INFORMATION TO USERS

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

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

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

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

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

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600
EFFICIENT VISUALIZATION OF LARGE DATASETS USING PARALLEL PROCESSING AND VISIBILITY COMPUTATION

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy in the Graduate School of The Ohio State University

By

Po-Wen Shih, B.S., M.S.

* * * * *

The Ohio State University 1999

Dissertation Committee:

Professor Roni Yagel, Adviser
Professor Roger Crawfis
Professor Wayne Carlson

Approved by
Department of Computer and Information Science
ABSTRACT

The increasing size and scene complexity of graphic datasets constantly demand more efficient visualization techniques. Large graphic datasets, with hundreds of megabytes of data or hundreds of thousands of polygons, are gradually becoming a norm in today's applications, such as CAD/CAM (Computer Aided Design/Manufacturing), architectural modeling, computational fluid dynamics, and medical imaging. Due to the size and complexity of datasets, real-time or interactive rendering in such applications presents a very challenging problem.

In this dissertation, we present algorithms specially designed for the purpose of rendering large datasets. The design of our algorithms are mostly based on the application of space subdivision and tracing of visual rays. By employing a space subdivision technique, our algorithms are able to break down a complex scene to many simpler subdivisions, each of which contains much fewer objects for a more efficient management. By adapting the concept of visual ray tracing, our algorithms are able to render only visible parts of a complex model. Since the size of our datasets is large, it is essential to avoid redundant computations spent on objects that would not contribute to the final result.
Our first algorithm is a parallel ray casting algorithm, implemented on a distributed-memory multiprocessor system using message-passing mechanism. Our contribution lies in the development of a parallel rendering algorithm which combines the rendering efficiency of an image-order approach (such as early ray termination and adaptive sampling), with the deterministic communication order of an object-order approach. Parallel ray casting algorithms usually suffer from a high communication cost since the traversal of rays usually leads to an irregular object-space access pattern. Our new algorithm totally eliminates such a problem by imposing a front-to-back access order for all object communications. A new fetch list mechanism and a complementary cache (message buffer) scheme are designed to take full advantage of such an order, resulting in great cache efficiency (no cache conflicts or thrashing) and a high degree of latency hiding. To achieve these benefits, the algorithm requires only a small fraction (several hundred kilobytes) of system memory as cache to operate optimally. The low cache memory requirement is an important characteristic, since the dataset will use most of the processor's memory. Our new parallel algorithm is very efficient, capable of rendering 21 frames per second for a volume of $256^3$ voxels and one frames per second for a volume of $512^3$ voxels on 128 processors. It is also a very scalable solution, reaching 80% of parallel efficiency for a volume of $512^3$ voxels on 128 processors. The algorithm can also be extended to render multiple images from a static scene simultaneously. Such a "multiframe" extension can further improve the rendering efficiency by taking advantage of the frame-to-frame cache coherence across multiple image frames.
Another algorithm presented in this dissertation is a new visibility computation algorithm to accelerate rendering of large architectural models. Since most architectural models are oriented (with floors and ceilings), we can use a 2D floor plan to represent the partitioning structure (e.g. rooms) of a 3D architectural model. A 2D regular grid is superimposed on the floor plan by applying a space subdivision, and visibility information is computed based on subdivided regions by using walls as major occluders. The result is an efficient object-space visibility determination algorithm, which can efficiently cull away most invisible objects with respect to a view point. Our algorithm is designed to be used as a preprocessing step to compute and save the visibility information in advance. To reduce the storage requirement for storing the preprocessed visibility information, a difference list scheme is developed, which employs a mechanism to save only the differential visibility information between neighboring regions, and provides a quick restoration (with user adjustable parameters) to the complete visibility during run-time. With the difference list scheme, visibility information can be stored at only a small percentage (about 11% to 15%) of the full set's memory cost, allowing a space subdivision of higher resolutions to be used in the visibility computation (and thus more accurate visibility information).
ACKNOWLEDGMENTS

First, I would like to thank my adviser, Dr. Roni Yagel, for his enormous support, encouragement, and guidance throughout my study at OSU. Without his constant encouragement and help, I would never have finished my Ph. D. degree. I would like to thank Dr. Roger Crawfis and Dr. Wayne Carlson for serving in my dissertation committee. Their feedbacks and comments greatly improve the contents and presentation of this dissertation. I would also like to express my deep gratitude to my fellow groupmates Naeem Shareef and Asish Law. Naeem volunteered his valuable time to proof-read the dissertation drafts and helped me improve my English writing. Asish provided many insightful discussions which contribute to a portion of my research.

I would like to thank Dr. Wayne Carlson, Dr. Roni Yagel, and Don Stredney for providing me the opportunity to work on various projects at ACCAD and OSC. The experience of working in these interdisciplinary environments help strengthen my research motivation and broaden my academic experience at OSU. I would also like to thank Phil Ritzenthaler at ACCAD and Dennis Sessanna at OSC for their timely helps in solving all kinds of computer problems that I have encountered during the course of my research.
Many people have helped shape my experience at OSU. Their influences on me, whether inside or outside the field of Computer Science, have become an invaluable treasury in my life. At the risk of leaving someone out, I would like to thank: Asish Law, Naeem Shareef, Raghu Machiraju, Ed Swan, Torsten Möller, Klaus Mueller, Yair Kurzion, Don Stredney, Dennis Sessanna, Richard Parent, Chiu-Huang Huang, Dhabaleswar Panda, Doug Pride, Roger Crawfis, Wayne Carlson, and of course, Roni Yagel. I would also like to thank my friends at the Columbus Chinese Christian Church for their constant prayers and support.

I would like to thank my father and mother, Ping-Yong and Ah-Ching Shih, for their unconditional support and faith in my pursue of the Ph. D. degree. My wife Jane, with all the efforts she made to support our family, deserves this degree more than I do.

Finally, and most importantly, I would like to thank Jesus Christ, my Lord and Savior, who is always full of grace toward His people. His unchanging presence in my most struggling times has been the most precious memory during my years at OSU. May all the achievements and honors be dedicated to the Son of God, who is the creator and true master of all sciences and technologies.
VITA

1966. ....................... Born - Taipei, Taiwan.

1988. ....................... B.S. Computer Science and Information Engineering,
                        National Taiwan University, Taipei, Taiwan.

1993. ....................... M.S. Computer and Information Science, The Ohio State
                        University, Columbus, Ohio.

1993 - present ............ Graduate Teaching and Research Associate, The Ohio State
                        University, Columbus, Ohio.

PUBLICATIONS

[1] Roni Yagel, David Reed, Asish Law, Po-Wen Shih, and Naeem Shareef, “Hardware

[2] Don Stredney, Roni Yagel, Wayne Carlson, Torsten Moeller, Po-Wen Shih, and
    Mark Fontana, “Assessing User Proficiency through Virtual Simulations”, presented
    at the conference of the Radiological Society of North America (RSNA), Pittsburgh,
    June 1997.

    Interactive Decision Making”, to appear in Proceedings the Thirty-Second Hawaii
    International Conference on System Science (HICSS-32), Maui, Hawaii, January
    1999.
FIELDS OF STUDY

Major Field: Computer and Information Science
# TABLES OF CONTENTS

ABSTRACT ........................................................................................................................... ii  

ACKNOWLEDGMENTS ..................................................................................................... v  

VITA ................................................................................................................................... vii  

LIST OF TABLES .............................................................................................................. xii  

LIST OF FIGURES .......................................................................................................... xiii  

1  Introduction ................................................................................................................... 1  

2  Direct Volume Rendering .......................................................................................... 10  
   2.1  Optical Models for Volume Rendering ......................................................... 11  
   2.2  Sequential Volume Rendering Algorithms ................................................... 14  
      2.2.1  Voxel Projection .............................................................................. 15  
      2.2.2  Shearing-Based Transformation ..................................................... 16  
      2.2.3  Splatting .......................................................................................... 17  
      2.2.4  Ray Tracing ...................................................................................... 18  
   2.3  Parallel Volume Rendering Algorithms ...................................................... 21  
      2.3.1  Object-Parallel Methods ................................................................. 22  
      2.3.2  Image-Parallel Methods ................................................................. 24  
      2.3.3  Hybrid Methods ............................................................................. 26
5.1 Overview of the Method ................................................. 103
5.2 Model Preparation ....................................................... 105
  5.2.1 Dataset Assumptions ............................................... 105
  5.2.2 Space Subdivision ................................................... 107
5.3 Visibility Computation .................................................... 109
  5.3.1 A Basic Algorithm .................................................. 109
  5.3.2 An Incremental Algorithm ....................................... 113
  5.3.3 The Corridor Blocking Test ...................................... 115
  5.3.4 Corridor Containment Test ....................................... 119
5.4 Walkthroughs with Visibility Information ......................... 122
5.5 Memory Management with Difference Lists ....................... 123
5.6 Results ........................................................................ 130
  5.6.1 Visibility Preprocessing .......................................... 130
  5.6.2 Difference Lists ..................................................... 133
  5.6.3 Walkthroughs ......................................................... 137
5.7 Discussion ...................................................................... 138
6 Conclusion ........................................................................ 142

Bibliography ........................................................................ 146
LIST OF TABLES

Table 4.1 Volume datasets ................................................................. 76
Table 4.2 Average frame time (in seconds) for rendering one frame per rendering session ................................................................. 78
Table 4.3 Average frame time (in seconds) for rendering 5 frames per rendering session ................................................................. 81
Table 4.4 Average frame time (in seconds) for different rendering settings. “1f” denotes single-frame rendering, while all others are multiframe rendering with different number of frames. Bold-faced numbers reflect a strong communication presence in the rendering time ................................................. 91
Table 4.5 Comparison of different parallel rendering algorithms ................. 97
Table 5.1 Abbreviations for various configurations of the Office (OF) and MassHall (MH) datasets ................................................................. 130
Table 5.2 Visibility preprocessing statistics for the Office and MassHall datasets ................................................................. 131
Table 5.3 Difference list statistics for the Office and MassHall datasets. The Diff_Threshold is 0.1 ................................................................. 133
Table 5.4 Walkthrough performance (all times are in seconds) ......................... 137
### LIST OF FIGURES

| Figure 3.1 | Four space subdivision schemes. (a) Cartesian grids, (b) quadtree/octree, (c) BSP tree, (d) k-D tree. A k-D tree is a BSP tree with only axially-aligned dividers. | 30 |
| Figure 4.1 | The Anchored Ray Transport (ART) algorithm | 43 |
| Figure 4.2 | Partitioning a volume dataset: (a) slices (b) shafts (c) cells | 45 |
| Figure 4.3 | Initialization of cells and rays. (a) The FTB list order was calculated for all cells with respect to the current view direction. All rays are initialized and “queued” at the first cell they enter. Notice that if ray $c$ advanced out of the volume, it would intersect with cells 3, 6, 9, 14, 19, 25, 29, 33 and 35 in an increasing rank order. (b) An illustration of the ray list structure with cells 1-4 and 35-36 shown with their initial contents. In this case, cells 1-4 are all “engaged.” | 49 |
| Figure 4.4 | Problems in the “prefetch-all” design. (a) If the unfinished rays go all the way through the object space, remote cells in the three areas enclosed by bold lines will be fetched but never used. (b) When the current anchor cell (cell A) is being processed, the prefetched list covers the bold area in the next column, which includes remote cell B and some others. The lack of ray processing between A and B may not provide enough time to hide communication latency for cell B, given that other remote cells between A and B are being requested ahead of cell B at the same time. The problem is even more obvious between cell B and C. | 55 |
| Figure 4.5 | Determine the best permutation of the FTB list order to minimize the “near-neighbor” problem | 58 |
| Figure 4.6 | A 2D example for the basic multiframe rendering. (a) Multiple frames are positioned around the dataset with their own view directions. (b) Frames with view directions within the same quadrant can be rendered together by the basic multiframe rendering extension | 64 |
Figure 4.7 Achieve the 180-degree multiframe rendering with dimension-reduced FTB order. (a) The basic multiframe extension. (b) Dimension-reduced method. (c) Using holding buffers to implement the dimension-reduced method. The solid cells indicate the required buffer size for the "anchor" object.

Figure 4.8 Determine the FTB permutation and the number of holding cells.

Figure 4.9 The spiral order: (a) cell ranks assignment (b) the effect of regular traversal (illustrated by the counter-clockwise coil) (c) the effect of reverse traversal.

Figure 4.10 Uses the 180-degree technique to implement the spiral order traversal. Home cells are filled with stripe and the cache window is filled with solid shade. The above figures show three snapshots of the spiral order traversal with a sliding cache window.

Figure 4.11 Different multiframe zones for (from left to right) 90-degree, 180-degree, and 360-degree multiframe extensions.

Figure 4.12 Images of the test volumes: (a) Sod (b) Cube (c) Head (d) Mix.

Figure 4.13 Speedup and scalability for rendering a single frame.

Figure 4.14 Speedup and scalability for 5-frame, 90-degree multiframe rendering.

Figure 4.15 Cell size analysis for (a) Mix256 (b) Mix512 (c) Cube128 (d) Cube256. Each data series represents the rendering time for a particular cell size. The cell size of voxels is applicable only to Mix512 since smaller volumes cannot generate enough cells for all processors. Rendering time is normalized to the time for the cell size of voxels.

Figure 4.16 Near/Far cache size analysis. (a) and (b): Rendering Mix512 with a cell size of voxels on different number of processors. (c) and (d): Rendering Mix512 on 32 processors with different cell sizes. Far cache is set to 0 when analyzing the near cache, but near cache is set to one cell when far cache is analyzed. Our algorithm requires at least one near cache buffer to execute.

Figure 4.17 Cache type analysis. The chosen cache size (near + far) is 20 cells for 60 cells for 1, and 200 cells for. Rendering time is normalized to the fastest one in each data series.

Figure 4.18 Comparison of rendering timing for different settings of (a) Head256 and (b) Mix256 datasets. Rendering time is normalized to the complete timing to show their relative performance.
Figure 4.19 Speedup for different rendering settings on a cluster of workstations with (a) ATM and (b) Ethernet networks.

Figure 4.20 Comparison of rendering times for different settings of Mix256 on an ATM network. Rendering time is normalized to the complete timing to show the relative performance.

Figure 4.21 Timing comparison for different multiframe approaches on an ATM network.

Figure 5.1 Blocking configuration for the simplified sight corridor approach. (a) Blocked. (b) Not blocked (since no single connected wall segments can block the entire corridor. Dark lines represent wall segments.

Figure 5.2 (a) Axial skeleton representation for both a closed curve and an open curve. The exterior curve has clockwise vertex order and the interior curve has a counter-clockwise order. (b) A simple line segment sketch and its axial skeleton representation.

Figure 5.3 Scan-conversion of a wall segment to a space subdivision grid. Wall cells are shown in solid squares. Each wall cell registers the wall segment by saving a pointer to the first vertex of the line. The colinear points represent points at unit cell length along the line segment. They are stored in an external list reachable from $v_i$.

Figure 5.4 The cell-to-cell visibility algorithm template.

Figure 5.5 A continuous corridor sweep and its corridor cells.

Figure 5.6 A blocking configuration from Src to Dest.

Figure 5.7 The incremental algorithm introduces more over-estimation and non-conservative results. Cell 1 is a corridor cell for all three different sight corridors and cell 2 is only covered by corridor A. In (a), cell 1 is claimed visible by corridor A and B and is indeed visible since the blocking in corridor C does not fully obscure the cell. In (b), even though the blocking in corridor B is enough to fully hide out cell 1, corridor A will make cell 1 still visible. For a non-conservative case, observe the upper-left corner of cell 2. Although cell 2 is claimed not visible by corridor A, it is difficult to tell whether the corner can be fully blocked by the given blocking configuration.

Figure 5.8 The extended sight corridor expands the coverage of the continuous sweep to include “bumps” not covered by the original sight corridor.
Figure 5.9  (a) The definition of a sight corridor. (b) An invalid blocking configuration through the neutral zone. .......................................................... 117

Figure 5.10  The corridor blocking test for the basic visibility algorithm. .......... 118

Figure 5.11  The corridor blocking test for the incremental visibility algorithm... 120

Figure 5.12  The corridor containment test. ..................................................... 121

Figure 5.13  A walkthrough of an architectural model with preprocessed visibility information. (a) Space subdivision grid and the object/wall placement. (b) A view of the 3D model. (c) A snapshot of the walkthrough. Objects in the PVS of the current observer's cell are highlighted. The arrow shows the observer's position. (d) Another snapshot. ............................................. 124

Figure 5.14  The difference list algorithm. ..................................................... 126

Figure 5.15  An example of the distance transform computation. Cells not in the region are shown in solid color, while difference edges in the region are shown by short bars. Max_Distance is set to 3. The figures show the status of computation at: (a) the beginning of a new region; (b) the end of the first regular round; (c) the end of the regular second round; (d) the end of the extra round. Small arrows show the direction a cell follows when searching for a well. ...... 129

Figure 5.16  The floor plan of the MassHall dataset. Objects are shown in bounding boxes. .......................................................... 131

Figure 5.17  PVS storage reduction for dataset MH150 showing various settings of Max_Distances and Diff_Threshold. The average PVS size is normalized to size when the difference list scheme is not used. ......................... 134

Figure 5.18  Visualization of the difference list scheme applied on the MH150 dataset. The Max_Distance is 20 and Diff_Threshold is 0.1. (a) Region distribution. Regions are color-coded by their region ID. The floor plan of the Masshall is shown here for comparing the regions with the original room partitions. (b) Well distribution (shown as white dots). (c) PVS size distribution. All wells store complete lists, while other cells store only the differences. The color bar shows the mapping scale. Black area represents cells with no stored objects. 136
CHAPTER 1

Introduction

Images and graphical presentations are one of the most natural means by which humans perceive and understand scientific phenomena and abstract notions. The intrinsic limitations of our perceptual system hinder us from interpreting the vast amount of information presented by a plethora of modern applications when the information is not characterized in a graphical form. In many application fields, visualization has become a necessary process for analyzing results (e.g. global climate modeling), or has introduced an alternative way to accomplish work other than the traditional approach (e.g. product prototyping or virtual surgical planning.) The recurring need for visual realism and responsive interaction is a driving force behind the continual development of new visualization techniques.

Since the introduction of graphic workstations in the 1970s, computer graphics has seen rapid growth and has become widely embraced by many disciplines. Applications such as Computer Aided Design/Manufacturing (CAD/CAM), architectural modeling, flight simulation, terrain rendering and virtual reality rely heavily on computer generated graphical representations. Traditional computer graphics uses surface-based primitives,
usually polygons, to represent 3D models. Models constructed in this way present only the shell structure of an object, whose interior is empty. Since polygonal primitives are planar surfaces, rendering a 3D model consists of sending a list of polygons in 3D space into a graphic rendering pipeline, which transforms, clips, interpolates, shades and scan-converts polygons to color pixels [31]. Since typical realistic models may contain hundreds of thousands of polygons [78], graphic rendering is inherently a computationally intensive and time consuming process. Even with the help of special, dedicated graphic acceleration hardware, rendering such a large polygonal dataset is only manageable at best, let alone achieving interactive or real-time performance, as is sought after by many applications.

The debut of supercomputers and the emergence of 3D digital data acquisition devices, such as computed tomography (CT) and magnetic resonance (MR) scanners, open a new dimension for graphic applications. Increasingly, larger 3D datasets are now available from scientific simulations running on supercomputers, and from 3D scanners which sample real world objects. Models generated by these approaches encode the data information, either in scalar or vector format, directly on spatial grid locations on a 3D lattice. A volume is a three dimensional array of such data points, where each data point, or voxel, represents a unit of space. Unlike surface-based models, volumes preserve the interior information of objects, which allows for easy implementation of operations such as addition, subtraction, collision detection, and deformation [48]. However, the enormous size of these datasets places an even larger workload on the graphic workstations over the
demands of surface-based models. For example, a medium resolution volume of $256^3$
voxels would require a storage of 16 megabytes (assuming one byte per voxel), while a
high resolution CT scan of a human body can generate more than half a gigabytes [1]. A
simple arithmetic calculation shows that without paying special attention to accelerate the
volume rendering process, it could require on the order of tera-flops (floating-point
operations per second) to achieve real-time performance (30 frames per second) for even a
moderate size volume and a simple illumination model [119]. Such a rigid computational
demand is well beyond the capacity of any graphic workstation available. Applications
such as computational fluid dynamics, astrophysical measurement, medical imaging and
non-destructive evaluation (NDE) all generate or operate on massive size 3D volumes.
Although computer hardware technology has consistently improved over the years, real­
time visualization of such immense datasets still presents a great challenge to researchers.

Volume graphics [48] is a growing subfield of computer graphics which focuses on
developing new techniques to efficiently manipulate and handle such gigantic 3D datasets.

This dissertation addresses the problem of visualizing these large datasets as just
described. For "large datasets," we assume graphic models that are too large for
interactive rendering (between several to fifteen frames per second) by current leading­
edge graphic workstations. Typical datasets contain hundreds of thousands of polygons or
hundreds of megabytes of voxels. The goal of this dissertation is to investigate new
techniques to overcome this performance barrier and to improve rendering speed up to a
higher interactive frame rate and ultimately to achieve real-time performance. Previous
research to accelerate graphics rendering for exceptionally large datasets has branched into four main directions: (i) utilization of special-purpose graphics hardware, (ii) optimization of software rendering algorithms, (iii) reduction of model complexity, and (iv) implementation on general-purpose parallel computers.

