '88 Proceedings)*, **22**, 4, pages 75-84, August 1988.

[CF90] A.T. Campbell and D.S. Fussell, Adaptive Mesh Generation for Global Diffuse Illumination Model, *Computer Graphics (SIGGRAPH '90 Proceedings)*, **24**, 4, pages 155-164, 1990.

[CG85] M. Cohen and D. Greenberg, The Hemi-Cube: A Radiosity Solution for Complax Environments, *Computer Graphics (SIGGRAPH '85 Proceedings)*, **19**, 3, pages 31-40, July 1985.

[CGIB86] M. Cohen, D. Greenberg, D. Immel, and P. Brock, An Efficient Radiosity Approach for Realistic Image Synthesis, *IEEE Computer Graphics and Applications*, **6**, 2, pages 26-35, March 1986.

[CH92] M. Cox and P. Hanrahan, Depth Complexity in Object-Parallel Graphics Architectures, *7th Workshop on Graphics Hardware*, **4**, 4, 1992.

[Cla76] J.H. Clark, Hierarchical Geometric Models for Visible Surface Algorithms, *Comm. ACM*, **19**, 4, April 1976.

[CPC84] R.L. Cook, T. Porter and L. Carpenter, Distributed Ray Tracing, *Computer Graphics*, **18**, 3, pages 137-143, July 1984.

[Dal87] W.J. Dally, Wire-Efficient VLSI Multiprocessor Communication Networks, In *Proc. Standford Conf. on Advanced Research in VLSI*, pages 391-415, March 1987.

[DHM88] T. Diede, C.F. Hagenmaier, G.S. Miranker and *el.*, The Titan Graphics Supercomputer Architecture, *IEEE Computer*, pages 13-30, September 1988.

[DHW93] Ed. F. Deprettere, P. Held and P. Wielage, Model and Methods for Regular Array Design. *Int. J. of High Speed Electronics*, **4**, 4, 1993.

[FP92] M. Feda and W. Purgathofer, Accelating Radiosity by Overshooting, In *Proc. of the 3rd Eurographics Rendering Workshop*, page 233-245, 1992.

[FPE89] H. Fuchs, J. Poulton, J. Eyles and *el.*, Pixel Planes 5: A Heterogeneous MultiprocessorGraphics System Using Processor Enhanced Memories, *Computer Graphics (SIGGRAPH '89 Proceedings)*, **23**, 3, pages 79-88, July 1989.

[FTI86] A. Fujimoto, T. Tanaka, and K. Iwata, ARTS: Accelerated Ray-Tracing System, *IEEE Computer Graphics and Applications*, **6**, 4, pages 16-26, April 1986.

[Fuc85] H. Fuchs *et al.*, Fast Spheres, Shadows, Textures, Transparencies and Image Enhancements in Pixel-Planes, *Computer Graphics (SIGGRAPH '85 Proceedings)*, **19**, 3, pages 111-120, July 1985.

[GH71] Gouraud and Henri, Computer Display of Curved Surfaces, Ph.D. Thesis, University of Utah, 1971.

[GHS92] G. Greiner, W. Heidrich and P. Slusallek, Blockwise Refinement - A New Method for Solving the Radiosity Problem, In *Proc. of the 4th Eurographics Rendering Workshop*, 1993.

[GJ79] M.R. Garey and D.S. Johnson, Computers and Intractability: a Guide to the Theory of NP-Completeness, Freeman, San Francisco, 1979.

[Gla84] A. Glassner. Space Subdivision for Fast Ray Tracing. *IEEE Computer Graphics and Applications*, Vol. 4, No. 10, pages 15-22, October 1984.

[GP89] S.A. Green and D.J. Paddon, Exploiting Coherence for Multiprocessor Ray Tracing, *IEEE Computer Graphics & Applications*, pages 12-27, November 1989.

[GTGB84] C. Goral, K. Torrance, D. Greenberg, and B. Battaile, "Modeling the Interaction of Light Between Diffuse Surfaces, *Computer Graphics (SIGGRAPH '84 Proceedings)*, **18**, 3, pages 212-222, July 1984.

[HB84] K. Hwang and F.A> Briggs, Computer Achitecture and Parallel Processing, McGraw-Hill, Inc., 1984.

[Hec90] P.S. Heckbert, Adaptive Radiosity Textures for Bidirectional Ray Tracing, *Computer Graphics (SIGGRAPH '90Proceedings)*, **24**, 4, pages 145-154, 1990.

[Hek93*a*] G. Hekstra, Intersection Computation using Bounding Plane Subdivision, Technical Report ET/N/Radio-007, Delft University of Technology, June 1993.