The utilization of special-purpose graphics hardware is the most straightforward solution to the increasing demand for real-time visualization of high resolution models. In this arena the traditional z-buffer based rendering pipeline has matured over the years and is incorporated as a standard into most of today’s rendering engines available commercially on graphic workstations [5][73]. Other types of acceleration hardware have also been investigated for both surface [71][84][34] and volume rendering [79][40][49]. Whereas, employing specialized hardware may achieve the best performance result, it could also incur the most expensive cost among all approaches.

Researchers have also optimized software rendering algorithms to achieve greater performance. Rendering methods, distinguished by the direction of transformations, can be divided into two groups: feed-forward and backward-feed. The feed-forward (or object-order, or forward viewing) approach traverses the object space to transform and project object primitives onto the screen. The backward-feed (or image-order, or backward viewing) approach casts rays from the eye, through each pixel on the screen, and into object space to collect object samples. Many performance-enhanced rendering algorithms for the feed-forward approach are available for volume rendering, due to the lack of
available hardware accelerators, such as those available to surface-based rendering. Techniques such as marching cubes [63] and other iso-surface based approaches [62] extract a surface structure from the volume and thus are able to utilize the efficient surface-based algorithms. Methods that directly render voxels without first transforming them into geometric primitives have been explored in several directions. Voxel projection [32][85], splatting [107], and shearing-based transformations [51][41] are among the more popular feed-forward methods that directly render a volumetric dataset. Optimization of backward-feed methods such as ray tracing and ray casting (a visibility ray-only version of ray tracing) has received extensive attention from researchers in both surface and volume rendering subfields [24][47][60][61][78][115][116][124]. A more detailed review of these direct volume rendering methods (which render voxels without first generating geometric primitives) will be given in Chapter 2.

Rendering performance can also be improved by reducing model complexity, which is usually achieved by applying level-of-detail (LOD) or visibility computation. LOD methods [35][13] display different approximations of a complex model according to its contribution to the final image. The approximations can either be precomputed as separate objects by applying polygon/mesh simplification methods [29][42], or be encoded into a unified model representation like progressive simplicial complexes [81], simplification envelopes [16], or hierarchical vertex trees [65].
Visibility computation, on the other hand, aims at reducing the rendering load by removing objects that are not visible from the viewpoint. Methods include simple techniques such as view frustum clipping and back-face culling [31], to complex ones like visibility determination in a general environment [39][123][118]. A particularly interesting group of applications for visibility computation is rendering models with densely populated large occluders, such as buildings and rooms [98]. In a large architectural model, scenes usually have great depth complexity with many large and static occluders (e.g., walls, pillars, closed doors and windows.) This allows for special visibility algorithms that can efficiently cull away large portions of complex models. In Chapter 3 we will have a more focused review of research in visibility computation.

The growing popularity of general-purpose, massively parallel processors (MPP) spawned a new wave of research on parallel rendering algorithms. Molnar, et. al. [70] introduced a sorting classification framework for feed-forward parallel rendering algorithms. Algorithms are classified into three categories according to when the redistribution of primitives occur: sort-first, sort-middle, and sort-last [70]. This framework also provides an analytical technique to compare and design new parallel rendering algorithms in each category. A common approach adopted by surface-based algorithms is the sort-middle distribution [27][20][5], while volume rendering algorithms usually exploit the sort-last design [58][107][67][52]. A concise review of issues and developments in parallel rendering can be found in [19]. Parallel implementation of
various volume rendering methods can be found on many different parallel architectures. A review of those major efforts will be included in Chapter 2.

In this dissertation, we propose two new techniques to improve the efficiency of rendering large datasets. The first technique is based on an implementation of a parallel ray casting algorithm on a distributed-memory multiprocessor system. Our contribution lies in the development of a parallel rendering algorithm which combines the rendering efficiency of an image-order algorithm (such as early ray termination and adaptive sampling) with the deterministic communication order of an object-order algorithm. Parallel ray casting algorithms usually suffer from a high communication cost due to the nature of random object-space traversals caused by different rays. Our new algorithm, on the contrary, avoids such a problem by imposing a front-to-back access order to coordinate all object communications. We designed a new fetch list mechanism and a complementary cache (message buffer) scheme to take advantage of such an order, allowing for a great cache efficiency (no cache conflicts or thrashing), and a high degree of latency hiding to improve rendering performance. The algorithm requires only a small fraction (several hundred kilobytes) of system memory as cache to operate optimally. This is an important characteristic, since most of the processor’s memory may be needed for the large sized dataset. Our new algorithm is very efficient, capable of rendering 21 frames per second for a volume of $256^3$ voxels and one frames per second for a volume of $512^3$ voxels on 128 processors. It is also a very scalable algorithm, reaching 80% of parallel efficiency for a volume of $512^3$ voxels on 128 processors. We also extend the algorithm to handle
simultaneous rendering of multiple animation frames (called *multiframe rendering* [56].)

By taking advantage of the frame-to-frame cache coherence in the multiframe rendering situation, our algorithm can further improve the rendering performance for a sequence of images.

The second innovation in this dissertation is a new visibility algorithm to accelerate rendering of large architectural models. Our method is especially tailored for architectural models as it takes advantage of various properties of buildings (*i.e.*, connected, vertically-standing wall segments). Since most architectural models are oriented, we utilize a 2D floor plan, which can be derived from a 3D architectural model, for computing visibility information. A 2D regular grid is superimposed on the floor plan by applying a space subdivision, and visibility information is computed based on subdivided regions. The result is a conservative, but fast, object-space visibility determination algorithm, which can efficiently cull away most invisible objects with respect to a viewpoint. Our algorithm is designed to be used as a preprocessing step for computing the visibility information during off-line. The preprocessed visibility information is then used during run-time to accelerate the walkthrough speed. To reduce the storage requirement for storing the visibility information, we developed an efficient storage scheme, called *difference lists*. The difference list scheme employs a mechanism to save only the differential visibility information between neighboring regions, and provides a quick restoration (with user-adjustable parameters) to the complete visibility during run-time. With the difference list scheme, visibility information can be stored at
only a small percentage (about 11% to 15%) of the full set's memory cost, allowing a space subdivision of higher resolutions to be used in the computation (and thus more accurate visibility information).

The dissertation is organized as follows. In Chapter 2, we will review volume rendering, and both sequential and parallel implementations for direct volume rendering methods. Chapter 3 will offer an introduction to the visibility computation problem in computer graphics. Our new latency-optimized, parallel ray-casting algorithm will be discussed in Chapter 4. In Chapter 5, we will describe our efficient visibility determination algorithm for architectural model walkthroughs. Finally, Chapter 6 summarizes our work and sets forth directions for future research.
CHAPTER 2

Direct Volume Rendering

Direct volume rendering techniques strive to render a 3D volumetric dataset by directly performing the rendering computation on each voxel, without first extracting an intermediate geometric representation from the volume. There are several advantages of this approach over the indirect, or surface-extraction based, volume rendering methods [63][62]. First, it allows the interior of substances containing a mixture of different materials to be faithfully represented as a translucent volume (via classification and mapping of transfer functions [25].) By directly rendering the volume, the local variation of light absorption and emission of each voxel can be recovered and contribute to the interpretation of the final image. Secondly, subtle interfaces between materials usually form fuzzy surfaces that could not be easily reconstructed by most surface-extraction methods [59]. The binary decision used to extract the surfaces may introduce spurious features (new surfaces or holes) that are not presented in the original dataset. Finally, in applications such as virtual surgical training, rendering of the underlying substance may be at least as important, if not more important, as the rendering of the shell surface of the dataset.
The major disadvantage of direct volume rendering has been its performance. The enormous size of volumes means that transforming and illuminating each voxel is a tedious and intensive computational task. In addition, the lack of well-defined geometric properties of sampled voxels also presents difficulty when applying shading and illumination computations. This chapter serves as a background review of foundational work in direct volume rendering. First, we review the most popular optical models used for direct volume rendering in the following section. In Section 2.2 and Section 2.3, respectively, we survey major sequential and parallel algorithms for direct volume rendering. Throughout this dissertation, we will use ‘volume rendering’ and ‘direct volume rendering’ interchangeably.

2.1 Optical Models for Volume Rendering

In his survey paper [68], Max reviews several light interaction models as well as details of the physical and mathematical properties for density clouds. He describes optical models that range from simple to complex, i.e. from individual particles (such as light absorption or emission) to interactions among many particles with other external illumination sources (such as single scattering with shadow or multiple scattering.) Since volume rendering is a very computationally intensive task, most algorithms (especially those intended for large volumes) trade the physical accuracy of lighting with rendering speed. In this section we only review those optical models that are the basis of simplified methods which are commonly used in the field. Refer to [68] for more complicated and physically accurate models.
The simplest optical model for a density cloud is the absorption-only model. Under this model the density cloud is a collection of continuous, infinitesimally small particles that absorb all incident light. If we assume that the light is transmitted from one side \((s = 0)\) to the other side \((s = l)\) of a cloud, then the light intensity (denoted as \(I(s)\)) at distance \(s + \Delta s\) inside the cloud can be expressed as:

\[
I(s + \Delta s) = I(s) - \tau(s)I(s)\Delta s
\]

(2.1)

where \(\tau(s)\) is called the extinction coefficient which defines the rate at which light is occluded by the particles. After dividing both sides by \(\Delta s\), rearranging the terms, and taking limits, we have the following:

\[
\frac{dI(s)}{ds} = -\tau(s)I(s).
\]

(2.2)

The solution of \(I(s)\) can be expressed as \(I_0T(s)\), where the function:

\[
T(s) = e^{-\int_0^s \tau(t)dt}
\]

(2.3)

represents the transparency at \(s\) and \(I_0\) is the light intensity at \(s = 0\). Therefore, we can define the opacity at \(s\) as:

\[
\alpha(s) = 1 - T(s).
\]

(2.4)

For a constant extinction coefficient, we have

\[
\alpha(s) = 1 - e^{-\tau s}.
\]

(2.5)

Besides the absorption property, the particles in the cloud can also emit light or reflect light from an external illumination source. We can model the emission-only behavior as:

\[
\frac{dI(s)}{ds} = C(s)\tau(s) = g(s)
\]

(2.6)
where $g(s)$ is the glow function and $C(s)$ is intensity per unit projected area. By modeling both absorption and emission, we have the following:

$$\frac{dI(s)}{ds} = g(s)-\tau(s)I(s). \quad (2.7)$$

Solving the above equation gives:

$$I(D) = I_0T(D) + \int_0^D g(s)P(s)ds \quad (2.8)$$

where the eye is represented at distance $s = D$, and $P(s)$ is the transparency between $s$ and the eye, and is defined as

$$P(s) = e^{-\int_s^D \tau(t)dt}. \quad (2.9)$$

The practical implication of Equation 2.4 is to derive the correct opacity for each voxel when the volume is scaled or is viewed diagonally. For example, given a constant opacity at the edge of a unit-length voxel (i.e., $\alpha(1) = \Omega$) we can first derive the extinction coefficient $\tau = \log \left( \frac{1}{1-\Omega} \right)$ (assuming $\tau(s)$ is also a constant function), and then use $\tau$ to correctly calculate the opacity when a voxel is not viewed at unit intervals. Alternatively, Equation 2.8 indicates that the intensity of light reaching the eye will be the intensity of background light ($I_0$) attenuated by the absorption of the cloud ($T(D)$) plus the overall accumulation of self-emittance from position $s$ ($g(s)$) attenuated by the portion of the cloud from $s$ to the eye ($P(s)$.) In the simplified case when $C(s)$ is a constant function in Equation 2.6, Equation 2.8 can then be simplified to

$$I(D) = I_0T(D) + C \left( 1 - T(D) \right) \quad (2.10)$$
which shows a composition form similar to the over operator as introduced by Porter and Duff [82] for front-to-back composition:

\[ I_{\text{new}} = I_{\text{old}} \alpha_{\text{old}} + I_{\text{layer}} (1 - \alpha_{\text{old}}) \]  
\[ \alpha_{\text{new}} = \alpha_{\text{old}} + \alpha_{\text{layer}} (1 - \alpha_{\text{old}}) \]  

(2.11)  

(2.12)

Therefore, a density volume can be rendered by compositing the contribution of parallel sheets of particles along a ray, similar to the image processing techniques used in [82]. Most volume rendering algorithms use these simplified optical models for purpose of achieving efficient implementation and acceptable visual results.

### 2.2 Sequential Volume Rendering Algorithms

Graphic rendering algorithms, in general, can be divided into two types: feed-forward and backward-feed. For volume rendering, the performance bottleneck for feed-forward methods lies in the transformation and resampling steps. Since each voxel has to be positioned in screen space before projection, the cost of transforming the entire volume can be overwhelming. After this transformation is done, the volume must be resampled properly into screen space to prevent artifacts such as holes or aliasing. Resampling an arbitrarily oriented volume can be very time consuming since the process requires accessing irregular index patterns. Most acceleration techniques for feed-forward methods address either one or both of these problems. For backward-feed algorithms, the majority of time is spent on advancing rays through object space and intersecting rays with data voxels. Numerous methods have been proposed that take advantage of various forms of coherency in order to reduce ray-voxel computations. Other methods focus on expediting
ray traversal via efficient space cruising. We will review both types of algorithms in this section.

### 2.2.1 Voxel Projection

Early voxel projection methods include both volume traversal scheme, the back-to-front (BTF) [32] and the front-to-back (FTB) [85] projections. These algorithms transform voxels slice-by-slice in the specified order and project the transformed slices onto the screen without resampling. Since voxels are projected in a strict viewing order, they do not require a z-buffer to solve the hidden voxel removal problem and can support semitransparency. However, since they avoid the resampling process, the scale factor between voxel and pixel sizes is limited to be less than \( 1/\sqrt{3} \), or artifacts will occur.

To accelerate the viewing transformation, a table look-up method is employed by [32]. The method is based on the observation that the view transformation of voxels involves only the multiplication of the viewing matrix with integers in \([1, N]\). Therefore, the multiplication can be pre-calculated and stored in a table to speed up the transformation calculation. Machiraju and Yagel [67] propose an incremental transformation method which processes voxels in BTF order with seed phase (one voxel), beam phase (one column), voxel phase (one slice), and slice phase (the entire volume.) Except for the seed phase, which requires a full transformation of one voxel, all other phases can be incrementally computed from the results of the previous phase. The incremental calculation requires only three extra additions per voxel. An octree hierarchy
is exploited in [69], where similar-valued voxels are grouped into an octree structure. Transformation of voxels is performed in a top-down order of the octree hierarchy. The aggregation of similar voxels at the higher level node (with a larger space coverage) can be transformed together and rendered as a uniform supervoxel. Solid texture (3D texture map) hardware has also been exploited by [11][23] to accelerate the projection of voxels. In this method a volume is treated as a 3D texture, and the texture hardware is used to perform resampling on polygons embedded in the texture. The resampled polygons (which are equivalent to transformed and resampled slices of voxels) are then composited to form the image.

2.2.2 Shearing-Based Transformation

Shearing-based methods tackle both performance bottlenecks of feed-forward methods together. The 3D view transformation is decomposed into a sequence of shearing operations, which makes transformations faster and resampling simpler. Hanrahan [41] performs the 3D view transformation with three passes of 1D transformations. Each pass includes a 1D shearing transformation followed by a 1D resampling (parallel to the shearing axis) of the sheared volume. The final image is produced by compositing the final resampled volume using either the BTF or FTB volume traversal. The advantage of this method is that the 1D transformation is much faster (which can be optimized to use only one addition for each voxel), and the 1D reconstruction is much easier. However, it suffers from low image quality due to multiple resamplings and an approximation of a 3D reconstruction filter by a series of 1D filters. Lacroute and Levoy [51] further optimize the
approach with a *shear-scale-warp* approach. Here, a volume is rendered by translating individual slices into the sheared object space, projecting the sheared slices to a 2D intermediate image, and finally a 2D image warp is performed from the intermediate image to the final image. In the perspective case, the shearing step is followed by a scaling step to properly reflect perspective distortion. The major achievement of the algorithm is that it requires only two resampling operations (one in volume and one in image), thus improving both speed and image quality. The algorithm also utilizes a similar dynamic screen approach as used in [85] to skip over transparent voxels and opaque pixels during the resampling and projection step.

### 2.2.3 Splatting

The splatting technique is first introduced by Westover in [106]. The main idea is to approximate an expensive 3D reconstruction in voxel space by a 2D reconstruction in image space using a precomputed footprint table of the 3D filter kernel. Using a BTF traversal, voxels are projected to the screen, where the contribution of each voxel (covering several pixels) is composited on the screen. A later enhancement to this method collects splats onto a slice, where splats are summed together and the slice composited as a layer to the screen [107]. The latter approach is a more accurate approximation since it avoids the opacity interference of voxels in the same slice.

Hierarchical splatting [53] is a further improvement to the efficiency of splatting because it allows a uniform space to be rendered by a larger splat. The volume is stored in
a multi-resolution representation in a pyramid data structure similar to mip-map [31]. At each level of the pyramid, the volume is represented with an octree average, with the original volume sitting at the bottom of the pyramid. The splatting procedure starts at the top level of the pyramid and traverses the nodes in the octree in a BTF order. If the average error of a node is within a user-specified threshold, the node is rendered as one splat (in correct size;) otherwise, its octree expansion at the next level is explored. The algorithm utilizes Gouraud-shaded polygons to approximate the footprint kernel, which allows for the use of conventional graphics hardware to accelerate the interpolation and compositing process. Similar approaches which utilize texture mapping hardware can be found in [114][21]. In these approaches a footprint kernel is precomputed and stored as a texture map. Rendering involves mapping such a footprint texture to a rectangle facing the viewer, and compositing these texture-mapped polygons to form the image. The main advantage here is that texture mapping, interpolation, and compositing are all supported by today's graphics hardware.

2.2.4 Ray Tracing

Ray tracing is one of the earliest methods adopted to render volumetric datasets, and as a result, is also one of the most exploited approaches in the field. Kajiya and von Herzen [47] propose a complex model for light scattering in a density volume and offer a two-pass approximation method. Sabella [86] uses a simpler absorption plus emission model which is similar to the one described in Section 2.1. Levoy [59] devises transfer functions to display iso-surfaces or boundary surfaces in his ray tracing approach. All
these methods have concentrated on the issue of ray-volume interaction, but rarely address accelerating the rendering process.

Other algorithms are designed to take advantage of various types of coherency existent in the ray tracing approach. Levoy [60] exploits image space coherency by applying an adaptive refinement strategy. First, rays are initialized at a small subset of pixels, and then additional rays are cast only if pixels in a near neighborhood differ from each other. A similar strategy is used by Danskin and Hanrahan [24] where the sampling resolution in object space is varied instead. They use a min-max pyramid volume representation, which allows a ray to quickly detect regions of interest and adjust the sampling distance accordingly. Later, Levoy [61] also proposes an octree-based scheme in which each octant is augmented by an occupancy flag, allowing a ray to skip empty space.

Other ray acceleration approaches use a 3D distance transform [15][124]. These methods take advantage of the regular cubical structure in a volume to encode distance information in the voxel grid, which allows rays to quickly space leap to the next data voxel. Avila, et. al. [6] extend the PARC (Polygon Assisted Ray Casting) approach, in which a convex polyhedral envelope surrounds the actual data volume. First, the envelope is rendered by the graphics hardware to decide the entry (the front faces of the envelope) and exit (the back faces) points of each ray. Rays are then sampled within the envelope only.
Yagel, et. al. [116] propose a discrete ray tracer which employs 3D raster rays to sample the voxel space. A 26-connected discrete ray is used to quickly advance through empty space until a data voxel is encountered, then a more accurate six-connected ray is used to perform ray-object intersection. A one-voxel proximity cloud surrounds all object regions to allow efficient switching between the two types of rays. Another discrete form of ray tracing is template-based ray tracing introduced by Yagel and Kaufman [115], which exploits inter-ray coherency in parallel projections. Parallel rays are generated from a base plane using a precomputed ray template, which guarantees that no voxel is skipped or over-sampled. An intermediate image is first rendered on the base plane, which is then projected back to the screen.

Coherent ray tracing of large graphic datasets has also earned attention from researchers. The main problem is how to efficiently manage memory, since the volume can not fit into memory. Pharr, et. al. [78] develops a memory coherent ray tracing scheme that spawns ray segments which are independently traced in order to adapt to the geometry currently in memory. Ray segments are grouped into a scheduling grid, which overlaps on top of a geometry grid, which partitions the volume. A cost function and a benefit function are used to direct data transfer between memory and the disk. Law [54] exploits a ray casting scheme to render compressed volumes, which guarantees that volume blocks are decompressed only once. The algorithm is a uniprocessor implementation of [56].
2.3 Parallel Volume Rendering Algorithms

Parallel rendering algorithms are classified according to how the computation workload or tasks are decomposed [19][74]. Such a classification framework possesses a great resemblance to the sorting classification for parallel feed-forward rendering algorithms [70]. In an object-parallel method, individual tasks are derived from data partitioning in object space. Each task contains a subset of objects and are rendered in parallel, which requires pixels to be composited with others to form the final image. Alternatively, an image-parallel method defines tasks based on an image space partition, where a task renders a portion of the screen from contributing objects. Generally speaking, the object-parallel approach more easily achieves load balancing, with the possibility of intensive network traffic during the compositing phase. The serialization and order-dependent nature of the compositing phase means that scalability to a large number of processors is difficult. The image-parallel approach avoids this integration step and spreads out the communication across the entire rendering phase. However, it may be prone to an excessive amount of data communication because object space traversal is highly irregular and view dependent. To balance the overhead between these two approaches, hybrid methods try to incorporate features from both object- and image-parallel methods. The rendering tasks are defined by partitions in both image and object spaces, and the rendering process has two stages, where intermediate results are communicated from the object-parallel tasks to image-parallel tasks, or vice versa. Note that in this classification scheme, the flavor of parallelism does not necessarily imply a certain rendering algorithm. For example, an object-parallel method may use segmented
ray tracing [45] to render a partial image on each processor and then composite partial results into a final image.

2.3.1 Object-Parallel Methods

Researchers have found embedded parallelism in several sequential volume rendering techniques. Westover [108] is the first to conduct a parallel implementation of his splatting technique on a network of SUN workstations. His work utilizes $N^2$ processes to splat in parallel each slice of voxels in FTB or BTF order into an image sheet buffer, and then composites these sheet buffers by $N$ compositing processes. The incremental transformation proposed by Machiraju and Yagel [67] is implemented on an IBM Power Visualization System (PVS), which is a shared-memory, multiple-instruction-multiple-data (MIMD) multiprocessor machine. First, each processor is assigned a slab of voxels, and then slices in each slab are transformed and splatted in BTF order to generate a partial image. The partial images are composited by a binary-tree hierarchy of processors to create the final image. Lacroute’s shear-scale-warp factorization [52] is successfully implemented in parallel on an SGI PowerChallenge. The implementation minimizes synchronization time using a dynamic load balancing strategy, and relies upon fast shared-memory accesses to hide the latency.

The shearing-based transformation [41] is optimized by several researchers on single-instruction-multiple-data (SIMD) multiprocessors such as CM-2 [88], MasPar MP-1 [105][112], and MasPar-MP-2 [113], and also on MIMD multiprocessors such as
Proteus [111]. Schroder and Salem [88] and Vezina, et. al. [105] employ a similar approach that decomposes an affine transformation into a series of shearing operations. The massively-connected processors in an SIMD environment allow shearing operations to be done in a quick, near-neighbor, regular communication pattern. These methods, however, suffer from low image quality since multiple 1D resampling passes (five in [88] and three in [105]) are required. The problem of poor reconstruction in this method is addressed by Wittenbrink and others [112][113][111] using permutation warping. Permutation warping is an efficient way to perform affine transformations (scaling is usually excluded [112]) and resampling of a 2D or 3D raster. The warping step performs a series of 1D shearing transformations on grid points in object space, in order to locate the resampled points (in object space) for grid points aligned in screen space. The strength here is that the resampled point is guaranteed to be in the near neighborhood of the original grid point in object space, allowing a high quality reconstruction to be done easily through regular communications. A permutation send (one-to-one non-conflict global communication) is carried out after the warping step to send the reconstructed samples to their destination grid points in the screen space. The method is especially suitable for SIMD multiprocessors since it consists of highly parallel operations which require only regular-pattern communication on a richly connected environment. Data dependent optimization (an octree-based hierarchical space representation) can also be exploited [113].
Segmented ray tracing is used by Ma, et. al. [66] and Silva [92] to take advantage of object-parallelism. The algorithm in [66] partitions the volume into a k-D tree hierarchy, traces rays in parallel within a partition on each processor, and composites the partial images from all processors in a binary-swap composition. The innovative binary-swap composition is a distributed version of a binary-tree composition, where at each level of compositing all processors under the current compositing subtree will share the workload to produce the result at the root. Their work demonstrates low communication and good scalability on a CM-5 and a network of workstations. Silva [92] employs a content-based, static load balancing strategy to generate the k-D tree-based object space partition. The content of each partition is estimated by the number of supervoxels it contains, which is determined by counting the aggregation of data voxels in the volume, as in [6]. Partial images are generated from each partition and a binary-tree hierarchy is used to composite the images. The algorithm is implemented on an Intel Paragon.

2.3.2 Image-Parallel Methods

Schroeder and Stoll [89] implement an idea similar to the ray template method in [115] by taking advantage of the fast shift communication in SIMD platforms, such as the CM-2 and Princeton Engine. Each voxel is mapped to one processor, where ray samples are interpolated and shaded. Rays are generated by drawing discrete lines, which propagate from processor to processor in lock-step via direct neighbor communication.
Nieh and Levoy \cite{Nieh93} implement a parallel ray tracer on the Standard DASH. The screen is divided into regions which are in turn subdivided into tiles. Each node on the DASH is assigned a set of regions, and renders to these screen portions by casting rays. If a node completes its work early, it can steal a tile from other nodes with unfinished regions. The algorithm makes use of the fast hardware caching capability of DASH to avoid communication latency of accessing the remote data. Alternatively, a ray caster on Fujitsu AP1000, developed by Corrie and Mackerras \cite{Corrie92}, uses a worker-farm paradigm to achieve dynamic load balancing. A master-slave model is used to dynamically distribute tasks from a centralized control processor to all other rendering processors, which can notify the control processor if its workload exceeds its capacity (by a rendering time threshold.) The task partition size is adaptively adjusted to try to achieve a more balanced load. Palmer, et. al. \cite{Palmer93} apply a dynamic k-D tree screen partition which can be adjusted in a frame-to-frame basis. They use the \textit{first moment of workload distribution} to estimate the potential workload in each screen partition, and adjust the position of the cut lines based on feedback from the previous frame. The algorithm is implemented on the SGI PowerChallenge Array, with the volume replicated on each SGI PowerChallenge node.

Law and Yagel \cite{Law92b, Law92a, Law92c} have developed a series of parallel ray casting algorithms on the Cray T3D that take advantage of various types of coherency. The CellFlow algorithm \cite{Law92b} exploits inter-frame coherency between slow changing frames when only new objects are transferred to the next few frames. The Active-Ray algorithm
[57], which implements a COMA (Cache-Only-Memory-Architecture) scheme and an efficient distributed directory-based cache coherency protocol, uses the spatial coherency to cache a stable subset of volumes for subsequent image frames. Object space coherency is exploited by the RayFront algorithm [56] which allows rays to traverse the object space coherently. Our work is mostly inspired by [56][57].

2.3.3 Hybrid Methods

Yoo, et. al. [121] develop a hybrid renderer utilizing the sort-middle architecture of the Pixel Planes 5 [34]. Volume slices are first distributed among the graphics processors, which transform and shade voxels, and then redistributed to the rendering processors, each responsible for rendering a portion of the screen. A rendering processor will receive voxels that map to its screen portion. The splatting technique is used to generate the final image. Since the order of the projected voxels is arbitrary, a special sorting step is required to perform correct compositing.

Montani, et. al. [72] implement a hybrid renderer on the nCube-2. Here, processors are divided into clusters, and the entire volume is replicated within each cluster. Also, each cluster is assigned a subset of screen scanlines. Partial images are computed within each cluster using the object-parallel segmented ray casting approach and then composite into the final image. No communication takes place across clusters.
Recently, Palmer, et. al. [77] investigate the effect of ray casting on a deep memory hierarchy. Their work is implemented on an SGI PowerChallenge Array, which contains a memory hierarchy of L1 cache, L2 cache, local main memory within a PowerChallenge node (each PowerChallenge node is itself a shared-memory multiprocessor), and distributed memory on other nodes. To fully exploit locality in all levels of the memory hierarchy, a hybrid approach is used to partition the tasks. The volume is partitioned among the PowerChallenge nodes, and then an object-parallel approach is employed to generate partial images from the partitions on each node. An image-parallel ray casting approach is used to generate a partial image at each node. Each processor is assigned a subsection of the screen to render, and volumes are stored in blocks that can best fit into the L1 and L2 cache. Blocks are processed one at a time. A reordering phase within each processor is required to perform correct compositing, which is similar to the method used in [121]. After all partial images are computed on each node, a one-level binary-swap [66] occurs at the node level to generate the final image. A dynamic load balancing scheme is also used to adaptively adjust the block migration and duplication in the case that volume replication is possible.
CHAPTER 3

Visibility Computation

Visibility computation is a classic problem in both computational geometry and computer graphics. The most common application of visibility computation in computer graphics is the determination of visible surfaces to correctly render. Commonly used visible surface determination algorithms include the Z-buffer algorithm, area-subdivision algorithms (such as Warnock's algorithm), list-priority algorithms (such as BSP trees), and scan-line algorithms [31]. These methods attempt to solve the exact visibility (although some methods may be bounded by the pixel resolution) from a given viewpoint.

On the other hand, computational geometry also provides several other ways to solve the same problem, such as the well-known aspect graph approach [80]. These solutions are usually more complicated, because they compute an exact solution for the global visibility problem between any two regions in space. While all methods are designed to be accurate, most are computationally expensive. For example, scan-conversion of all polygons to determine visibility, even with an efficient hardware Z-buffer implementation, is still very time-consuming, especially when the model is complex and many polygons are in fact invisible to the viewer.
In light of these accurate but costly algorithms, researchers have designed efficient conservative algorithms that possibly overestimate the visible objects/polygons in a scene. The idea is to use the conservative algorithms as a preprocess before the model is applied to an accurate visibility algorithm, in the hope that this extra (but fast) processing step can cull most polygons that are definitely invisible from the current viewpoint. The conservative approach guarantees that all polygons that pass the accurate visibility test also pass the conservative test, which results in no change to the final image. Our algorithm in Chapter 5 belongs to this class. We review algorithms in this class in the following sections by first introducing space subdivision, a technique commonly used in these algorithms, followed by presenting major contributions from both object-space methods and image-space methods.

3.1 Space Subdivision

Most visibility algorithms use space subdivision in one way or another. The object space is usually partitioned into a number of smaller regions, or cells, to allow for efficient local computation. In order to minimize the overestimation of visibility computation, an ideal space subdivision should guarantee that any two interior points in a cell are visible to each other, i.e. all occluders are placed at boundaries of a cell. However, this is not always possible for some subdivision schemes.

The simplest space subdivision scheme, Cartesian grid subdivision (Figure 3.1(a)), is easily expressed with 2D or 3D arrays, and provides random access to each cell
via a simple index calculation. No extra storage overhead is required, as for subdivision structures, and sight rays can be efficiently traversed from cell to cell. Its major disadvantage is the inability to adapt cell sizes to data distribution, which may result in over- or undershooting in particular areas.

Another popular space subdivision scheme is the \textit{quadtree/octree subdivision} (Figure 3.1(b))[87]. Here, perpendicular planes recursively subdivide space into quadrants/octants. The recursion continues until each cell has minimal size or data values are below a threshold. The scheme is hierarchical, so cell resolution is adjustable according to the data distribution, and a simple regular structure is maintained. All are achieved at the expense of memory overhead for the hierarchical structure and a slightly more complicated cell traversal than that used for the Cartesian grid.
Besides the regular grid-based subdivision techniques, other methods directly exploit the spatial characteristics of the dataset. One famous example is the binary space partition (BSP) tree (Figure 3.1(c)) [33]. Instead of using pre-determined parallelepiped, a BSP tree dynamically chooses opaque elements in the dataset (usually a large planar objects such as walls) for dividing boundaries. Each opaque element divides space into two halves, and then each half space is recursively subdivided further by other opaque elements. Cells in this method satisfy the ideal property because all opaque elements fall on cell boundaries. When all opaque boundary elements are axially aligned, the BSP tree becomes a k-D tree (Figure 3.1(d)) [98], where each cell compactly represents k half-open extents \([min_0, max_0), \ldots, [min_k, max_k, l)\). The assumption for the boundary orientation means the k-D tree is easier to implement and maintain than the general BSP tree. Although it is possible for a BSP/k-D tree subdivision to provide an elegant and simple tree representation for a subdivided space, its construction is usually more time consuming compared with previous schemes.

### 3.2 Object-Space Methods

Object-space methods conservatively estimate visibility information directly by querying object space. The standard view frustum clipping and back-face culling are examples of such approximations. Garlick, et. al. [36] apply an object-space octree subdivision to accelerate view frustum clipping. Here, objects are organized in an octree hierarchy such that cells outside the view frustum are removed quickly. Shade, et. al. [90] employ a BSP tree to cull objects and use images generated from the current frame as a
texture to display objects in the next several frames. They calculate the *angular discrepancy* error to decide when to re-evaluate the visibility information. Zhang and Koff [122] group surface normals into clusters and encode cluster information in each polygon using a two-byte *normal mask* in an accelerated back-face calculation algorithm. To determine back-faces, a *back-face mask*, which represents all back-facing normal clusters, is used to test against each polygon’s normal mask using a single logical *AND* operation.

Kumar, et. al. [50] develop a hierarchical back-face method which is claimed to achieve sub-linear performance. This method requires that the polygonal model be represented by a hierarchy of clusters based on the normals and the locations of polygons. For each polygon cluster, it defines space partitions such that all polygons in the cluster are either back-faces, front-faces, or mixed when a viewpoint is located in different partitioned regions. They adapt a viewpoint tracking system to incrementally update back-faces from previous frames. All the methods we mentioned estimate visibility by checking only the relationship between the viewpoint and each individual object, and do not consider inter-object occlusion. Therefore, their results may still contain a high overestimation of the actual visible set.

Cell-to-cell visibility is successfully used in architectural walkthrough systems [3][97][64]. These algorithms partition the input model into *cells*, which roughly correspond to rooms in a building. They compute a *potentially visible set* (PVS) [3] of objects for each cell by determining the visibility relationship between cells. Visibility information for a cell is simply defined as the union of all objects visible from any point in
the cell and is collected during preprocessing or on-the-fly. The first method to use cell-based PVS is the *stochastic ray casting* algorithm [3][4], which determines visible objects by random discrete sampling. Objects intersected by rays cast from a finite set of stochastic points on cell boundaries are added to the cell’s PVS. In general, an excessive number of rays are needed to correctly obtain a PVS within a certain error bound, and the *underestimation* problem exists because some visible objects may not be reported. Another method proposed in [3] is the *shadow volume* algorithm, which tries to establish a list of *invisible* objects from a given cell. The method treats each non-opaque area as light source, which casts light to surrounding opaque objects to create shadow volumes. Objects are considered invisible from any viewpoint within the originating light area if they are entirely covered by the shadow volume.

Teller [97] develops a cell-based PVS visibility algorithm using *portal stabbing*, which is designed specially for architectural models with major occluders that are planar and axially aligned. The algorithm partitions the model via a k-D tree, where each cell in the subdivision is surrounded by opaque boundaries with some non-opaque “openings”, called *portals*. Portals are traversed through from one cell to the next until they are fully obscured by opaque boundaries, and visited cells are considered visible from the originating cell. Although the author develops a linear programming method to solve the portal stabbing problem, the method, in general, requires a long computation time to solve all portal sequences. The method is later generalized to allow non-axial placement of
opaque occluders by using a regular BSP tree partition [101]. However, no performance
details are given for this extension.

Coorg and Teller [99][100] improve the shadow volume method with temporal and
spatial coherency. Both algorithms produce a conservative overestimation of the actual
visible set because only a single or a group of connected convex polyhedra are recognized
as occluders to generate shadow volumes. Visibility is determined at cell level in space
subdivision, which is an octree in [100] and a k-D tree in [99]. The decision to avoid
object-level visibility computation reduces the computation needed for the shadow
volumes. Occluders are chosen dynamically during run-time from a candidate set, which
was precomputed by an off-line screening algorithm using several criteria. In [100], a
relevant plane approach computes only the changes in visibility from frame to fame. This
idea is based on the observation that an occludee object changes its visibility state only
when the viewpoint moves across some particular “planes” formed by selective edges and
vertices on both the occluder and occludee objects. By carefully registering these visual
events to their relevant planes, the algorithm efficiently updates visibility information
from one location to another without having to recompute the entire visibility information.
Their later method [99] caches the occlusion relation (a list of supporting and separating
planes required for testing the shadow volume) on each visited k-D tree cell. Thus, the
algorithm is able to use information from the previous frame to decide whether the
occlusion relation needs to be recomputed.
Yagel and Ray [118] consider a general class of walkthrough problems where the virtual environment includes three types of cells: void, solid, and wall. A void contains no opaque data and allows an observer to reside in it. A connected set of void cells form a passage through which an observer travels in the virtual world. A solid is filled with imaginary opaque material, so no light penetrates it and no viewpoint may reside in it. A wall is a cell that contains a interface between opaque material and a passage. The algorithm performs a regular grid space subdivision and then uses simple configurations of solid cells to decide whether sight lines pass from one cell to another. The advantages of this algorithm are its simplicity and generality. However, it does require the model to possess dense solid area in order to efficiently find the blocking configuration.

3.3 Image-Space Methods

Unlike object-space methods, image-space methods decide visibility with object rasterization on the screen. The hierarchical Z-buffering method, proposed by Greene, et. al. [39], utilizes two hierarchical data structures: an object-space octree and an image-space Z-pyramid. The Z-pyramid is a quadtree hierarchy constructed using contents from the Z-buffer, where each pixel at a pyramid level records the farthest Z value from the four quadrant pixels at a lower level. Visibility queries are performed on the Z-pyramid, such that polygons are tested against Z values at different hierarchical levels to check if they are fully occluded by previous rendering results. Object-space coherency is achieved by first testing the visibility of the octree cells. The algorithm can also achieve temporal coherency by using the Z-pyramid values from the previous frame as the basis for the next
frame. If the change of the scene is small between consecutive frames, then most invisible objects are removed from the very beginning. The only problem is that most graphics hardware do not support the Z-pyramid operations, and a software simulation is very expensive.

Continuing from his previous effort, Greene later proposes a hierarchical polygon tiling method [38]. The new method, in general, is treated as a depth-priority Warnock algorithm with an image-space acceleration. The center piece of the algorithm is a coverage mask pyramid, where each coverage mask is an $N \times N$ array of pairs of bit fields, which indicate which area is covered (inside) and which area is vacant (outside.) The complement of the union of both fields defines the area that is active (intersected.) Using these bit fields and the logical bit-mask operations, the algorithm performs the Warnock-like area-subdivision visibility in a fast and hierarchical way. The object space is partitioned by a BSP tree so that objects are traversed in FTB order which allows for efficient culling of hidden objects. A similar approach is exploited by Zhang, et. al. in their hierarchical occlusion map approach [123]. They use an occlusion map (opacity) pyramid as a visibility tester, where only areas with opacity values less than a given threshold are considered for visibility testing. Their method supports fusion of small objects into larger occluders to improve culling results.

The portal stabbing approach developed in [97] has also been modified to take advantage of image-space acceleration. In [64][44], tracing of portal sequences are
performed in the image space using 2D screen projections. The algorithm projects vertices of each portal into screen space and defines a 2D bounding box of the projection to be a cull box. If the cull box has non-zero extent, the cell adjacent to the originating cell through the projected portal is considered visible. The bounding boxes of the objects resident in the visible cell are subsequently projected and tested against the cull box to resolve object visibility. As each successive portal is traversed, the aggregate cull box is intersected with more portal projections to reflect the change of portal passage. The idea of portal projection is further developed to allow the use of pre-rendered images as portal textures [2] to replace the complex geometry behind the portal. The texture images are pre-rendered from reference viewpoints, and are warped to display an approximated image from an arbitrary viewpoint [83].

The shadow volume method is combined with an image-space projection by Hudson, et. al. [46]. Their algorithm is similar to the object-space shadow volume algorithm except for two differences: first, the shadow is cast from the current viewpoint, instead of an area; second, they use 2D screen projections of the silhouettes of occluder and occludee to test visibility, instead of using supporting/separating planes as in [99][100]. The visibility computation is performed against a bounding volume hierarchy, which can be an octree, k-D tree, or any hierarchy with rectangular boxes.
CHAPTER 4

Rendering Large Datasets with Latency-Optimized
Distributed Ray Casting

Designing an efficient parallel rendering algorithm can be a challenging task since most parallel algorithms introduce overheads which are not present in their sequential counterparts. Typical overheads experienced by parallel algorithms include [19]:

- **Communications among processors**: which includes the *initialization overhead* (e.g., packing/unpacking messages, system buffer allocations, etc.), and *communication latency* (time taken by a message to travel from one processor to the other).

- **Additional or redundant computations**: which includes the *additional computations* necessary to update all auxiliary data structures, and *redundant computations* in rendering due to parallelization.

- **Additional memory requirements**: which includes memory for replicated data or auxiliary data structures.

- **Delays due to uneven workloads**.