[Hek93*b*] G. Hekstra, Definition and Structure of Hemisphere and Viewing Rays, Technical Report ET/N/Radio-009, Delft University of Technology, September 1993.

[Hek93*c*] G. Hekstra, Derivation of a Suitable 2-dimensional φ, θ, Sampling Grid for Fast Cell Traversal, Technical Report ET/N/Radio-015, Delft University of Technology, September 1993.

[HMS86] J.P. Hayes, T. Mudge, Q.F. Scott and *al.*, . A Microprocessor-based Hypercube Supercomputer, *IEEE Micro*, *6*, 5, pages 6-17, October 1986.

[HSA91] P. Hanrahan, D. Salzman and L. Aupperle, A Rapid Hierarchical Radiosity Algorithm, *Computer Graphics (SIGGRAPH '91Proceedings)*, **25**, 4, pages 197-206, August 1991.

[HW91] E.A. Haines and J.R. Wallace, Shaft Culling for Efficient Ray-Traced Radiosity, *2nd Eurographics Workshop on Rendering*, May 1991.

[ICG86] D. Immel, M. Cohen, and D. Greenberg, A Radiosity Method for Non-diffuse Environments, *Computer Graphics (SIGGRAPH '86 Proceedings)*, **20**, 4, pages 133-142, August 1986.

[ISC84] B. Indurkhya, H.S. Stone and L. Xi-Cheng, Optimal partitioning of randomly generated distributed programs, *IEEE Trans. Software Eng.*, **12**, 3, pages 483-495, 1986.

[JKV92] F.W. Jansen, A.J.F. Kok and T. Verelst, Hardware Challenge for Ray Tracing and Radiosity Algorithms, *7th Workshop on Graphics Hardware*, pages 123-134, September 1992.

[Kaj86] J.T. Kajiya, The Rencering Equation, *Computer Graphics (SIGGRAPH '86 Proceedings)*, **20**, 4, pages 143-150, 1986.

[KCN90*a*] C.T. King, W.H. Chou and L.M. Ni, Pipelined Data-Parallel Algorithms: Part I-Concept and Modeling, *IEEE Trans. Parallel and Distributed Systems*, **1**, 4, pages 470-485, oct. 1990.

[KCN90*b*] C.T. King, W.H. Chou and L.M. Ni, Pipelined Data-Parallel Algorithms: Part II-Design, *IEEE Trans. Parallel and Distributed Systems*, **1**, 4, pages 486-499, oct. 1990.

[KJ91] A.J.F Kok and F.W. Jansen, Source Selection for the Direct Lighting Computation in Global Illumination, In *Proc. of the 2nd Eurographics Rendering Workshop*, 1991.

[KJ92] A.J.F Kok and F.W. Jansen, Adaptive Sampling of Area Light Sources in Ray Tracing Including Diffuse Interreflection, *Proceedings Eurographics'92*, 1992.

[KJW93] A.J.F. Kok, F.W. Jansen and C. Woodward, Efficient, Complete Radiosity Ray Tracing using a Shadow-Coherence Method, *Visulal Computer*, 1993.

[KM85] Kaplan and R. Michael. Space Tracing: A Constant Time Ray Tracer. *ACM SiGGRAPH '85 Course Notes 11*, pages 22-26, July1985.

[KN84] H. Kasahara and S. Narita, Practical Multiprocessor Schedling Algorithms for Efficient Parallel Processing, *IEEE Trans. Computers*, **33**, 11, pages 1023-1029, 1984.

[Kru88] B. Kruatrachue and T. Lewis, Grain Size Determination for Parallel Processing, *IEEE Software*, pages 23-32, Jan. 1988.

[KV90] D. Kirk and D. Voorhies, The rendering Architecture of the DN10000VS, *ACM Computer Graphics*, **24**, 4, pages 299-307, August 1990.

[Lo88] V.M. Lo, Heuristic Algorithms for Task Assignment in Distributed Systems, *IEEE Trans. Computers*, **37**, 11, pages 1384-1397, 1988.

[McG82] J.R. McGraw, The VAL Lanuage: Description and Analysis, *ACM Transaction on Programming Languages and Systems*, **4**, 1, pages 45-82, Jan. 1982.

[McL88] J. McLeod, HP Delivers Photo Realism on an Interactive System, *Electronics*, pages 95-97, March 1988.

[McW92] J. McWhirter, A Worked Example in Algorithmic Engineering, In *Proc. SPIE Advanced Signal Processing Algorithms, Architectures, and Implementations III*, pages 20-30, March 1992.

[MHI88] K. Murakami, K. Hirota, and M. Ishii, Fast Ray Tracing, *FUJITSU Sci. Technical Journal*, Vol. 24, No. 2, pages 150-159, June 1988.