Some parallel rendering algorithms may be more prone to certain types of overhead than others. For example, although object-parallel algorithms [66][88][111] may require fewer communications than the image-parallel algorithms [76][56][75], the former will experience more redundant computations than the latter. Similarly, algorithms implemented on
the shared-memory architectures [75][77] have less concerns of communication latency than those implemented on the distributed-memory architectures [57][92][113]. However, the distributed-memory algorithms can be easily ported to different platforms as they do not rely on special architectural support for memory management. Since the trade-off of overheads may affects the algorithm's performance in different applications, the algorithm design must take into account the characteristics of the intended applications in order to achieve the best results.

For parallel rendering of large datasets, we design our algorithm on a distributed-memory multiprocessor system using message-passing mechanism. The main reason for choosing such a design platform is to allows for a general portability of our algorithm across different systems. For example, we would like to be able to run our algorithm on the modern massively parallel processors (MPP), such as the Cray T3E, as well as on other low-cost parallel computing systems, such as a network of workstations (NOW) [28][37]. With message passing, our algorithm can achieve not only such portability, but also a great flexibility to schedule message sequences for optimal results in different environments.

Several issues stand out when we consider designing an efficient parallel rendering algorithm for large datasets. First of all, due to the vast amount of data, communication overhead will be the first performance bottleneck to overcome. Since data is distributed across many processors, failing to provide remote data in time will force the
computational tasks to wait, which would decrease the overall rendering efficiency. The desired characteristics of such an algorithm is therefore to minimize the number of communications (i.e. incurring minimal initialization overheads and reducing network congestion), and to achieve good latency hiding results. “Latency hiding” is a technique to reduce the “wait” time between computational tasks by overlapping communications with computations. A good latency hiding mechanism can virtually erase the trace of communication latency from the rendering time, which is crucial in dealing with large datasets.

The second important design issue is the high computational and storage requirements. Rendering large datasets, such as volumes, are both computational and memory intensive. The desired way to do so is to process only those data which will contribute to the final image (i.e. avoiding redundant computation), and to conserve system memory for the dataset. Most distributed-memory systems utilize additional memory on each processor to cache data fetched from remote processors. These additional memory, called cache buffers, allow a processor to keep local copies of remote data for repeated references. The more memory dedicated to the cache buffers, the fewer remote accesses to the same item. However, since most system memory will be used by the dataset, the designed algorithm must be able to optimize its operations with only a limited number of cache buffers. Without proper arrangement, excessive cache conflicts will happen when too many remote data compete for a small number of cache buffers. The worst case scenario is cache thrashing, which occurs when the cache size is too small to
hold a minimal working set, resulting in cache misses for every access attempt. Such a problem can greatly degrade the performance of a parallel algorithm.

Bearing the above design issues in mind, we propose our Anchored Ray Transport (ART) algorithm for rendering large datasets. The ART algorithm is a ray casting-based parallel rendering algorithm, which is implemented on a distributed-memory multiprocessor system using message-passing mechanism. The parallel ray casting approach guarantees no redundant rendering computations due to its image-parallel characteristic, and allows our algorithm to take full advantage of an image-order approach, such as adaptive sampling [8][60] and early ray termination [31]. Unlike other image-order methods [18][75], which may suffer from poor object-space coherency and high communication overheads, we develop a special mechanism to allow highly coherent data accesses in object space, and achieve good latency hiding results. As a result, no cache conflicts or cache thrashing will occur, and the communication overhead is kept at minimal. In addition, the most unique feature of our algorithm is that it can achieve all the above benefits with only a very small amount of cache memory (several hundred kilobytes per processor). The algorithm can also be extended to take advantage of the frame-to-frame cache coherence for rendering multiple, successive image frames.

This chapter is organized as follows. In the next section, a general overview of the algorithm is given, followed by detailed discussions for each major stage of the algorithm (Section 4.2 to Section 4.4). The cache management mechanism is presented in Section 4.5.
4.1 Overview of the Method

The ART algorithm is outlined in Figure 4.1. It consists of three stages: preprocessing, initialization, and rendering. The dataset and screen are pre-classified and spatially subdivided during the preprocessing stage. Images are rendered by casting rays from the screen pixels and sampling along the ray in object space, as in traditional ray-casting. However, instead of shooting and completing one ray at a time, all participating rays are initialized and propagated through object space simultaneously. The ordering for processing rays is determined by a front-to-back (FTB) traversal of the volume, which also decides cache replacement and communication scheduling for all waiting cell requests. Utilizing a FTB order means that our algorithm is able to traverse object space coherently, resulting in high cache coherence and no cache conflicts. In fact, our algorithm can assure that no cache thrashing will occur even with a very limited amount of cache memory. The overall result is a scalable parallel rendering algorithm which is both memory and communication efficient.
The Preprocessing stage:

Dataset is subdivided into cells and randomly distributed to processors.
Screen is subdivide into tiles and block-randomly distributed to processors.

ART algorithm executed on each processor:

/* Initialization Stage: */
Determine the front-to-back list (FTB_list) for all cells in the dataset.
for each ray R from the screen tiles do
    Add R to the ray_list of the first cell it intersects.
    if the cell data is not local then
        Add the cell to the fetch_list
Define anchor_cell to be the first cell with non-empty ray_list in FTB_list.

/* Rendering Stage: */
while anchor_cell is not NULL do
    if the anchor_cell data is available then
        Trace all rays in the ray_list until they are completed or stopped by missing cell data.
        Add the stopped rays to the ray_list of the missing cells.
        Add any new missing cells to the fetch_list.
        Find the next anchor_cell.
        Release all cache buffers allocated by cells between the old and new anchor_cell.
    else /* out-of-order processing */
        From FTB_list, pick a locally available cell with a non-empty ray_list.
        Process the rays on this cell just like an anchor cell,
        but do not move the anchor cell nor release cache buffer.
        Receive/send the pending cell requests in the fetch_list.

Figure 4.1 The Anchored Ray Transport (ART) algorithm.
Our algorithm does not assume any special characteristics of the underlying dataset. The only requirement for the dataset is that it is spatially divisible and ray castable. Therefore, our ART algorithm is generally applicable to various types of graphical data, such as polygons, implicit surfaces, triangle meshes, or volumes. In our implementation, however, we choose to implement the ART algorithm on volume datasets because (i) volume rendering has always been a visualization challenge due to the immensity of datasets, and (ii) volume datasets fit well in our cell partitioning scheme since voxels are axially aligned. In the following sections we discuss details of our algorithm design in the context of volumetric datasets.

4.2 Preprocessing Stage

Data partitioning is usually performed by parallel rendering algorithms in order to exploit potential parallelism and to achieve better load balancing. For volume rendering, there are several natural ways to partition a volume: slices, shafts, and cells, as shown in Figure 4.2. Slices are partitioned in only one dimension, while shafts and cells are partitioned in two and three dimensions, respectively. Although our algorithm does not require a specific type of partitioning, cell partitioning usually achieves a more uniform object space distribution, and the communication cost is less sensitive to view direction [74]. Each cell contains a small subset of voxels and serves as the smallest unit of data to be transferred between processors.
Cells are randomly distributed across processors to avoid explicit cell clustering or the "hot spot" effect. Each processor is assigned some home cells, for which the processor serves as a home processor. A home processor is responsible for providing permanent storage for its home cells and responding to the requests for its home cells from other processors. Empty cells (cells with only null voxels) are pre-classified and the cell numbers are pre-distributed to all processors. This information allows the rendering process to skip over the empty cells and avoids unnecessary cell communication. In fact, it is possible to further encode the cell volume to take advantage of existing ray-casting acceleration techniques, such as hierarchical spatial representation [61], proximity clouds [15] [124], and discrete ray tracing [116].

The screen space is tessellated into 2D tiles similar to the volume partitioning just described. Tiles are distributed block-randomly in order to ensure that the tiles are uniformly distributed to processors and to reduce the occurrence of explicit patterns.
Arranging the screen tiles in this way helps to achieve a good static load balancing for most usages. Although other advanced load balancing schemes, such as [92][43][75], could be adopted, the above static scheme has proven to serve well in most situations.

4.3 Initialization Stage

Each processor's local memory is divided into two parts: one part stores the home cells from the preprocessing stage, and the other serves as a cache pool. Home cell memory is the permanent storage space for the pre-distributed home cells, and it is used to serve local/remote references to these cells. The cache pool is organized as a list of cache buffers, each of which is capable of holding one copy of a data cell. The cache buffers provide a way to reduce the number of remote references by caching local copies of the remote cells. The cache buffers are allocated dynamically and cells stored in cache may be replaced on demand. The design and purpose of the cache pool resembles the hardware cache used in shared memory multiprocessors [75][77], but our cache management is much simpler since all buffers are read-only. The amount of memory needed by the home cells is equal to the total number of cells divided by the number of processors. The size of the cache buffer is user adjustable and typically is a fraction of the home cell memory.

Front-to-back (BTF) and back-to-front (BTF) orders are often used in rendering algorithms to allow for efficient in-order composition for intermediate results. The BTF order was first used by Frieder, et al. [32] to project slices of voxels directly onto the 2D screen, and later in splatting [106][107] and line bundles [21]. The FTB order is used by
many object-order algorithms, such as dynamic screen [85], shearing-based methods [41][51], and others [56][117]. In our algorithm, we define:

- **Front-to-back list (FTB list):** a list of all volume cells sorted in front-to-back order.

Before the rendering process begins, each processor first generates a FTB list according to the position and orientation of the screen. As an image-order method, the purpose of the FTB list in our algorithm is to rank all data cells in such a way that, when a ray passes through the object space, it will intersect with the cells in a monotonic rank order. For example, if a ray first enters cell $A$ and later intersects with cell $B$, the FTB list assures that $\text{rank}(A) < \text{rank}(B)$ is always true. The FTB list is view-dependent, and thus needs to be recomputed if the view parameter changes.

The ART algorithm uses a parallel projection to cast rays from screen tiles into object space. Since all rays are parallel, the same FTB list holds true for all rays from the same screen. All rays emanating from the local screen tiles are initialized and then their intersections with the first cells are calculated. If a ray does not intersect with any non-empty cell, it is simply given a background color and terminates. All other initialized rays are added to the ray lists of the cells that they enter. Each cell has a ray list, which is a linked-list structure to hold unfinished rays within the cell. It is akin to bucket sorting [17] all unfinished rays into “cell” buckets. A ray list allows the algorithm to quickly locate and process all rays in a cell once the remote cell becomes available. To facilitate the discussions of the algorithm in the following sections, we define the following terms:

- **Engaged cell:** a cell with a non-empty ray list.
• **Hot cell**: an engaged cell which is available in either the home cell memory or the cache pool.

• **Cold cell**: an engaged cell which is not present in a processor’s local memory.

• **Anchor cell**: the *current* lowest-ranked engaged cell in a processor’s FTB list. Each processor has its own anchor cell whose position in the FTB list depends on the rendering status.

The collection of cold cells represents a set of “must-have” cells for future computation. Requests for these cold cells are managed by a *fetch list* (discussed in Section 4.5). Figure 4.3 shows one possible FTB list for a 2D cell array and several engaged cells with their ray list contents.

### 4.4 Rendering Stage

Images are rendered by sampling the object space at regular intervals along the ray. Each ray propagates through the object space in a greedy manner: it will advance as far as it is able to, until the ray is finished or hits a missing cell on its path. A ray is considered *finished* if it exits the bounding box of the scene or its accumulated opacity exceeds a predetermined threshold. Once finished, rays will be permanently removed from the rendering process. Rays interrupted by missing cells are pushed onto the ray lists of the responsible cells, and requests for the missing cells are added to the fetch list. Rays will pass over empty cells since they have been pre-classified and made known to all processors.
Figure 4.3 Initialization of cells and rays. (a) The FTB list order was calculated for all cells with respect to the current view direction. All rays are initialized and "queued" at the first cell they enter. Notice that if ray c advanced out of the volume, it would intersect with cells 3, 6, 9, 14, 19, 25, 29, 33 and 35 in an increasing rank order. (b) An illustration of the ray list structure with cells 1-4 and 35-36 shown with their initial contents. In this case, cells 1-4 are all "engaged."

The ART algorithm gets its name from the special way it handles the unfinished rays, via stepping on the anchor cell. The anchor cell is chosen because it possesses two important properties:

- Any cells ranked lower than the anchor cell are guaranteed to have flushed their ray lists.
- Rays currently queued at higher-ranked cells will not enter the anchor cell in the future.
Therefore, once we finish the ray list of the anchor cell, the cell is considered “done” for its lifetime; i.e., the cell will not be sampled again by any existing rays on the processor. If we strictly follow the FTB order to process each anchor cell’s ray list, we can efficiently dispose cells that are no longer needed by the algorithm. If any of these anchor cells reside in cache buffers, we can safely reclaim their cache memory since they will not contribute to the final image anymore. Cache buffers replaced in this way do not generate cache conflicts or thrashing. Our cache management system relies on this special processing order to achieve its efficiency.

The rendering process starts from the anchor cell on each processor and proceeds in the following way. If the current anchor cell is hot, all rays queued at it are advanced immediately in the way described before. Once all rays have left the cell, any cache buffer allocated to the anchor cell (if it is not a home cell) will be released immediately. If, on the other hand, the anchor cell is currently a cold cell, the algorithm will avoid waiting for the cell, by skipping over it to process other hot cells in the FTB list. Depending upon the network speed, it may take the processing of one or several hot cells before the anchor cell becomes hot, from which the normal processing order will resume. Such an out-of-order processing capability helps hide the communication latency for a cold anchor cell, but the breeching of the FTB order also keeps us from replacing any cache buffers held by these preempted hot cells. Even though we have cleared all rays pending on these cells, we cannot guarantee that rays emerging later will not enter these cells. Releasing their cache memory prematurely may incur extra remote accesses in later computation. However,
continually holding the cache buffers also presents a potential cache lock-up problem, which is defined as:

- **Cache lock-up problem**: when cache buffers were held by cells with empty ray lists, preventing new cell requests from being issued due to the lack of available cache buffers.

In the next section we will propose an approach to reduce, if not completely eliminate, this problem. Finally, after the current anchor cell has been completed, the processor goes on to find the next anchor cell candidate and continues ray processing from there, until no more engaged cells are left in the FTB list.

Each time after a hot cell has been processed, the processor also checks the fetch list for pending cell requests and uses a non-blocking receive command to check its communication buffer for messages. A mechanism similar to active messages [26] is used to handle different transactions. For example, if the message is a request for a cell, the corresponding cell information is immediately retrieved from the home cell memory and sent to the requesting processor via a non-blocking send. If the message contains cell information, it is read from the message buffer and inserted into the processor's local cache buffer. The communication routine interleaves the handling of message sending and receiving to avoid deadlock due to the filling up of communication buffers.

### 4.5 Cache Management

The design of the cache management system needs to address two main issues: how to request the remote cells and how to manage cell buffers for them. As mentioned
before, cell requesting and caching is an important issue since, as an image-order approach, each ray advances at a different pace depending on its spatial location and the distribution of available cells. Without proper coordination, cell requests can easily overwhelm the limited cache memory, resulting in excess communication overhead and poor cache performance. Large datasets intensify this problem, making it more difficult for the cache memory to hold a stable working set. One solution is to simply increase the size of cache memory. Law and Yagel [57] report that for a COMA-based (Cache Only Memory Architecture) distributed ray casting algorithm, it would require 64MB of memory to render a 16MB volume on 16 processors without causing significant cache misses and thrashing. That is a 4-to-1 ratio of memory overhead for the dataset only, not counting memory used by other auxiliary data structures such as cell arrays or ray stores. The memory requirements would be even greater in our case, since the home cells are permanently stored on each processor, regardless of whether they are needed by the local processor. This high memory overhead suggests that we need to carefully tailor the cache system in order to efficiently handle larger sized datasets.

In the previous section, we have seen that the rendering order of the ART algorithm formulates an efficient way to reuse cache buffers without causing cache conflicts or thrashing. The faster the cache can be readily reused, the better the algorithm can hide the communication latency. Since the availability of cache buffers depends on how smoothly the anchor cell order is followed, we need to make sure that most anchor cells, if not all, are available when the rendering process needs them. Therefore, the main
objective of our cache design is to streamline the supply of anchor cells. Additionally, we want to keep the following objectives in mind:

- Avoiding cache conflicts and thrashing.
- Minimizing the required cache size.
- Mitigating the cache lock-up problem.
- Hiding communication latency.
- Minimizing the number of communications.

4.5.1 An Initial Design and its Problems

Since the main objective is to “streamline the supply of anchor cells,” an initial design prefetches all cells in a strict FTB order using as much as the cache memory allows, as in [56]. Each processor tries to fill up its cache with the next $n$ remote cells following the current anchor cell, where $n$ is the number of cache buffers. Since anchor cells will appear in a FTB order, this approach hopes that if any of the next few anchor candidates are not home cells, they can be caught in these $n$ prefetched cells. Each time a cache buffer is released, the next remote cell on the line is immediately requested, filling the cache pool. Since our algorithm mostly follows a strict FTB order to process rays and release cache buffers, such a “prefetch-all” design takes advantage of this predetermined order to cache remote cells before they are actually explored, resulting in efficient latency hiding.
However, this simple design strategy has some major drawbacks. First, it does not take advantage of the information revealed by ray processing. One strength of the ray casting approach is that you can easily achieve early ray termination: if a ray is finished at one cell, there is no need to bring in more cells behind the current one for this ray. By requesting all remote cells in the FTB order, it neglects to distinguish which remote cell is actually needed. It ends up fetching all remote cells as long as some rays have not finished, even though those unfinished rays would probably need just a few more cells along the ray direction (as shown in Figure 4.4(a)). One side effect of this problem, is that sometimes it may hinder the algorithm's capability for latency hiding, as illustrated in Figure 4.4(b). The problem generally occurs when the next anchor cell is remote and is located near the end, or beyond the range, of the current prefetching list. When the processor finishes the current anchor cell and moves on to the next one, it would skip over many prefetched cells, sometimes even before their arrival to the processor, due to the lack of pending rays for these cells. This “cell skipping” movement may cause the processor to wait for the next anchor cell since cell requests made earlier still need to be completed and there may not be enough computation work to hide all the communication latency. Such a situation is not uncommon. When more rays are finished, remaining rays may scatter among different data cells which are distant from each other.

In addition to the above problems, the design will also cause excessive initialization overhead and heavy network traffic. For each cell being served/requested, the processor has to allocate a message buffer, pack/unpack the data cell to/from the
Figure 4.4 Problems in the “prefetch-all” design. (a) If the unfinished rays go all the way through the object space, remote cells in the three areas enclosed by bold lines will be fetched but never used. (b) When the current anchor cell (cell A) is being processed, the prefetched list covers the bold area in the next column, which includes remote cell B and some others. The lack of ray processing between A and B may not provide enough time to hide communication latency for cell B, given that other remote cells between A and B are being requested ahead of cell B at the same time. The problem is even more obvious between cell B and C.

message buffer, make system calls to provide the service, and consume some network bandwidth to transmit the message. As each processor indiscriminately prefetches all data cells, the associated system overhead and network traffic become enormous. Such a
"prefetch-all" design is good if most of the rays sample the entire volume. However, in real world applications, volumes tend to be either opaque or mixed with varying degrees of opacity based on classification [25][124], and rays rarely sample the entire volume. In the following section, we design our fetch list and cache system to avoid these problems and allow efficient cache management for all types of volumes.

### 4.5.2 The Fetch List Design with Near/Far Caches

Our algorithm utilizes a fetch list to achieve ordered accessing of remote cells with the help of ray processing information. To take advantage of the ray information, a cell request is made only after at least one ray has stalled on it. This approach differs from the "prefetch-all" design in that it will not avoid any first-time cache miss; instead, by working closely with the cache system, it helps to prevent cache conflicts and thrashing.

To allow efficient cache utilization and reduce the chance of cache lock-up, the fetch list is designed to fetch only those needed cells (which some rays are waiting for) in the FTB order. To achieve this objective, we use a priority queue [17] to implement the fetch list, with priority being each request's cell rank in the FTB list. Cell requests are added to the list in the order of their discovery, but are sent to the home processors according to their rank order. The fetch list is periodically checked after the completion of the ray list of a cell, and cell requests are issued depending on the availability of cache buffers. There will be no contention for the cache buffers since the number of the issued requests will never exceed the number of available buffers. In the case when the cache
memory is very limited, the priority queue will postpone the fetch of farther cells to allow candidates for the nearer anchor cells to be fetched in a more timely manner. Such an arrangement helps to prepare the anchor cells for the immediate future, therefore potentially speeding up the cache replacement process, which would eventually benefit other postponed requests.

Although the fetch list always tries to fulfill cell requests in FTB list order, it cannot guarantee that the requests will maintain a consistent FTB list order across separate iterations. Therefore, it is not unusual for cells requested in the earlier iterations to have higher FTB list ranks than those discovered in the later iterations. An extreme case is when the rays coming off the current anchor cell immediately engage its neighbor cell. Since this newly engaged cell has the lowest FTB list rank among all future cells, it will instantly become the new anchor cell and the next cell to be requested in the fetch list. As a result, prefetching is not applicable for this cell. Such a near-neighbor problem can be greatly reduced with a careful permutation of the FTB list order. Figure 4.5 shows three valid permutations of the same FTB list order for a given view direction. Permutation (a) is highly susceptible to the near-neighbor problem, as cells are traversed in consecutive rank order along the view direction. Permutations (b) and (c), on the other hand, greatly reduce the problem with their new cell ordering. Generally speaking, the best FTB list permutation to avoid the near-neighbor problem is the one which generates no consecutive cell ranks along any major directions of the view vector.
Figure 4.5 Determine the best permutation of the FTB list order to minimize the “near-neighbor” problem.

To work with the fetch list mechanism, the cache memory is logically divided into two parts: a near cache and a far cache. The near cache is a small amount of cache memory reserved for the potential cold cells within a certain vicinity of the current anchor cell. It is used to ensure that these cold cells, regardless of the timing of their appearance, have a guaranteed priority for available cache. This helps remedy the iteration-dependent cell order problem found in the fetch list. A lower-ranked cold cell discovered in the later iterations can now secure a buffer from the near cache if it is expected to contribute in the near future, i.e. if the cell is located close enough to the current anchor cell in the FTB list. The reserved cache allows the fetch list to immediately request the near-range cold cells without waiting for the release of other cache buffers. It provides more prefetching cushion to hide the communication latency for these cells, therefore reducing the chance of missing anchor cells in the future. The near cache also resolves a potential deadlock.
situation in the near-neighbor scenario: if the current anchor cell is a home cell and all cache buffers have been allocated, the next anchor cell will never be fetched since no buffers can be released unless the new anchor cell has moved past the cached cells. By reserving near cache for the near range cells, the algorithm guarantees a deadlock-free transition among anchor cells. The amount of the near cache can be very flexible, ranging from one cell (which is required to avoid deadlock) to several cells (depending on the overall amount of cache memory).

Besides the near cache, the rest of the cache memory constitutes the far cache and is used to store any other cold cells beyond the range of the near cache. Cells currently not qualified for the near cache will compete for the far cache, and in each iteration the fetch list tries to satisfy as many cell requests as possible in FTB order. While the near cache controls the availability of the anchor cell and its successors, the far cache is used as an "inventory" storage to prefill some needed cells in a long term basis. As the anchor cell advances in the FTB order, cells previously stored in the far cache will eventually become candidates for the near cache. Once entering the near cache range, these cache buffers (originally belonging to the far cache) will be switching their identities to become part of the near cache, allowing other cache buffers (originally belonging to the near cache) to be released to join the far cache, maintaining a dynamic balance between the two cache groups. Therefore, there is no special set-aside allocation for the near cache as any cache buffer can become either near or far during the course of the execution. Another major function for the far cache is to provide additional computational tasks. If the anchor cell
traversal stalls, cells stored in the far cache can provide extra work for an out-of-order execution, to fill up the computation pipeline. This is crucial when the algorithm is implemented on machines having a slow network, as it takes more time to complete a cell transfer in these environments.

Once allocated, a cache buffer can be replaced only after the anchor cell moves past it in the FTB order. Such a replacement policy poses no problem for the near cache, since the cached cells are usually very close to the anchor cell. For far cache, however, it may affect cache utilization since cells in the far cache are by definition farther away from the anchor cell, and thus will hold the cache buffer for a longer time before being replaced. A cache lock-up problem could occur if the far cache is not replaced fast enough and out-of-order executions arise frequently. To allow more flexible management for the far cache, while preventing unnecessary cache replacements, we define a preemptive index \( P_{Index} \), as below, to monitor workloads and the buffer allocation status in the far cache:

\[
P_{Index} = f_{ree}_{\text{far}} + \frac{h_{ot}_{\text{far}} + \omega \cdot f_{etching}_{\text{far}}}{a_{llocated}_{\text{far}}}
\]  \hspace{1cm} (4.1)

where:

- \( f_{ree}_{\text{far}} \) is the number of the cache buffers (each cache buffer is capable of holding one remote cell) currently available in the far cache.

- \( h_{ot}_{\text{far}} \) is the number of hot cells currently stored in the far cache.

- \( f_{etching}_{\text{far}} \) is the number of cache buffers allocated for those remote cells currently on the way to the far cache.

- \( a_{llocated}_{\text{far}} \) is the number of cache buffers currently being allocated in the far cache. \( a_{llocated}_{\text{far}} + f_{ree}_{\text{far}} \) is the total number of cache buffers in the far cache.

60
- $\omega$ is a parameter between zero and one, which adjusts the "early contribution" from those cells currently being fetched. Using $\omega$, cells currently being fetched are treated as "partial" hot cells in the workloads estimation. Once a fetched cell arrives in the cache buffer, the remaining contribution $(1 - \omega)$ of the cell will be added to the equation.

$PIndex$ decides when the preemptive cache replacement policy will be in effect for the far cache. Our objective is to delay any preemptive replacement until the $PIndex$ drops below a given threshold (a value between zero and one). By definition in Equation 4.1, $PIndex$ will have the following properties:

- If there still exist some available cache buffers in the far cache (i.e. $free_{far} > 0$), $PIndex$ will be greater than one. Therefore, the preemptive cache replacement policy will not be activated (since we try to keep all allocated buffers as long as possible to avoid multiple accesses of the same cell later).

- Since the allocated cache buffers ($allocated_{far}$) also include those buffers held by cells with empty ray lists, the inequality $allocated_{far} \geq hot_{far} + fetching_{far}$ is always true.

- If the far cache is exhausted (i.e. $free_{far} = 0$), $PIndex$ will become the hot cell ratio in the far cache and its value will always be between zero and one.

Once enabled, the preemptive cache replacement will allow new cell requests to replace those far cached cells whose ray lists have been processed by the out-of-order execution. We apply a heuristic policy which chooses eligible cache buffers in a reverse FTB order for the preemptive replacement. Such an order is chosen because the farther the cell, the less likely it will be reused by different groups of rays (due to early ray termination and opacity thresholding), therefore reducing the chance of fetching the same
cell again. Other approaches, such as a random order or the least-recently-used (LRU) order, can also be used.

The entire cache management system is designed to support the anchor cell-oriented rendering approach used in the ART algorithm. It achieves its primary goal—maintaining a smooth supply of anchor cells—by using the fetch list and the near cache. Cache conflicts and thrashing are totally eliminated as cell requests are carefully scheduled to avoid any cache buffer contention. This comes without sacrificing the performance, as the utilization of each cached cell is maximized by allowing all rays of the cell to advance before it is replaced by other cells. Therefore, most of the cells (except those being preempted in the far cache) are being retrieved from the remote processors exactly once. Moreover, since only the needed cells are fetched, network congestion is reduced, as is the initialization overhead for communications. The communication latency is mostly hidden due to overlapping the cell fetching with the computation pipeline, and the occasional out-of-order cell processing.

4.6 Multiframe Rendering

In previous sections, we have shown a parallel ray-casting algorithm which is both cache and communication efficient. The guaranteed thrashless behavior and minimal communication make our algorithm suitable for parallel rendering large datasets. Rendering animations of a large dataset presents a similar and more challenging problem. Here the excessive amount of work comes not only from the dataset itself, but also from
the multiple animation frames. Without taking advantage of the possible frame-to-frame cache coherence, direct application of the ART algorithm to each individual frame would result in a proportional increase in the communication cost and computation time. We will extend our ART algorithm to address the multiframe issue in this section.

### 4.6.1 Basic Multiframe Capability

The required mechanism for our first multiframe extension is already built in the algorithm design and can be exploited without any additional cost. The algorithm's anchor cell-based rendering approach can correctly handle any rays that follow the same FTB order. Since the FTB order is only determined by view direction, frames close to each other or sharing a similar direction can be initialized and rendered together by the algorithm. If the dataset is static (i.e. without moving or animating objects), the above multiframe approach can be used. This idea is shown using a 2D example in Figure 4.6. Figure 4.6(a) shows several image frames and their view directions from a single rotation about the object. Figure 4.6(b) shows the classification of image frames based on their view vectors. Here, view vectors are used as the image frame's “directional signatures”, which can be grouped by four quadrants. All frames belonging to the same quadrant in Figure 4.6(b) will be able to share a common FTB order. In this case, frame $a$, $b$, and $c$ can be processed simultaneously with the FTB order shown in Figure 4.6(a). The advantage of rendering multiple images simultaneously is that all cache contents will be shared across several frames, and the frame-to-frame cache coherence can be explored to avoid multiple fetches of the same remote cells. In this way our algorithm is able to amortize the parallel
rendering overhead, like the cost for building the FTB list and transferring the cells, in each group of frames and improve the original algorithm’s efficiency under the multiframe situation.

Figure 4.6 A 2D example for the basic multiframe rendering. (a) Multiple frames are positioned around the dataset with their own view directions. (b) Frames with view directions within the same quadrant can be rendered together by the basic multiframe rendering extension.

The above multiframe extension is especially suitable for animations generated from a continuously moving viewpoint around a static volume. The smoothly changing images in consecutive frames make them a perfect target for exploring the frame-to-frame cache coherence. Nevertheless, such a multiframe capability only exists for frames with view vectors confined to the same octant in 3D space. Across different octants, reinitialization of the FTB list and additional cell communication is mandatory. Therefore,
this multiframe extension is best used in animation with only moderate (less than 90 degrees of rotation and within the octant boundary) changes in the view direction. Besides the view restrictions, all multiframe extensions also require a significant amount of memory to store the unfinished rays. Each unfinished ray would not only need to save the image color, but also all intermediate information, such as its original screen coordinates (two short integers), opacity value, parametric value for the ray (one integer), and a ray pointer (for forming the ray list; one integer). To conserve the ray memory, we do not store the world coordinates of each ray. Instead, we save the world coordinates of the origin and the base vectors of each image frame, and construct the world coordinates of each ray from its screen coordinates on the fly. We also use a fixed point representation (two bytes) to store the image color and opacity, resulting in a total of 20 bytes needed per ray. Since the algorithm requires all rays to be instantiated during the initialization stage, the maximum number of frames that can be rendered together will greatly depend on the image resolution and the amount of available system memory.

4.6.2 Multiframe from 180-degree View Direction

While the basic multiframe extension is straightforward and easy to implement, the restriction on view direction limits its practical application. The restriction is a reflection of the way the FTB order is calculated, where the order is determined by the signature of the view direction. When the view direction moves across the octant boundary, the same order no longer holds. To overcome such a limitation, Law [54] proposes a ZZ-buffering method to expand the range for different view directions to 180
degrees. This effectively merges two octants into one multiframe zone, leaving four zones in 3D space. In order to allow rays to be traced from both octants, the ZZ-buffering method applies segmented ray tracing approach [66] to trace partial ray segments from the second octant. A special mechanism is used to perform splitting/merging of ray segments, and requires extra memory to store (at most) two partial segments per ray. Considering the heavy demand for ray memory in the multiframe processing, such an approach may actually further limit the number of frames processed. Law also proposed an approach combining FTB, BTF and ZZ-buffering to accommodate an all-around 360 degree viewing [54], but it was achieved at the expense of disabling the early ray termination and adding even more memory requirement. These difficulties prompted us to look for a better solution to expand the multiframe capability.

Our solution to expand the view range to 180 degrees is based on using dimension-reduced FTB order. Figure 4.7 illustrates the idea for a 2D example. In Figure 4.7(a), cells are arranged in a 2D FTB order from left to right horizontally and bottom to top vertically. Traversing cells in this order will advance ray A without any problem, but will possibly cause ray B to stall with a broken cell path. The problem is due to the fact that when the algorithm moves the anchor to cell 5 and processes the ray list, ray B will enter some remote cells (cell 3 and 4) that are ranked lower than the current anchor cell. Since these cells have been visited before and are considered 'done' for all computation, they no longer exist in local memory and will not be fetched again, causing ray B to wait for them indefinitely. The main reason for the above problem is that ray B only conforms to the
FTB order vertically, but not horizontally. Therefore, in order to process both rays simultaneously, we need to “reduce” the horizontal ordering and only arrange the cells in a 1D FTB order vertically. Figure 4.7(b) shows a shaft-like ordering which is compatible with the direction of both rays. The cost of applying such a dimension-reduced FTB order is that now we need to keep the entire anchor *shaft*, instead of only one anchor *cell*, available in cache memory. However, further investigation of the problem reveals a more efficient way to implement the dimension-reduced approach. Figure 4.7(c) shows that, in order for ray $B$ to pass the first shaft (cell 1 to 5), it requires holding only two finished cells (cell 3 and 4) in addition to the anchor cell (cell 5). The requirement for ray $B$ to pass other shafts (cell 8 to 12, cell 15 to 19, etc.) is similar, which would be holding either two or three finished cells (not including the anchor cell) in each shaft, depending on the ray’s entry point to each shaft. We call these extra cell buffers that should be withheld before an anchor cell as *holding buffers*, and their size depends on the angle that the incoming ray takes to approach the reduced dimension. The worst case scenario is when the direction of ray $B$ is horizontal, where we need to reserve one full line of cells as holding buffers.

Given a volume of $n^3$ cells and $P$ processors (each holds $n^3/P$ home cells), the reduced dimension approach would require at most $n$ cells per processor for holding buffers, which is only $(100 \cdot P)/n^3$ percent of the memory to store the home cells. For a volume of $2^5 \times 2^5 \times 2^5$ cells rendered on 32 processors, this translates to only 3% of cache overhead per processor to implement the reduced dimension approach.
Figure 4.7 Achieve the 180-degree multiframe rendering with dimension-reduced FTB order. (a) The basic multiframe extension. (b) Dimension-reduced method. (c) Using holding buffers to implement the dimension-reduced method. The solid cells indicate the required buffer size for the "anchor" object.

The integration of the 180-degree multiframe extension requires minimal changes to the original ART algorithm. First of all, view directions from all participating frames are used to decide which dimension to reduce. The reduced dimension corresponds to the major direction (X, Y, or Z) along which the view vectors show different signatures. Once it is decided, projection from view vectors helps to choose a permutation of FTB order that requires the least number of holding buffers (the algorithm is shown in Figure 4.8). Since our algorithm has already reserved some buffers for near cache, we simply extend near cache to cover these holding buffers. All these modifications are done in the initialization stage with little or no extra overhead. During the rendering stage, everything works exactly as the original algorithm. The only difference is the way the extended near cache works: it will keep cells stored in the holding buffers in place for the current anchor.
Algorithm to decide the optimal FTB order for 180-degree multiframe extension:
Assume all view vectors are in Multiframe Zone I (±,+,+) (refer to Figure 4.11); other zones can
be similarly applied.
Assume there are \( c \) cells in the volume.
Let \( FTB\_list(x, y, z) \) be the FTB_list rank for cell at \((x, y, z)\), where \( 0 \leq x, y, z \leq C - 1 \).

1. Project each view vector onto the four quadrants on XY-plane and XZ-plane, namely,
   \( \theta_1 = XZ(+,+), \theta_2 = XZ(-,+), \theta_3 = XY(+,+), \theta_4 = XY(-,+). \)
2. Intersect each projected vector with \( Y=1 \) (on XY-plane) or \( Z=1 \) (on XZ-plane).
3. Find the maximal intersected values (in absolute value) along the X-axis on the four
   quadrants (\( \theta_1 \) to \( \theta_4 \)), name them \( \mu_{\theta_1} \) to \( \mu_{\theta_4} \), respectively.

   Let \( \mu_0 \) be the minimal among \( \mu_{\theta_1} \) to \( \mu_{\theta_4} \).

4. The number of holding buffers will be \( \text{ceiling}(\mu_0) \).
5. The FTB order is determined by the following rules:
   - \( \text{if (} \theta \text{ is } XZ(+,+)) \text{ then define } FTB\_list(x, y, z) = y \cdot C^2 + z \cdot C + (C - x) \)
   - \( \text{else if (} \theta \text{ is } XZ(-,+)) \text{ then define } FTB\_list(x, y, z) = y \cdot C^2 + z \cdot C + x \)
   - \( \text{else if (} \theta \text{ is } XY(+,+)) \text{ then define } FTB\_list(x, y, z) = z \cdot C^2 + y \cdot C + (C - x) \)
   - \( \text{else if (} \theta \text{ is } XY(-,+)) \text{ then define } FTB\_list(x, y, z) = z \cdot C^2 + y \cdot C + x \)

Example:
The following illustration shows that four view vectors are projected onto XZ and XY planes to
generate eight projected vectors. Quadrant maximal values on X-axis are labeled \( A \) to \( D \). Since \( C \)
is the minimum among those values, it will require only two holding buffers by arrange the cells
front-to-back in \([+X,+Z,+Y]\) order.

![Figure 4.8 Determine the FTB permutation and the number of holding cells.](image)

cell, and allows any cell within the near cache range to be requested immediately, even if
it ranks lower than the current anchor cell.
4.6.3 Multiframe from 360-degree View Direction

The ultimate goal for the multiframe design is to allow the algorithm to include frames from any view direction in one rendering pass. The 180-degree multiframe expansion is useful when the viewpoint is mostly turning or moving along the reduced dimension. If the change of view direction does not coincide with any major axes, or a more sophisticated maneuvering is required, the resulting frames would inevitably fall into different multiframe zones and require multiple passes. One possible solution is to apply the reduction technique to more than one dimension, but it would require a significant increase in the amount of cache memory, especially when more processors are used. For example, for a volume of \(2^5 \times 2^5 \times 2^5\) cells it would require a near cache of at least \(2^{10}\) cells to apply a 2D reduction to the FTB order. If the algorithm is run on four processors, each processor would hold one-fourth of the entire volume as home cells, which is \(2^{13}\) cells per processor. Therefore, the per-processor cache memory overhead is \(12.5\% \ (2^{10} + 2^{13})\) of the data size. However, the memory overhead would become 100% when 32 processors are used! Such a high memory requirement prevents the dimension reduction technique from being efficiently used in the 2D case.

To expand the view range beyond 180 degrees, we propose a new spiral order to work in conjunction with the FTB order. A spiral order is a two dimensional cell order which arranges cells from the edge toward the center of a 2D cell plane in a coil shape (shown in Figure 4.9(a)). If we traverse the cells (and trace the ray list of each cell) according to the spiral order shown in Figure 4.9(b), rays from any view direction would
Figure 4.9 The spiral order: (a) cell ranks assignment (b) the effect of regular traversal (illustrated by the counter-clockwise coil) (c) the effect of reverse traversal.

advance toward the inner coils simultaneously. When the traversal reaches the center cell, all unfinished rays have at least traveled across their midpoints toward the other half of the plane. A reverse traversal of the same order will push rays to complete their journey (Figure 4.9(c)). Therefore, by applying two passes of traversal (one forward spiral and one reverse spiral), all rays entering a 2D plane can be processed simultaneously. We use this two-pass technique to build a 360-degree multiframe rendering approach for a 3D volume.

Volume cells are ordered such that within each 2D slice they follow the spiral order, and the slices are arranged in a FTB order. Rays entering the same slice are processed together by the two-pass spiral-traversal method, and the inter-slice ray propagation is handled by the FTB order. The most important aspect of this approach is that the two-pass slice processing can be done without the cost of a high cache memory overhead. It is achieved by applying the 180-degree technique to handle each segment of cells on a 2D spiral path.
independently. A fixed number of holding buffers forms a drifting "cache window," moving along with anchor cells as they are visited in the spiral order, to allow rays from any direction on the 2D plane to be advanced without causing the broken cell path problem (Figure 4.10). By adopting this memory efficient approach to implement the spiral order traversal, we will allow frames with more diversifying view directions to be included in one multiframe rendering session. Figure 4.11 illustrates examples of different multiframe zones allowed in the basic 90-degree, 180-degree, and 360-degree multiframe extensions. Note that except for the 90-degree extension, there are other ways to partition 3D space for the other two approaches, depending on which dimensions are reduced. The direction signature for each multiframe zone is also shown in Figure 4.11, with the reduced dimension indicated by "±."

Figure 4.10 Uses the 180-degree technique to implement the spiral order traversal. Home cells are filled with stripe and the cache window is filled with solid shade. The above figures show three snapshots of the spiral order traversal with a sliding cache window.
Figure 4.11 Different multiframe zones for (from left to right) 90-degree, 180-degree, and 360-degree multiframe extensions.

The spiral/FTB hybrid approach possesses some different characteristics than the regular FTB approach. The first notable difference is its requirement for a two-pass traversal for each slice, rather than the one-pass traversal needed by a FTB approach. The two-pass traversal will visit each cell twice, which is comparable to applying the 180-degree multiframe rendering twice to cover all frames on the same half space. However, the integrated two-pass approach allows rays from all directions to advance simultaneously, potentially increasing the chance for rays from different directions to share a common cached cell. For example, in Figure 4.10 two rays from opposite directions use the same cached instance of cell 12 to move forward. If the 180-degree multiframe rendering approach is used, cell 12 would have been fetched twice since those two rays may belong to different multiframe zones. Therefore, the 360-degree extension is guaranteed to be at least as efficient as the 180-degree version when the view directions spread more than 180 degrees. The spiral/FTB hybrid approach also supports the early ray...
termination, an important feature excluded by Law's FTB/BTF/ZZ-buffering method. Although Law's method visits each cell only once, it would fetch all remote cells compared to at most two fetches for each needed cell in our hybrid approach. Our approach is advantageous for practical applications since most renderings do not access all voxels in a volume.