[MHS86] K. Murakami, K. Hirota, and H. Sato. Ray Tracing System Using The Voxel Partitioning on a Cellular Array Processor, *Graphics and CAD*, **22**, 2, pages 1537-1538, 1986.

[MM83] H. Matsumoto and K. Murakami, Fast Ray Tracing Using The Octree Partitioning, In *Proc. 27th Inform. Proc. Conf*, pages 15-22, 1983.

[MO86] R. Morison and S. Otto, The Scattered Decomposition for Finite Elements, *J. Sci. Comput.*, **2**, 1986.

[MVD92] M. Moonen, P. Vanpoucke and Ed F. Deprettere, Parallel and Adaptive Gigh Resolution Direction Finding, In *Proc. SPIE Advanced Signal Processing Algorithms, Architectures, and Implementations III*, pages 219-230, March 1992.

[Nik87*a*] R.S. Nikhil, Id Nouveau Reference Manual, Part I: Syntax, Technical Report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, ambridge, MA 02139, April 1987.

[Nik87*b*] R.S. Nikhil, Id World Reference Manual, Technical Report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, ambridge, MA 02139, April 1987.

[NOK83] H. Nishimura, H. Ohno, T. Kawata and *el.*, Links-1: A Parallel Pipelined Muiticomputer System for Image Creation, In *Proc. 10th Symp. Computr Architecture*, pages 387-394, 1983.

[Ost87] A. Osterhaug, Guide to Parallel Programming on Sequent Computer Systems, Sequent Computer Systems, Inc., 1987.

[RW80] S.M. Rubin and T. Whitted, A 3-Dimensional Representation for Fast Rendering of Complex Scenes, *Computer Graphics (SIGGRAPH '80 Proceedings)*, **14**, 3, pages 110-116, 1980.

[Sar87] V. Sarkar, Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors, Technical Report CSL-TR-87-328, Stanford University Computer Systems Laboratory, April, 1987.

[SC78] E.M. Sparrow and R.D. Cess, Radiation Heat Transfer, Hemisphere Publishing Corp., 1978.

[SD92] L.S. Shen, and Ed. F. Deprettere: A Hierarchical Memory Structure for the 3D Shelling Technique. Sixth Annual European Computer Conference, 1992

[SDP90] L.S. Shen, Ed. F. Deprettere, and P. Dewilde, A New Space Partitioning Technique to Support a Highly Pipelined Parallel Architecture for the Radiosity Method, *Fifth Eurographics Workshop on Graphics Hardware*, 1990.

[SDP91*a*] L.S. Shen, Ed. F. Deprettere, and P. Dewilde, An Intermediate Data Structure to Support a Highly Pipelined Parallel Architecture for the Radiosity Method," in *Algorithms and Parallel VLSI Architectures*, North-Holland, 1991.

[SDP91*b*] L.S. Shen, Ed. F. Deprettere, and P. Dewilde, A New Space Partitioning for Mapping Computations of the Radiosity Method onto a Highly Pipelined Parallel Architecture, *ProRISC Symposium*, 1991.

[SD92] L.S. Shen, and E. Deprettere, A Parallel-Pipelined Multiprocessor System for the Radiosity Method, *Seventh Eurographics Workshop on Graphics Hardware*, 1992.

[Sei85] C.L. Seitz, The Cosmic Cube, *Communications of the ACM*, **28**, 1, pages 22-33, January 1985.

[SE87] P. Sadayappan and F. Ercal, Nearest-neighbor Mapping of the Finite Element Graphs onto Processor Meshes, *IEEE Trans. Computers*, **36**, 12, pages 108-1424, 1987.

[SE90] P. Sadayappan and F. Ercal and J. Ramanujam, Cluster Partitioning Approaches to Mapping Parallel Programs onto a Hypercube, Parallel Computing, 13, North-Holland, pages 1-16, 1990.

[Shi90] P. Shirley, A Ray Tracing Method for Illumination Calculation in Diffuse Specular Scenes, *Proc. Graph Interface*, pages 205-211, 1990.

[SLD91] L.S. Shen, F.A.J. Laarakker, and E. Deprettere, A New Space Partitioning for Mapping Computations of the Radiosity onto a Highly Pipelined Parallel Architecture (II). *Sixth Eurographics Workshop on Graphics Hardware*, 1991.

[SP89] F. Sillion and C. Puech, A General Two Pass Method Integrating Specular and Diffuse Reflection, *Computer Graphics (SIGGRAPH '89 Proceedings)*, **23**, 3, pages 335-344, 1989.

[SRHJ78] Siegl, Robert, Howell and R. John, Thermal Radiation Heat Transfer, Hemisphere Publishing Corp., 1978.