The implementation of the spiral order requires two fetch lists: a regular fetch list, which arranges requests in an increasing cell order, and a reverse fetch list, which arranges requests in a decreasing cell order. During the first pass, any missing cell with a rank within or greater than the near cache range (which includes the holding buffers) will be added to the regular fetch list. The regular fetch list schedules these cell requests to help rays move toward the inner coils of the spiral, or in general, toward the higher ranked cells. Other missing cells will be held in the reverse fetch list, as requests for these cells always come from rays already on their way out of the spiral. When the second pass begins, everything is done in a reverse way: cells are visited in a decreasing order and the reverse fetch list becomes active to schedule its cell requests. Missing cells with ranks greater than the reversed near cache range will be pushed into the regular fetch list, as these cells are visited by rays penetrating into the higher-ranked slices. Any other cell requests will be scheduled in the reverse fetch list to help rays moving outward to leave the spiral. The algorithm performs such a two-pass processing for each slice in FTB order, allowing unfinished rays to be propagated through different slices in a correct order. Compared to all previous multiframe extensions, the only extra memory overhead for the
hybrid approach is the need for a second fetch list, which is almost negligible under the multiframe situation.

The addition of the multiframe capability not only allows the ART algorithm to render large datasets in multiframe animation as efficiently as in a single image, it also enables the algorithm to handle the *perspective projection* as well. So far we have been limited to parallel projection only since our algorithm uses a static FTB order to handle all rays. With perspective projection, each ray from the same screen will have a distinctive ray direction and possibly a different FTB order. While the original ART algorithm would need multiple sessions to render these perspective rays, the 360-degree multiframe technique can simultaneously process these rays as if they are coming from different parallel-projected frames. The only restriction is that the movement of the viewpoint may not have as much freedom as that in the parallel projection, since a single perspective projected image now covers a subset of view directions in the multiframe zone.

### 4.7 Test Datasets

Table 4.1 lists the volume datasets used in this section as shown in Figure 4.12. Sod128 is the electron density volume of superoxide dismutase (SOD) and Head256 is a MRI head scan dataset, both taken from the UNC Chapel Hill volume repository. Cube datasets are simple volumes with one small cube laying on top of another larger cube, and come in two different flavors: Cube128 is set to be fully opaque and Cube256 to highly transparent. Mix256 and Mix512 datasets are volumes generated from the following
<table>
<thead>
<tr>
<th>Volume</th>
<th>Volume Size</th>
<th>Default Cell Size</th>
<th>Screen Size</th>
<th>Opacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sod128</td>
<td>$128^3$</td>
<td>$8^3$</td>
<td>$128^2$</td>
<td>transparent (0.01)</td>
</tr>
<tr>
<td>Cube128</td>
<td>$128^3$</td>
<td>$16^3$</td>
<td>$128^2$</td>
<td>fully opaque (1.0)</td>
</tr>
<tr>
<td>Head256</td>
<td>$256^3$</td>
<td>$16^3$</td>
<td>$256^2$</td>
<td>fully opaque (1.0)</td>
</tr>
<tr>
<td>Cube256</td>
<td>$256^3$</td>
<td>$16^3$</td>
<td>$256^2$</td>
<td>fully transparent (0.005)</td>
</tr>
<tr>
<td>Mix256</td>
<td>$256^3$</td>
<td>$16^3$</td>
<td>$256^2$</td>
<td>mixed (0.003 to 0.3)</td>
</tr>
<tr>
<td>Mix512</td>
<td>$512^3$</td>
<td>$32^3$</td>
<td>$512^2$</td>
<td>mixed (0.002 to 0.2)</td>
</tr>
</tbody>
</table>

Table 4.1 Volume datasets

geometric objects: an outer, almost fully transparent sphere (opacity = 0.002) surrounding an inner, moderately transparent sphere (opacity = 0.02), within which several other solid nuclei (opacity = 0.2) are symmetrically distributed (Figure 4.12(d)). The Mix datasets are created to simulate volumes intermixing materials of different opacity, such as soft tissues, fat, bones etc. found in a medical volume. Figure 4.12 shows the rendered images of the test volumes. All images are rendered with one ray per pixel, and samples are taken at unit intervals with no interpolation. Any ray with an accumulated sampled opacity of 0.9 or greater is regarded as finished. Illumination at each sample point is calculated by using the first-order gradient normal (central difference) and the Phong illumination model with three light sources.
Figure 4.12 Images of the test volumes: (a) Sod (b) Cube (c) Head (d) Mix
4.8 Experimental Results on the Cray T3E

The algorithm was first implemented on the Cray T3E at the Ohio Supercomputing Center (OSC). The T3E at OSC is configured with 128 processing elements (PEs) interconnected via a fast 3D-torus interconnection network. Each PE consists of a 300 MHz DEC 21164 super-scalar RISC processor and 128 Mbytes local memory, with a peak memory bandwidth at 1.2 Gbytes/sec. The high performance 3D-torus network allows efficient message routing between processors and is capable of a peak throughput of 480 Mbytes/sec in each direction. The Cray T3E PVM (Parallel Virtual Machine) library [7] is used to handle all message communications in our algorithm.

4.8.1 Speedup and Scalability

<table>
<thead>
<tr>
<th># processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sod128</td>
<td>0.587</td>
<td>0.320</td>
<td>0.180</td>
<td>0.097</td>
<td>0.052</td>
<td>0.030</td>
<td>0.017</td>
<td>0.013</td>
</tr>
<tr>
<td>Cube128</td>
<td>0.290</td>
<td>0.150</td>
<td>0.084</td>
<td>0.050</td>
<td>0.034</td>
<td>0.021</td>
<td>0.015</td>
<td>0.012</td>
</tr>
<tr>
<td>Head256</td>
<td>2.757</td>
<td>1.477</td>
<td>0.842</td>
<td>0.487</td>
<td>0.289</td>
<td>0.163</td>
<td>0.085</td>
<td>0.047</td>
</tr>
<tr>
<td>Cube256</td>
<td>6.480</td>
<td>3.389</td>
<td>1.716</td>
<td>0.935</td>
<td>0.560</td>
<td>0.323</td>
<td>0.166</td>
<td>0.094</td>
</tr>
<tr>
<td>Mix256</td>
<td>14.384</td>
<td>7.341</td>
<td>3.757</td>
<td>1.981</td>
<td>1.049</td>
<td>0.566</td>
<td>0.307</td>
<td>0.180</td>
</tr>
<tr>
<td>Mix512</td>
<td>99.872</td>
<td>50.387</td>
<td>25.411</td>
<td>13.063</td>
<td>6.743</td>
<td>3.534</td>
<td>1.811</td>
<td>0.967</td>
</tr>
</tbody>
</table>

Table 4.2 Average frame time (in seconds) for rendering one frame per rendering session

To test the parallel performance of the algorithm, we render five images from an orbiting viewpoint spanning 60 degrees around the center of each dataset. The rendering times are then averaged and listed in the following tables. Due to the memory limitation, the single-processor rendering time for Mix512 is collected by performing the
computation on one processor with the dataset distributed across two processors. Table 4.2 shows the average frame time when the basic algorithm is used to render each individual image independently. The tables show that the algorithm is capable of generating images at a very high frame rate: for example, 21 frames per second for Head256 and one frames per second for Mix512 on 128 processors. Figure 4.13 shows the speedup and scalability (defined as \( \frac{\text{Speedup}_{\text{actual}}}{\text{Speedup}_{\text{ideal}}} \)) for the single-frame rendering results. The single-frame algorithm demonstrates good performance and scalability on large or non-opaque volumes, reaching 80% of efficiency (speedup 103) for Mix512 on 128 processors and 73% (speedup 46.8) for Mix256 on 64 processors. Generally speaking, our algorithm relies on intensive computational tasks, which usually could be found in large or non-opaque volumes, to successfully reduce the impact of communication latency. If the rendering process can not generate enough computations to overlap with communications, performance will start to degrade due to the presence of communication overheads. For example, for the Sod128 dataset, the single-frame algorithm can manage only 61% (speedup 19.5) on 32 processors and 54% (speedup 34.5) on 64 processors. Opaque volumes (such as Head256) share a similar problem as rays rarely sample more than a few voxels before it exceeds the preset threshold. In these situations, we need to increase the computational tasks for the small or opaque volumes. In the following experiment we will use the 90-degree multiframe rendering to help these small volumes achieve the desired balance between computations and communications.
Figure 4.13 Speedup and scalability for rendering a single frame.
Table 4.3 reports the timings when all five images are rendered simultaneously by using the 90-degree multiframe algorithm, and Figure 4.14 shows the speedup and scalability. While the performance for Mix512 and Mix256 remain solid, they do not improve much over their single-frame counterparts, with increases in performance of only 5.8% and 10.5% on 128 processors, respectively. However, with the five-frame rendering, smaller and opaque volumes pick up the extra computation loads to improve their performance. Figure 4.14 shows that the efficiency for Sod128 improves to 76% (speedup 24.4) on 32 processors and 65% (speedup 41.9) on 64 processors after rendering five frames simultaneously. Comparing to the single-frame rendering, the performance is improved by 20% and 17.6%, respectively. We expect to see more improvement in performance when more frames are rendered simultaneously.

4.8.2 Cell Size

The cell size determines the smallest unit of volume that is transferred among the processors. Since a remote cell will be requested as long as some rays are waiting for it,
Figure 4.14 Speedup and scalability for 5-frame, 90-degree multiframe rendering.
Figure 4.15 Cell size analysis for (a) Mix256 (b) Mix512 (c) Cube128 (d) Cube256. Each data series represents the rendering time for a particular cell size. The cell size of $64^3$ voxels is applicable only to Mix512 since smaller volumes cannot generate enough cells for all processors. Rendering time is normalized to the time for the cell size of $32^3$ voxels.

The size of a cell may determine the effectiveness of interprocessor communication. Generally speaking, a smaller cell size allows a more accurate amount of data to be transferred, but may result in a higher number of messages. It also stalls rays more frequently, which adds more runtime overhead in restarting them, but allows a finer grain of parallelism. We tested our algorithm on several volumes for different cell sizes. Our
results show that for Cray T3E, the optimal cell size is $32^3$ voxels, regardless of the number of processors or volume used (Figure 4.15). This consistency of the optimal cell size allows us to prestore only one version of the subdivided volume, saving both time and disk storage to preserve and load volumes of different cell sizes.

Although $32^3$ voxels is the preferred choice, in some cases we may want to use a smaller cell size. As we will show in the next section, the effectiveness of cache memory is measured by how many cells it can provide. That is, in order to achieve the best cache performance, a certain number of cache cells need to be available on each processor. However, at a cell size of $32^3$ voxels, each processor may host only a few home cells when a large number of processors are involved. Therefore, it is very likely that the number of cache cells may exceed the number of home cells on each processor, resulting in high memory overhead. Choosing a smaller cell size allows more home cells to be distributed to each processor to solve the problem. Figure 4.15 shows that cell size $16^3$ performs close to optimum in several volumes and presents a good alternative to balance the performance and memory overhead.

4.8.3 Cache Memory

Cache memory plays a pivotal role in rendering performance as it is the major resource to help in hiding communication latency. Our algorithm logically divides cache memory into two types: the near cache, which is used to store near-future cell requests or support advanced multiframe extension; and the far cache, which is a general cache pool
Figure 4.16 Near/Far cache size analysis. (a) and (b): Rendering Mix512 with a cell size of $32^3$ voxels on different number of processors. (c) and (d): Rendering Mix512 on 32 processors with different cell sizes. Far cache is set to 0 when analyzing the near cache, but near cache is set to one cell when far cache is analyzed. Our algorithm requires at least one near cache buffer to execute.

to host any other fetching requests. Both types of cache are interchangeable and a cache buffer may be tagged near or far several times during the course of execution. In Figure 4.16, we show the effect of different cache sizes on the rendering time under different numbers of processors and cell sizes. Some interesting observations were made from these
results. First, the optimal cache size seems to be independent of the number of processors used. Although the number of home cells per processor is four times fewer on 32 processors than on 8 processors, our algorithm is able to fulfill cell requests with the same efficiency using the same cache size on each processor (Figure 4.16(a) and (b)). The excellent caching efficiency is mainly due to prioritizing the cell requests in the fetch list. It allows cells closer to the anchor to be fetched first and improves the turnaround time on cache buffers. Secondly, as shown in Figure 4.16(c) and (d), different cell sizes, result in different optimal cache sizes. For the near cache, the optimal cache size is 20 cells for a cell size of $32^3$ voxels or 60 cells for $16^3$ voxels. As for the far cache, the optimal cache size for both cell sizes are about the same at 60 cells. Both types of cache would require at least 200 cells when the cell size is set to $8^3$ voxels. Although the number of needed cells decreases as the cell size increases, the overall memory taken by the cache pool is actually more for larger cells due to the faster cubic growth rate in cell size. The extra amount of cache memory may prevent the larger cell size from being efficiently utilized with a large number of processors.

The comparison of rendering times for different mix of near/far caches is shown in Figure 4.17. The optimal cache sizes for $32^3$, $16^3$, and $8^3$ voxels are chosen to analyze the effect of blending near/far cache ratio. For our test case, the performance difference is very limited, with most of them performing within 4% of the best mix. For single-frame rendering, the above results suggest a very flexible allocation for the near/far caches. We attribute such flexibility to the efficient out-of-order execution, which helps to close the
Figure 4.17 Cache type analysis. The chosen cache size (near + far) is 20 cells for $32^3$, 60 cells for $16^3$, and 200 cells for $8^3$. Rendering time is normalized to the fastest one in each data series.

A characteristic difference between near and far caches. For 180-degree or 360-degree multiframe renderings, however, the size of the near cache will be adjusted to reserve enough cache for the holding buffers needed in the implementation.

4.8.4 Rendering Time Analysis

To examine the constitution of rendering time, we compare the timings of Head256 and Mix256 from Table 4.2 (single-frame, denoted as $sf$) and Table 4.3 (multiframe, denoted as $mf$) with the following special timings:

- **Complete timing**: parallel rendering the images with a fully duplicated volume on each processor (*i.e.* no communication overhead.)

- **Ideal timing**: the theoretical lower bound reachable by a linear speedup. (*i.e.* no communication overhead, perfect load balancing.)
Figure 4.18 Comparison of rendering timing for different settings of (a) Head256 and (b) Mix256 datasets. Rendering time is normalized to the complete timing to show their relative performance.

- **Only-one-cache timing (cs=1):** parallel rendering the images with only one cache buffer. (i.e. full communication latency.)

By using the above special timing, we could observe the following informations:

- **Optimized communication latency:** comparing the regular timing to the complete timing.
- **Unbalanced loads:** comparing the complete timing to the ideal timing.
- **Full communication latency:** comparing regular timing to the only-one-cache timing.
- **Latency hiding efficiency:** comparing the full communication latency with the optimized communication latency.

We have observed in Figure 4.18 that the communication overhead is reduced by almost one half at 128 processors by applying multiframe rendering (from 57% to 27% for
Head256 and from 30% to 17% for Mix256). The high communication latency experienced by Head256-lf confirms our earlier observation that the lack of computation in smaller or opaque volumes may not be able to efficiently absorb the communication time. Comparing to the full communication latency, the multiframe rendering saves over 76% of the wait time for Head256 and 80% for Mix256, demonstrating a decent capability to hide most of the communication work. We expect to see an even higher ratio when more frames or a larger volume are involved. Finally, we also observe an unbalanced load gradually growing as the number of processors increases. At 128 processors it roughly equals the communication overhead experienced by the multiframe timing. This observation suggests that our simple static load balancing strategy works fine for a moderate number of processors (64 or less), but does not scale up well for a larger number of processors. Silva [92] shows a content-sensitive volume partitioning scheme based on axially-aligned BSP trees, which might help our algorithm to distribute volume cells more fairly to each processor. Nieh and Levoy [75] and Heirich and Arvo [43] demonstrate some dynamic load balancing schemes that seem applicable to the redistribution of unfinished rays. As the number of processors increases, we may need to investigate a more adaptive scheme to better balance the workloads.

4.9 Experimental Results on a Network of Workstations (NOW)

Network of workstations (NOW) present a different class of parallel computing platform. Instead of using specially designed hardware components, most NOW systems use standard workstations, off-the-shelf network interface cards, and local area networks
to connect all computing resources. They are easy to build and expand, and cost much less than a dedicated parallel computer like the Cray T3E. The only limiting factor is the network speed, which is usually at least one or two orders of magnitude slower than a specially designed parallel computer. However, many new standard networking technologies such as FDDI (100 Mbps), Ethernet2 (100 Mbps), and ATM (150 Mbps) have emerged recently to provide much improved bandwidth for these systems. As the computing power of a single workstation approaches the level of yesterday’s supercomputer’s, NOW’s have become an economical and feasible way to explore the power of parallel computing.

We ported our algorithm to the Workstation Cluster System (WCS) at OSC to evaluate its performance on such a loosely coupled parallel machine. The WCS at OSC currently consists of 11 Silicon Graphics O2 systems, each equipped with a 150 MHz R10000 processor, 128 Mbytes RAM, and a 4 Gbytes harddisk. The communication ports on each O2 include an Ethernet (10 Mbps) and an ATM OC-3 network interface. Compared to the Cray T3E’s interconnection network (480 MBytes/sec), the relatively slow communication interface on these workstations provides a good measure of the effectiveness of the algorithm’s latency hiding capability.

4.9.1 Speedup and Scalability

We tested our algorithm on the WCS with both ATM and Ethernet connections. The average frame time and speedup are shown in Table 4.4 and Figure 4.19. The first
Table 4.4 Average frame time (in seconds) for different rendering settings. "1f" denotes single-frame rendering, while all others are multiframe rendering with different number of frames. Bold-faced numbers reflect a strong communication presence in the rendering time.

<table>
<thead>
<tr>
<th># processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cube128-1f (ATM)</td>
<td>1.83</td>
<td>10.59</td>
<td>9.61</td>
<td>9.02</td>
</tr>
<tr>
<td>Cube128-20f (ATM)</td>
<td>1.83</td>
<td>1.05</td>
<td>0.67</td>
<td>0.58</td>
</tr>
<tr>
<td>Cube128-40f (ATM)</td>
<td>1.83</td>
<td>1.01</td>
<td>0.55</td>
<td>0.37</td>
</tr>
<tr>
<td>Mix256-1f (ATM)</td>
<td>37.48</td>
<td>67.00</td>
<td>55.45</td>
<td>47.11</td>
</tr>
<tr>
<td>Mix256-10f (ATM)</td>
<td>37.48</td>
<td>19.87</td>
<td>10.75</td>
<td>6.43</td>
</tr>
<tr>
<td>Mix256-20f (ATM)</td>
<td>37.48</td>
<td>19.85</td>
<td>10.17</td>
<td>5.56</td>
</tr>
<tr>
<td>Cube128-20f (Ethernet)</td>
<td>1.83</td>
<td>1.14</td>
<td>1.05</td>
<td>1.865</td>
</tr>
<tr>
<td>Cube128-40f (Ethernet)</td>
<td>1.83</td>
<td>1.01</td>
<td>0.56</td>
<td>0.93</td>
</tr>
<tr>
<td>Mix256-10f (Ethernet)</td>
<td>37.48</td>
<td>21.57</td>
<td>12.45</td>
<td>17.57</td>
</tr>
<tr>
<td>Mix256-20f (Ethernet)</td>
<td>37.48</td>
<td>21.05</td>
<td>11.25</td>
<td>11.69</td>
</tr>
</tbody>
</table>

Figure 4.19 Speedup for different rendering settings on a cluster of workstations with (a) ATM and (b) Ethernet networks.

observation from these results is that, even with the faster ATM interface, the single-frame rendering scenarios (Cube128-1f and Mix256-1f) show a big drop in performance when
employing multiple processors. This is due to the higher communication overhead in these “standard” networking environments, as comparing to Cray T3E’s high-speed interconnection network, and the lack of sufficient computational tasks within a single image to effectively hide the communication latency. The high communication cost is also observed in the multiframe rendering scenarios as they require more simultaneous frames than those on the Cray T3E to operate optimally. The ATM connection seems to pick up the network traffic quite well, achieving steady speedups across all numbers of processors with the increase of frame numbers in multiframe rendering. The Ethernet connection, however, could only sustain communication load for up to four processors even after applying an aggressive multiframe strategy. The sudden decline in rendering performance with eight processors is believed due to the saturated network bandwidth on the Ethernet network.

Due to the workstation’s memory constraint, we could not run Mix512 dataset on either one or two processors. We set up WCS with an ATM connection to render Mix512 dataset using four and eight processors. The average frame time for Mix512 (with 10-frame multiframe rendering) is 82.07 seconds for four processors and 44.63 seconds for eight processors. It shows a good scalability from four to eight processors (about 92% efficiency), and demonstrates the effectiveness of our algorithm to collaborate both processing and memory resources from multiple computers in a networked environment.
4.9.2 Rendering Time Analysis

![Graph showing rendering times for different settings of Mix256 on an ATM network.](image)

Figure 4.20 Comparison of rendering times for different settings of Mix256 on an ATM network. Rendering time is normalized to the *complete* timing to show the relative performance.

Figure 4.20 shows that the measured communication latency is about 15% of the rendering time when Mix256 is rendered with 20 frames in a multiframe rendering on 8 processors. This reflects a reduction of 92% of the wait time with respect to the full communication latency, which is about twice the total rendering time. Compared to the results on the Cray T3E, the latency hiding efficiency is better on WCS since its mediocre network throughput provides more room for improvement by our algorithm. The capability to hide most latency in both specialized and standard network environments makes our algorithm valuable to a wide range of parallel computers. The load imbalance factor is not a problem with such a small number of processors as Figure 4.20 shows almost identical performance between the *complete* and *ideal* timings.
4.9.3 Multiframe Comparison

Figure 4.21 Timing comparison for different multiframe approaches on an ATM network.

Our previous results show the impact of multiframe rendering on the algorithm’s performance: the multiframe approach reduces the communication cost and increases the rendering load to balance the computation and communication for smaller or opaque volumes. Our various extensions for multiframe rendering allow such a performance advantage to be exploited in a flexible manner. Figure 4.21 shows the overall rendering time for Mix256 on 8 processors by rendering a series of images from a viewpoint rotating around the Y-axis for 360 degrees. For such a rendering task, the 90-degree multiframe extension would require four separate rendering sessions, while the 180-degree extension needs two and the 360-degree extension manages only one. Generally speaking, the rendering performance improves as a more accommodating multiframe algorithm is used. The performance boost from the 90-degree extension to the 180-degree extension is
particularly notable, especially when rendering 10 frames. This is partially due to the lack of simultaneous frames in the 90-degree scenario (only two to three frames per session in this case), and partially because of the relatively low-overhead modification to achieve the 180-degree extension. The 360-degree extension requires a more sophisticated implementation and does not register the same kind of improvement over the 180-degree extension. However, the 360-degree extension still needs only 43% to 64% of the respective rendering time comparing to the 90-degree algorithm. Due to the memory constraint, we could not practically render hundreds of frames in a 360-degree rotation simultaneously. However, such a constraint does not limit the applicability of our algorithm in these situation. For a rotation covering too many frames, we could always interleave the rendered frames to achieve the entire rotation cycle with fewer frames. For example, to cover a 360-degree rotation with 200 frames, we could always render frames \([1,11,21,...,191]\) first, followed by frames \([2,12,22,...,192]\), etc. By applying the rendering in such an interleaving order, users can get a progressive update to the rendering results, with the benefits of the efficiency of multiframe rendering.

### 4.10 Discussion

Table 4.5 compares rendering time and scalability of several other parallel rendering algorithms to our method. Our ART algorithm demonstrates good performance and scalability on both the MPP and the NOW platforms (86% and 84%, respectively). As for other MPP implementations with scalability over 80%, Neih and Levoy [75] relies on DASH's efficient hardware cache protocol to access non-local data, while Ma, et al. [66]
gains on its unique composition pattern to reduce network congestion. Also, Corrie and Mackerras [18] requires volume duplication across clusters of neighboring processors or a relatively large amount of least-recently-used (LRU) cache size (about two to six MBytes per processor) to operate at an acceptable level. On the contrary, the ART algorithm would require only 100 to 640 KBytes of cache memory per processor, dependent upon the cell size. Other algorithms implemented on a NOW report a parallel efficiency of 52% [54] and 92% [66]. Ma, et al. [66] demonstrates an impressive scalability over an Ethernet network. In large part, this is due to the object-parallel approach, where communication only occurs in the compositing phase. However, their high scalability also reflects the computation-dominant nature of the object-parallel approach, as the slow rendering time reveals the possibility of much redundant computation (since there is no early ray termination).

Our work is mostly inspired by the work of Law [56][57], but with an attempt to optimize the latency behavior of a parallel ray-casting algorithm on a distributed memory system. The ray-front algorithm [56] reduces cell communication by advancing all rays in lock-step in a FTB cell traversal, which mimics a wave-front moving across object space. Although the algorithm can avoid cache thrashing and possesses a basic multiframe capability, it relies on an inefficient “prefetch-all” approach for latency hiding and does not allow for early ray termination. The active-ray algorithm [57], on the other hand, incorporates ray reordering and temporal coherency to achieve an impressive latency hiding result for smooth animations (less than three degrees per frame). The major
Table 4.5 Comparison of different parallel rendering algorithms.

<table>
<thead>
<tr>
<th>Method</th>
<th>Platform</th>
<th># processors</th>
<th>Volume/Image Size</th>
<th>Rendering Time (secs)</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machiraju[67]</td>
<td>IBM PVS</td>
<td>10</td>
<td>100^3/173^2</td>
<td>1.67</td>
<td>0.60</td>
</tr>
<tr>
<td>Palmer[77]</td>
<td>SGI Power Challenge Array</td>
<td>32</td>
<td>840 x 2595 x 480 / 800 x 600</td>
<td>1.61</td>
<td>0.67</td>
</tr>
<tr>
<td>Lacrouette[52]</td>
<td>SGI Power Challenge</td>
<td>16</td>
<td>256^2 x 226/256^2</td>
<td>0.077</td>
<td>0.74</td>
</tr>
<tr>
<td>Neih[75]</td>
<td>Stanford DASH</td>
<td>48</td>
<td>256^2 x 226/416^2</td>
<td>0.70</td>
<td>0.83</td>
</tr>
<tr>
<td>Corries[18]</td>
<td>Fujitsu AP1000</td>
<td>128</td>
<td>256^2 x 109/512^2</td>
<td>53.78</td>
<td>0.85</td>
</tr>
<tr>
<td>Montani[72]</td>
<td>nCube-2</td>
<td>128</td>
<td>97^2 x 116/350 x 250</td>
<td>5.14</td>
<td>0.74</td>
</tr>
<tr>
<td>Ma[66]</td>
<td>CM-5</td>
<td>512</td>
<td>256^3/512^2</td>
<td>3.52</td>
<td>0.87</td>
</tr>
<tr>
<td>Ma[66]</td>
<td>HP9000/730 with Ethernet</td>
<td>8</td>
<td>256^3/512^2</td>
<td>47.15</td>
<td>0.92</td>
</tr>
<tr>
<td>Law[55]</td>
<td>Cray T3D</td>
<td>128</td>
<td>256^3/256^2</td>
<td>0.38</td>
<td>0.60</td>
</tr>
<tr>
<td>Law[56]</td>
<td>Cray T3D</td>
<td>128</td>
<td>512^3/512^2</td>
<td>1.31</td>
<td>0.69</td>
</tr>
<tr>
<td>Law[57]</td>
<td>Cray T3D</td>
<td>128</td>
<td>256^3/256^2</td>
<td>0.16</td>
<td>0.76</td>
</tr>
<tr>
<td>Law[54]</td>
<td>Dec-Alpha 3000 with FDDI</td>
<td>8</td>
<td>128^3/128^2</td>
<td>0.60</td>
<td>0.52</td>
</tr>
<tr>
<td>Silva[92]</td>
<td>Intel Paragon</td>
<td>64</td>
<td>256^2 x 937/400^2</td>
<td>0.78</td>
<td>0.46</td>
</tr>
<tr>
<td>Wittenbrink[111]</td>
<td>Proteus</td>
<td>32</td>
<td>256^3/256^2</td>
<td>4.32</td>
<td>0.69</td>
</tr>
<tr>
<td>ART Algorithm</td>
<td>Cray T3E</td>
<td>128</td>
<td>512^3/512^2</td>
<td>0.91</td>
<td>0.86</td>
</tr>
<tr>
<td>ART Algorithm</td>
<td>SGI O2 with ATM</td>
<td>8</td>
<td>256^3/256^2</td>
<td>5.56</td>
<td>0.84</td>
</tr>
</tbody>
</table>

disadvantage is that a large cache memory (about four to six times the volume size) is required in order to stabilize the rendering time. Our new algorithm combines features of both approaches, by using a deterministic object-order traversal (FTB cell order) with an efficient latency hiding mechanism (out-of-order ray processing and scheduled cell fetching). Rays are traced in a normal way, i.e. no lock-step, thus improving rendering efficiency by avoiding overheads in managing multiple ray steps. The regular
communication order in object-space improves cache efficiency and avoids redundant fetching of cells. Such an order also helps in establishing a caching system without cache conflicts and thrashing. The extended multiframe capability is also general enough to handle both animations and perspective projection. All these features are achieved without the overhead of an enormous amount of cache memory.

Our method has some disadvantages and limitations. The most notable disadvantage is the additional ray memory required to hold the unfinished rays. For example, an image of \(512^2\) pixels requires 5 MBytes of memory to initialize all rays. Such a compulsory requirement implies two limitations. First, the algorithm is memory efficient only when rendering large datasets (\(512^3\) voxels and up), especially with the multiframe situation. For example, simultaneously rendering 10 frames of images of \(512^2\) pixels for a volume of \(512^3\) voxels requires 50 MBytes overall for ray storage, which is roughly 39\% of the dataset memory. Rendering the same number of images for \(256^2\) pixels and \(256^3\) voxels requires 78\% of the dataset memory, while the configuration of \(128^2\) pixels and \(128^3\) voxels requires 156\%! Our earlier experiments for achieving stable performance under a slower network environment show that, many simultaneous frames are required for a small dataset. Therefore, when the dataset is small, it might be beneficial to simply duplicate the dataset on each processor and render, rather than apply the multiframe algorithm. Secondly, the ray storage requirement implies that memory efficiency is not only data size dependent, but also image resolution dependent. In some cases, where a high resolution image is rendered from a moderate size dataset, the ray
storage may exceed the volume memory even in a single frame situation. For parallel projection of an equilateral volume $N^3$, it is observed that images larger than $2N \times 2N$ pixels will not contain any new rendering information. Other limitations include the different multiframe extents (90 degrees, 180 degrees, or 360 degrees) covered by different extensions, and all of them require static dataset. Lastly, an efficient extension to a full ray tracing — with the capability of spawning secondary rays — requires further research.
CHAPTER 5

Rendering Large Architectural Datasets with Visibility Computation

Virtual walkthroughs of synthetic buildings have been used in applications such as architectural evaluation, design presentation, and virtual reality systems, where a computer generated architectural model is explored and examined by controlling a virtual observer. Such a virtual examination is critical in verifying the aesthetic perception and regulation compliance of an architectural design before the actual construction takes place [96]. In order to maintain the illusion of reality during such a virtual walkthrough, the visualization system must update screen images beyond an interactive frame rate (e.g. 10 to 15 fps). Otherwise, users may be distracted by the jerky display and quickly lose the sense of interaction [10].

Generating an image from a 3D model includes finding the right subset of visible surfaces and rendering them in a correct order. Traditionally, this is called the visible surface determination problem, and is solved by a combined effort of back-face culling, view frustum clipping, and hidden-surface removal algorithms [31]. The back-face culling
uses the direction of a surface normal to see if the surface needs to be displayed to the current viewpoint. The view frustum clipping, on the other hand, remove portions of a model that do not project to the viewing plane. Both methods employ fast, simple calculations to coarsely screen out invisible objects. A hidden-surface removal algorithm then solves the ordering problem for the remaining objects to render a correct image. While such a general framework is capable of rendering any situation, for some cases it may not operate most efficiently. For example, when a model has great depth complexity along the view direction, back-face culling and view frustum clipping will pass a significant number of objects due to their loose qualification criteria. That would mean that the more expensive hidden-surface removal step would be responsible to process these objects, when in fact, many of them may actually be invisible to the eye due to occlusion. Moreover, the entire computation is highly view-dependent. It can only be applied during rendering time and whenever the view changes, the entire computation needs to be performed again. For large datasets, applying such a process on the entire model may be very time-consuming.

Realistic architectural models with detailed interior structures and decorations typically consist of hundreds of thousands of polygons. The complexity of these models is well beyond the rendering capacity of current graphic workstations for interactive applications. However, most architectural models share a unique characteristic. Groups of connected opaque occluders (i.e. walls) partition the entire model space into many individual regions (i.e. rooms). An observer may have only a limited view due to
occlusion between regions. This implies that the subset of potentially visible objects at any given location will be much smaller than the entire model. As a result, the rendering speed can be dramatically improved if we render only this small visible portion of the model.

Following this direction, we propose a new visibility determination algorithm for architectural model walkthroughs. The algorithm is based on a cell-to-cell, object space method, which precomputes the visibility information for each cell region. Unlike other visibility algorithms for these environments, which require prerequisite knowledge of portals in a model, our algorithm employs a simple space subdivision scheme and a fast discrete algorithm to determine the potentially visible subset in each subdivision. By controlling the resolution of the dividing grids, our algorithm can trade accuracy with processing time. The result is a fast, easy-to-apply algorithm which can efficiently cull a complex architectural model using a conservative visibility approximation. This chapter is organized as follows. Section 5.1 gives a general overview of the algorithm. Detailed steps to prepare an architectural model for visibility computation is covered in Section 5.2, followed by the discussion of our new cell-to-cell visibility algorithm in Section 5.3. Section 5.4 demonstrates how the visibility information is incorporated into the walkthrough process. In Section 5.5, we propose a memory-saving scheme, called difference lists, for the preprocessed visibility information. The implementation results and discussions are included in Section 5.6 and Section 5.7, respectively.
5.1 Overview of the Method

While computer modeling techniques allow for the creation of sophisticated and imaginary architectural models, most real world architectural designs deal with partitions of space which are oriented, that is, with a fixed reference to the direction of up (ceilings) and down (floors). Typically, walls are the major space partitioning elements in a model, where they vertically extrude from the floor and connect to the ceiling. Walls, ceilings and floors together form space enclosures [120], which define individual spatial regions with opaque partitioning elements, or occluders.

Our algorithm is an object-space visibility computation method which takes advantage of occlusion caused by the inherent occluders in a building environment. The model preparation stage of the algorithm, first partitions the input dataset into individual space regions by applying a space subdivision. Each region is called a cell, which is the fundamental unit of space that the visibility information is computed upon. As for the input dataset, we assume both a 3D model and a 2D floor plan of the model are available. While the 3D model contains everything included in the virtual world, the 2D floor plan only depicts the wall definitions of an architectural structure. Floor plans have been linked with architectural design since its inception, and when combined with section and elevation drawings, form the basis of a 3D model. Our use of 2D floor plans is based on the observation that most walls are vertical extrusions from the floor. The complexity of the visibility problem is greatly reduced by projecting the walls to a 2D floor plan. Both
the 3D model and the 2D floor plan are processed by the same space subdivision step, generating a one-to-one correspondence between cells from both types of data.

After the model preparation step, the visibility computation stage precomputes the potentially visible set (PVS) of objects for each defined cell region. An exact solution for the visible set is desired, but not necessary, because the computational cost of this solution may be large. In fact, the problem may be better addressed by the hidden-surface removal step within the graphics rendering pipeline. Our goal is to precompute a conservative estimation of the objects that are visible from any location within a given cell region. Our algorithm utilizes a simplified sight corridor approach, which will declare the blockage of a sight corridor [118] only if there exists a group of connected wall segments which totally prevent two cells from seeing each other (Figure 5.1(a)). Although such a criterion would definitely miss some blocking configurations (an example is shown in Figure 5.1(b)), we have observed a good culling ratio in architectural environments since wall segments are mostly connected together.

The precomputed visibility information is loaded before the start of the walkthrough stage. During the walkthrough, the cell where a virtual observer is located is determined and the objects stored in the cell's PVS are rendered. The list of potentially visible objects remains the same as long as the observer stays within the cell, since the visibility information stored in the PVS is omnidirectional. However, as the observer turns or moves, the graphics rendering pipeline performs its own culling to display objects only
Figure 5.1 Blocking configuration for the simplified sight corridor approach. (a) Blocked. (b) Not blocked (since no single connected wall segments can block the entire corridor. Dark lines represent wall segments.

within the current view frustum. Rendering efficiency is greatly improved because the total number of objects is reduced and, more importantly, so is the number of occluded objects within the view frustum. We will discuss the detailed design of these stages in the following sections.

5.2 Model Preparation

5.2.1 Dataset Assumptions

Even though our algorithm assumes that any wall occluders must be perpendicular to a referenced floor, it allows for arbitrary 2D wall orientations on the floor. For architectural models, such an assumption is fair enough since most walls in a building are vertical extrusions. Walls with other 3D orientations are ignored as occluders. Since the floor plan is in most cases an integral part of architectural design, it may not pose much of
a burden in a formal architectural project. Most models from architectural CAD (Computer Aided Design) applications have both 2D and 3D representations. However, the case where models constructed directly in 3D without a 2D floor plan, especially for those specifically designed for graphics rendering purposes only, requires preprocessing to get a 2D floor plan, either by orthographic projection to the floor or intersecting a horizontal cut plane for a cross section of the model. In either case manual intervention may be required to separate occluders from other objects in the model, or to modify the floor plan result to conform to the required format (see below).

Our 2D floor plan is represented as axial skeletons. An axial skeleton [120] is generated from a closed curve by pairing an exterior curve of the same shape with a coincident interior curve whose vertex order is reversed. Open curves (or line segments in our context) are treated as a degenerate case with no interior curves (Figure 5.1(a)). A larger skeleton is formed from a sequence of curves, by merging their axial skeletons sequentially (Figure 5.1(b)). An architectural modeling system allows for a full set of modeling operations on axial skeletons, such as adding/deleting walls, adding/deleting doors, windows, or plain openings of a variety of types [120]. We use the axial skeleton as our floor plan format because it is compatible with most CAD software and provides simplicity in handling the wall segments. Note, that in regard to visibility computations, the vertex order in an axial skeleton is not important.
5.2.2 Space Subdivision

Our algorithm partitions the model space using a 2D regular grid space subdivision. Each small region, or cell, maintains a local data list which stores a list of 3D objects whose bounding box is inside or intersects with the cell. Objects are not divided, i.e. all cells that intersect an object's bounding box will register that object. This approach allows the use of a level-of-detail (LOD) approximation [13][35] to further accelerate rendering performance. Space subdivision also applies to the 2D floor plan, where each wall (line) segment is registered in the wall list of the cell it intersects. The axial skeleton representation for both a closed curve and an open curve. The exterior curve has clockwise vertex order and the interior curve has a counter-clockwise order. (b) A simple line segment sketch and its axial skeleton representation.
representation stores wall segments as a set of non-crossing polylines, where each polyline forms a closed curve from a vertex list \([v_0, v_1, \ldots, v_{n-2}, v_{n-1}, v_0]\). For simplicity, we use the index of the first vertex to identify each line segment, \(i.e., \overline{v_i v_{i+1}} = l_i\). Each line segment is scan-converted into a 4-connected discrete line. Each cell visited by the discrete wall line will add the index of the wall to its wall list. A list of discrete colinear points is stored with each line segment, where each point is formed by stepping along the line segment in unit cell length. These points are used by the incremental visibility algorithm to quickly find the location where the wall segment exit a sight corridor (Section 5.3.3). Figure 5.3 gives an illustration for a single wall segment and its colinear points. We will use the terms data cell and wall cell to refer to a cell with a data list and a wall list, respectively.

The resolution of the space subdivision represents a trade-off between computational accuracy and memory space. Since the PVS of a cell is a union of all visible objects from any point within the cell, a larger cell size implies a higher degree of estimation in PVS, hence requiring more processing time during the walkthrough stage. Alternatively, a smaller cell size improves the accuracy of the visibility computation, but requires a longer time to compute and more memory to store the visibility information. For precomputed visibility information, we propose in Section 5.5 an efficient storage scheme which takes advantage of the spatial coherence among cells to reduce up to 89% of the required storage.
Figure 5.3 Scan-conversion of a wall segment to a space subdivision grid. Wall cells are shown in solid squares. Each wall cell registers the wall segment by saving a pointer to the first vertex of the line. The colinear points represent points at unit cell length along the line segment. They are stored in an external list reachable from $v_i$.

5.3 Visibility Computation

5.3.1 A Basic Algorithm

The pseudo code shown in Figure 5.4 illustrates the general idea of our cell-to-cell visibility computation scheme. The $Source\_Set$ and the $Destination\_Set$ are sets of cells that are involved in the visibility computation, where a cell $Src$ collects visibility information from the other cell $Dest$. A sight corridor $S\_Corridor$ is constructed for each pair of cells from the $Source\_Set$ and the $Destination\_Set$. The choice of these two sets

109
should contain all possible corridors of interest in their combination. Corridor cells (explained later) in each S_Corridor are examined to see if any wall segments in their wall lists obscure the entire sight corridor. This result is encoded in S_Corridor and is used to update the PVS in the source cell of the corridor.

```/* The cell-to-cell visibility algorithm template */
for each cell Src ∈ Source_Set do
  for each cell Dest ∈ Destination_Set do
    S_Corridor = InitializeCorridor(Src, Dest);
    S_Corridor.blocked = false;
    for each corridor cell C_Cell ∈ S_Corridor do
      for each unvisited wall segment Wall ∈ C_Cell.wall_list do
        if S_Corridor.blocked is false then
          S_Corridor.blocked = CorridorBlockingTest(Wall, C_Cell, S_Corridor);
        else
          goto UPDATE;
    UPDATE: UpdateVisibleSet(Src.PVS, S_Corridor);
```

Figure 5.4 The cell-to-cell visibility algorithm template.