[SSS74] I.E. Sutherland, R.F. Sproull and R.A. Schumacker, A Characterization of Ten Hidden-Surface Algorithms, *ComputingSurveys*, **6**, 1, March 1974.

[ST85] C. Shen and W. Tsai, A Graph Matching Approach to Optimal Task Assignment in Distributed Computin Systems using a Minmax Criterion, *IEEE Trans. Computers*, **34**, 3, pages 197-203, 1985.

[TKM84] M. Tamminen, O. Karonen and M. Mantyla, Ray-Casting and Block Model Conversion Using Spatial Index, *Computer-Aided Design*, **16**, 4, pages 203-208, July 1984.

[TMC87] Connection Machine Model CM-2 Technical Summary, Technical Report HA87-4, Thinking Machine Corp., April 1987.

[Tor87] J. Torborg, A Parallel Processor for Graphics Arithmetic Operations, *ACM Computer Graphics*, **21**, 3, pages 197-204, July 1987.

[WCG87] J. Wallace, M. Cohen, and D. Greenberg, A Two Pass Solution to the Rendering Equation: A Synthesis of Ray-Tracing and Radiosity Methods, *Computer Graphics (SIGGRAPH '87 Proceedings)*, **21**, 4, pages 311-320, July 1987.

[WEH89] J.R. Wallace, K.A. Elmquist and E.A. Haines, A Ray Tracing Algorithm for Progressive Radiosity, *Computer Graphics (SIGGRAPH '889Proceedings)*, **23**, 2, pages 315-324, 1989.

[WHG84] H. Weghorst, G. Hooper and D.P. Greenberg, Improved Computational Methods for Ray Tracing, *ACM Tran. Graphics*, **3**, 1, pages 52-69, 1984.

[Whi80] T. Whitted, An Improved Illumination Model for Shaded Display, *Communications of the ACM*, **23**, 6, pages 343-349, June 1980.

[WJC88] Van Wyk, Jr, and G. Christopher, A Geometry-based Insolation Model for Computer-Aided Design, Ph.D Thesis, The University of Michigan, 1988.

[WRC88] G.J. Ward, F.M. Rubinstein and R.D. Clear, A Ray Tracing Solution for Diffuse Interreflection, *Computer Graphics (SIGGRAPH '88 Proceedings)*, **22**, 4, pages 85-92, 1988.

Acknowledgements

This research has been supported in part by the commission of the EEC under the ESPRIT program, project BRA 3280 (NANA), the Dutch Technology Foundation under contract DEL 99.1982 and by ITRI (a research organization in Taiwan).

I wish to express my gratitude to all those who contributed to the completion of this research, in particular, I would like to thank:

1. Prof. P.M. Dewilde for providing me the opportunity to participate in a wonderful and challenging project and his advice and guidance in solving general problems;
2. Dr. Ed F. Deprettere for continuously stimulating me to make greater progress and for his patience while reading my preliminary drafts of the thesis;
3. Prof. F.W. Jansen and A.J.F. Kok for many valuable discussions, especially in the development of the parallel image rendering algorithm;
4. Prof. G.L. Reijns and J.P.M. van Oorschot for supporting the evaluation of BONeS® and many useful discussion on network design;
5. Dr. J. Bu and G. Boersma from TPD-TNO at Delft for discussing the specifications of the prototype system;
6. G.J. Hekstra and M.T. Verelst for many fruitful discussions during the development of the parallel algorithm and architecture, and preparing beautiful pictures for my thesis;
7. The secretary Corrie Boers-Boonekamp for assisting where needed;
8. ARCAVIS 3D-COMPUTERGRAPHICS for providing the model of scene *man* and
9. ERSO-ITRI in Taiwan for providing the opportunity of doing researh here.

Finally, my most special thanks go to my wife, Li-Wen, and three children for being with me for four years and constantly supporting my work all the times.

About the Author

Li-Sheng Shen was born in Taipei, Taiwan, on February 1, 1955. In 1976, he received B.Sc. degree from the National Chiao Tung University, Taiwan. Two years later, he received M.Sc. degree from the same university. In 1980, he joined Industrial Technology Research Institute (ITRI) in Taiwan, where he completed several Integrated Circuits, including single-chip microcomputers, general-purpose microprocessors and graphic controllers. In 1985, he became the manager of Microcomputer IC Design Department of Electronics Research and Service Organization (ERSO). At the end of 1988, he joined the Network Theory Section of the Department of Electrical Engineering at the Delft University of Technology on basis of a research program between TNO and ITRI. In July 1989, he participated in the radiosity project and embarked on his work towards a Ph.D. He has contributed to the development of a parallel image rendering algorithm and architecture. At present, his main research interests are in the fields of computer graphics, virtual reality and parallel processing.