Next, we provide a more detailed explanation of each key step. Function InitializeCorridor() is responsible for setting up all initial data structures used for the current corridor, including the initialization/reinitialization of all cells, counters, look up tables, and flags. This function initializes S_Corridor, an abstract data type (ADT) that contains all corridor-related information. Conceptually, S_Corridor represents a continuous corridor sweep from source cell Src to destination cell Dest (Figure 5.5).
follows that any cell which entirely or partially intersects the boundary of a continuous corridor must be examined, because such cells may contain wall segments that block $S_{Corridor}$. However, the connectivity of wall cells implies that it is sufficient to check only a subset of cells as long as the selected subset does not miss any blocking configuration. An example subset would be cells stamped by an 8-connected Bresenham line [9] from $Src$ to $Dest$, since the 4-connected wall cells are guaranteed to be 8-tunnel free. These candidate cells are the *corridor cells* for $S_{Corridor}$ and will be the only intermediate cells investigated by the visibility algorithm. Figure 5.5 shows the 4-connected wall cells intersecting with the 8-connected corridor cells in a corridor sweep between two cells.

![Figure 5.5 A continuous corridor sweep and its corridor cells.](image)

Blocking configurations are tested by the function $CorridorBlockingTest()$ (discussed in Section 5.3.3) with each wall (line) segment. A negative return by the function implies that a group of connected walls extended from a chosen wall segment
fails to form a successful blocking configuration. For efficiency reasons, these visited wall
segments are excluded from further testing for the current corridor. To prevent from re-
evaluating these visited walls, we add a "corridor stamp" to each vertex in the floor plan.
Every time a new \textit{S\_Corridor} is initialized, a global counter is increased by one to
generate a new stamp value. Wall segments visited by the current \textit{S\_Corridor} will save the
new stamp value in their starting vertex. An \textit{S\_Corridor} will only test those new wall
segments with a stamp value different from its own. If a round-off error occurs when the
global counter increases, then all stamp values are reset. This scheme provides a simple
and efficient method to avoid stepping over visited cells.

Given a sight corridor from \textit{Src} to \textit{Dest}, the basic algorithm decides if \textit{Dest} is
visible from \textit{Src}. If a blocking configuration is found, \textit{Dest} is expected to be \textit{invisible} from
\textit{Src}. Otherwise, objects in the data list of \textit{Dest} are added to the PVS of \textit{Src}. This claim
holds for the reverse direction from \textit{Src} from \textit{Dest} due to commutativity in the visibility
relationship. If there are \textit{N} cells (enumerated $0, 1, 2, ..., N-1$) in the space subdivision, the
two outer loops in Figure 5.4 will execute (at most) a total of $N^2(N-1)/2$ iterations to cover
all combinations. More precisely, \textit{Source\_Set} will have cells from $[0, 1, 2, ..., N-1]$ and the
\textit{Destination\_Set} from $[Src+1, Src+2, ..., N-1]$. The number of iterations is greatly reduced
by skipping corridors with empty data cells at both ends.
5.3.2 An Incremental Algorithm

A major drawback of the basic algorithm is computational redundancy where given a source cell, the blocking information of a certain destination cell can not be shared by another destination cell along the same corridor. Figure 5.6 illustrates this weakness. The original sight corridor is defined from Src to Dest. While the algorithm detects a blocking configuration at corridor cell M and finds that Dest is invisible from Src, it cannot determine whether D1 or D2 are visible from Src. Thus, new sight corridors have to be constructed and the same wall group has to be examined again to determine visibility. To solve this inefficiency the incremental algorithm, rather than simply answering a yes-or-no question for a particular destination cell, finds the aggregate visibility for the source cell in a given corridor. For example, instead of simply declaring Dest invisible, the incremental algorithm will also report that all cells from M to C are visible from Src in Figure 5.6.

![Figure 5.6 A blocking configuration from Src to Dest.](Image)

To maximize the benefit of incremental visibility processing, the initial destination cell should be as far away from the source cell as possible. On a regular grid space
subdivision, it implies that destination cells should be on the boundary of the grid. For each source cell, if sight corridors are defined to extend to each and every boundary cell, we can guarantee to cover all possible destination cells and maintain the efficiency of the incremental approach. Therefore, using the same algorithm template in Figure 5.4, an incremental algorithm uses the same Source_Set as the basic algorithm but changes Destination_Set to the set of all boundary cells. The total number of corridors initiated will be $N \cdot 4\sqrt{N}$. The incremental algorithm also requires a different version of CorridorBlockingTest() function that records the trace of wall segments.

Figure 5.7 The incremental algorithm introduces more over-estimation and non-conservative results. Cell 1 is a corridor cell for all three different sight corridors and cell 2 is only covered by corridor A. In (a), cell 1 is claimed visible by corridor A and B and is indeed visible since the blocking in corridor C does not fully obscure the cell. In (b), even though the blocking in corridor B is enough to fully hide out cell 1, corridor A will make cell 1 still visible. For a non-conservative case, observe the upper-left corner of cell 2. Although cell 2 is claimed not visible by corridor A, it is difficult to tell whether the corner can be fully blocked by the given blocking configuration.
It is possible for the incremental algorithm to introduce more overestimation than the basic algorithm, because the cell coverage varies for different cell locations and sight corridors (Figure 5.7). However, a more serious drawback is that it may produce a non-conservative result (Figure 5.7(b)). To ensure this conservative property, we make sure that a blocking configuration of a sight corridor can totally obscure any invisible corridor cells. One way to ensure this is to perform blocking on a wider corridor which entirely covers corridor cells (Figure 5.8). Without loss of generality, assume the angle of a corridor is less than 45 degrees. To correctly represent a continuous line in discrete space, the chosen cells are closer to the continuous line than any other cells at an integer grid. The Bresenham line algorithm is a midpoint algorithm, and when applied to this context implies that \( \Delta y \leq 0.5 \cdot cell\_length \), where \( \Delta y \) is the \( Y\)-direction distance between the center of any corridor cell and the continuous line. Also note that \( \Delta y \) is the maximum vertical length of each "bump" not covered by the original sight corridor. We expand the extent of a sight corridor \( \pm 0.5 \cdot cell\_length \) along the \( Y\)-direction, so that all corridor cells are entirely covered. The extended sight corridor guarantees a conservative result, and avoids the overestimation problem caused by different corridor evaluations. However, some overestimation is introduced by the wider corridor size because it is more difficult to block it.

5.3.3 The Corridor Blocking Test

Given a seed wall segment, the corridor blocking test determines whether a wall trace from the seed wall reaches out to both sides of the corridor boundary. The
Figure 5.8 The extended sight corridor expands the coverage of the continuous sweep to include "bumps" not covered by the original sight corridor.

Incremental algorithm also finds the farthest cell that is visible from the source cell. To facilitate such a test, we first define the space partitioned by a corridor. An illustration is shown in Figure 5.9(a). The side boundaries are the corridor's exterior boundary lines parallel to the line direction, and the end boundaries are the interior cell boundary lines of both the source and destination cells. The inside area of a corridor is defined by the closed concavity formed by the side and end boundaries. The outside area of a corridor is the union of the two open half-spaces bounded by the tangential extension of the two side boundaries. The remaining space partition is called the neutral zone, an area where no side information is available. Our algorithm only initiates a wall trace from inside a corridor, and immediately terminates one when it leaves. Therefore, a valid blocking configuration forms a connected wall trace inside the corridor and reaches the two open half spaces without stepping into the neutral zone. The prohibition of the neutral zone is due to the
fact that a wall trace may reach both sides of a corridor through the neutral zone without fully obscuring the corridor (Figure 5.9(b)). In practice, the initialization of a sight corridor will automatically mark the wall lists of the source and destination cells as “visited” before any blocking test is conducted.

![Diagram of a sight corridor and an invalid blocking configuration through the neutral zone](image)

Figure 5.9 (a) The definition of a sight corridor. (b) An invalid blocking configuration through the neutral zone.

The pseudo code for the corridor blocking test for the basic visibility algorithm is shown in Figure 5.10. To trace a wall connection, simply follow the two vertices of the seed wall to other vertices. Since the floor plan has non-overlapping polylines, there is no need for vertex back tracking. Function `CorridorContainmentTest()` (discussed in the next section) checks whether a vertex is inside or outside the `S_Corridor`. The evaluation result of the trace determines the blockage of the corridor.
Figure 5.10 The corridor blocking test for the basic visibility algorithm.

This test for the incremental visibility algorithm is trickier (Figure 5.11). The algorithm assumes that the corridor cells are generated in an ascending index order from source to destination, which means the farther corridor cell has a greater index. In order to correctly determine which cells are visible, the algorithm needs to know where the blocking configuration occurs. The algorithm uses two variables, First_Max and Second_Max, to record the farthest cells the first and second traces have ever crossed. For example, as shown in Figure 5.6, even though the trace starting from cell M leaves both sides of the corridor at cell D2, the farthest visible cell is C because the second trace
previously reached it. In this case First_Max is D2 and Second_Max is C. Each time a new vertex is found to be inside the corridor, the associated corridor cell of the vertex is used to update the max variables. If the vertex is outside all corridor cells, the corridor cell which has the same major-axis cell coordinate as the vertex's containing cell is used (the major axis refers to the axis of the longer projection of the line). Once a trace leaves the corridor, the algorithm needs to determine at which corridor cell (or the closest corridor cell in the major-axis measure) the trace crosses the side boundary. To do this the precomputed colinear points are used to fine-trace the out-going line segment. The corridor cell associated with the exit point is then used to finalize the max information. The function finally returns a blocking flag value and stores the index of the farthest visible cell in S_Corridor.

5.3.4 Corridor Containment Test

The corridor containment test checks whether a vertex is located inside or outside a sight corridor. Since no walls intersect the neutral zone, every vertex will necessarily return a result in one of the following values: inside, plus-outside, or minus-outside (plus and minus are used to distinguish the two different half spaces in the outside area).

We've developed a geometric approach that quickly identifies the positional relationship between a vertex and a corridor. Without loss of generality, assume the slope of the corridor is between zero and one, as shown in Figure 5.12. For a sight corridor from cell \((x_0, y_0)\) to cell \((x_f, y_f)\), the center line equation is expressed as:
/* Corridor Blocking Test for the incremental algorithm */
/* Assume walls that traverse the neutral zone are marked “visited.” */

CorridorBlockingTest (Wall, C_Cell, S_Corridor)
{
/* First trace: traverse from the first_vertex to the prev_vertex */
Trace = Wall.first_vertex;
Trace.cstamp = S_Corridor.cstamp; /* mark visited */
First_Side = CorridorContainmentTest(Trace, S_Corridor);
First_Max = C_Cell; /* record the farthest corridor cell ever reached */

while First_Side is INSIDE and Trace.prev_vertex has not been visited do
    First_Max = MaxCell(Trace.C_Cell, First_Max);
    Trace = Trace.prev_vertex;
    Trace.cstamp = S_Corridor.cstamp; /* mark visited */
    First_Side = CorridorContainmentTest(Trace, S_Corridor);

if First_Side is OUTSIDE then
    Use the colinear points at Trace to decide where the wall goes outside and let the associated corridor cell at the exit point be Max_Out;
    First_Max = MaxCell(First_Max, Max_Out);

/* Second trace: traverse from the second_vertex to the next_vertex */
Repeat same operations for Second_Side, Second_Max and Wall.second_vertex, with the trace following the direction of Trace.next_vertex;

if First_Side \cup Second_Side indicates both sides of S_Corridor then
    S_Corridor.Max_Cell = MaxCell(First_Max, Second_Max);
    return true;
else
    S_Corridor.Max_Cell = S_Corridor.Dest;
    return false;
}

Figure 5.11 The corridor blocking test for the incremental visibility algorithm.

\[ y = mx + c, \text{ where } m = \frac{y_1 - y_0}{x_1 - x_0} \text{ and } c = y_0 - mx_0. \] (5.1)

Assume the cell length is \( S \), the Y-directional distance between side boundaries and the center line will be:
Given a vertex \( (P_x, P_y) \), we test if the \( Y \)-directional distance between the vertex and the center line is greater than \( N_{\text{basic}} \). If it is, the vertex is outside the corridor; otherwise, it is inside. To calculate the \( Y \)-distance from \( (P_x, P_y) \) we define:

\[
\Delta y_p = P_y - mP_x - c. \tag{5.3}
\]

The vertex is inside the corridor if the following assertion is true:

\[
-N_{\text{basic}} < \Delta y_p < N_{\text{basic}}. \tag{5.4}
\]

After substituting and rearranging the terms, we have

\[
(-N_{\text{basic}} + c) < P_y - mP_x < (N_{\text{basic}} + c). \tag{5.5}
\]

The expressions at both ends of Equation 5.5 are constants and may be precomputed as well as \( m \) and \( c \) when a corridor is initialized. Thus, a containment test for a vertex requires only one multiplication, one addition and two comparisons. If the middle expression of Equation 5.5, \( P_y - mP_x \), is greater than the right-end expression, the vertex is
marked plus-outside. If it is less than the left-end expression, it is marked minus-outside. By using the Y-directional distance (or X-directional distance when the slope is steeper), we avoid the need to compute the expensive calculation to find the perpendicular distance to the center line. The same computation is utilized for the extended sight corridor simply by replacing $N_{\text{basic}}$ in Equation 5.5 with the following term:

$$N_{\text{extended}} = S + \frac{S \cdot |m|}{2}. \quad (5.6)$$

### 5.4 Walkthroughs with Visibility Information

During the walkthrough stage, the rendering process requires a slight modification to take advantage of the precomputed visibility information. Conceptually, we send the PVS to the rendering pipeline according to the observer's cell location. However, most high level visualization toolkits (such as SGI Performer or Inventor) require the construction of a scene database, which is usually a hierarchy of objects in the model. The rendering engine constantly traverses and renders the scene database without user intervention and takes advantage of the efficient tree hierarchy and caching of object nodes. To avoid maintaining a scene database for each PVS, we use an approach similar to the LOD approximation by modifying the original scene hierarchy to allow conditional rendering for each object. For each displayable object, we insert a switch node (e.g., a `pfSwitch` node in Performer or a `SoSwitch` node in Inventor) between itself and its parent node. During the pre-cull callback (a functional stage executed right before the rendering action takes place), our run-time routine updates these switch nodes according to the PVS in the current cell. By switching on/off object nodes, we avoid building separate databases.
for each view by adjusting the original scene database to reflect the visibility status. Since we can directly access the switch nodes by an external pointer array, the update only requires linear time proportional to the length of the PVS. For a large scene database, we employ a quadtree-based switch hierarchy to quickly turn on/off an entire group of objects.

Figure 5.13 shows images from a walkthrough of an architectural dataset described in the Results section. The snapshots highlight the portion of the model that is turned on for a given observer’s cell location. This omnidirectional PVS is used to cull and render based on the observer’s view direction. The images show clearly that our algorithm can efficiently cull most objects behind a contained area, which greatly reduces the depth complexity for all directions. Such an “occlusion culling” operation is not possible through the traditional view frustum or back-face culling steps.

5.5 Memory Management with Difference Lists

The idea behind the visibility precomputation is to trade off space for time. By storing a PVS at each cell, simpler models are dynamically rendered using the observer’s location. However, memory and disk storage requirement to store PVS’s may be massive, especially for large complex models using a high resolution space subdivision. To reduce this overhead, we propose the use of a difference list to store visibility information found during preprocessing. Our choice is based on the observation that many cells in this case show a high degree of spatial coherency, where cells in a close neighborhood tend to have
Figure 5.13 A walkthrough of an architectural model with preprocessed visibility information. (a) Space subdivision grid and the object/wall placement. (b) A view of the 3D model. (c) A snapshot of the walkthrough. Objects in the PVS of the current observer's cell are highlighted. The arrow shows the observer's position. (d) Another snapshot.
a similar (or even the same) PVS. When the difference between the PVS's of neighboring cells is small, we need to store the complete list only for a few cells and use the difference lists to restore a correct PVS for other cells. Given two PVS's $S_a$ and $S_b$, difference lists are computed as follows:

$$S_{ab} = S_a - S_b = S_a \cap \overline{S_b} \quad \text{and} \quad S_{ba} = S_b - S_a = S_b \cap \overline{S_a}.$$  \hfill (5.7)

If either $S_a$ or $S_b$ contains a complete list, the other can be reconstructed by:

$$S_a = S_b - S_{ba} + S_{ab} \quad \text{or} \quad S_b = S_a - S_{ab} + S_{ba}.$$  \hfill (5.8)

Once a cell is restored, its neighbors are restored using its PVS. This means that in a walkthrough application, where an observer typically travels between connected cell blocks, most PVS's are constructed in a single restoration step with almost no performance cost. Multi-step restoration is required when an observer abruptly "jumps" into a new cell or the new cell is not connected to the originating cell by difference lists. The difference list scheme efficiently stores precomputed information with only a small cost in performance in the walkthrough.

To implement this approach we define a subset of cells called *wells*, which contain complete PVS's, and a set of *difference edges* that connect non-well cells to at least one well (which guarantees the restoration of a PVS for any cell). A difference edge is defined by a two-way link between two cells, representing the difference lists $S_{ab}$ and $S_{ba}$. Each cell computes difference edges with its 4-connected neighbors, and splits the storage responsibility with its neighbors by saving only the four out-going links in its cell (the other four in-coming links are saved as the out-going links in their respective neighbor.
cells). In order to choose a working subset of wells and generate a connected network for PVS restoration, we utilize the heuristic algorithm shown in Figure 5.14.

/* The difference list processing algorithm*/
/
Step 1: compute the difference lists where applicable. */
for each cell A in the space subdivision grids do
  for each 4-connected neighbor B of cell A do
    if \[ \frac{|S_{AB}| + |S_{BA}|}{|S_A| + |S_B|} \leq Diff\_Threshold \]
      then Compute the difference edge between A and B;

/* Step 2: assign wells and compute distance transform for each region */
for each cell S in the space subdivision grids in a scanline order do
  if S does not belong to any region then
    Start a new region with S.distance = 0 and S.well = true;
    Region_Done = false;
  /* Regular distance transform rounds */
  while Region_Done is false do
    Perform the distance transform loop until stable, where:
    Only cells directly or indirectly get assigned distance by S participate in the distance propagation;
    Each participating cell records a distance value and the direction it comes from;
    Region_Done = true;
  for each cell C in the current region do
    if (C.distance mod (2*Max_Distance)) equals to 0 then
      C.distance = 0;
      C.well = true;
      Region_Done = false;

/* An extra clean-up round for Max_Distance+1 ...2*Max_Distance-1 */
for each cell C in the current region do
  if C.distance equals to Max_Distance+1 then
    C.distance = 0;
    C.well = true;
  Perform the distance transform loop as before;

Figure 5.14 The difference list algorithm.
The algorithm consists of two major steps: computing difference lists and applying distance transform. In the first step, whether two neighbor cells compute a difference edge is determined by a threshold (\textit{Diff\_Threshold} or \textit{Difference Threshold}) because if two cells are very different, the cost of storing four difference edges may be higher than storing two separate complete lists. The difference between two cells is measured by

\[
\frac{\|S_{AB}\| + \|S_{BA}\|}{\|S_A\| + \|S_B\|}, \tag{5.9}
\]

where in the equation the numerator is the number of objects in the difference lists and the denominator is the number of objects in the complete lists. A difference edge is computed only when the measured difference in Equation 5.9 is less than \textit{Diff\_Threshold}. The application of a difference threshold means that some cells may be totally isolated while others may form several connected regions, which are processed in the next step to establish a PVS restoration network within each region.

In order to identify connected regions and compute a restoration path for each cell, a modified distance transform algorithm [104] is used to propagate distances through difference edges. Cells reachable by a sequence of difference edges are identified as one region and their city-block distances to the closest well are computed. The maximum distance a non-well cell should travel before it reaches a well is defined by a user parameter \textit{Max\_Distance}. It prevents forming long PVS restoration sequences which may spend too much time to reconstruct during the walkthrough stage. A distance of zero is assigned to the first encountered cell in a region using a greedy approach, which also tags it as the first well in the region. The distance transform algorithm is applied to the region,
and assigns distance values to other cells in the region with respect to the first well. When a cell receives a new distance value it also stores the direction from which the value was propagated, and uses it to guide the multi-step PVS restoration during walkthroughs. At the end of distance transform, all cells with distance values equal to $2n \cdot \text{Max\_Distance}$, where $n$ is a positive integer, are tagged as new wells and re-assigned a distance of zero. The assignment of new wells triggers the next round of distance transform, and the entire process repeats until no cell is located beyond $2 \cdot \text{Max\_Distance}$ steps from a well in the region.

Our greedy approach is guaranteed to terminate because, after adding new wells as "transceivers" at distances of $2n \cdot \text{Max\_Distance}$, all cells will be within $2 \cdot \text{Max\_Distance}$ steps away from some wells. By placing a well at every $2 \cdot \text{Max\_Distance}$ steps, the algorithm guarantees that cells between two wells can be reached in no more than $\text{Max\_Distance}$ steps from one of the two wells. However, there may exist some "surplus" cells whose distances to the closest well are between $\text{Max\_Distance} + 1$ and $2 \cdot \text{Max\_Distance} - 1$ after all regular rounds of distance transform are completed. The algorithm uses an extra round of distance transform to convert the distances of these cells. Our algorithm trades off the number of wells for processing time, which needs at most two regular rounds plus one extra round of distance transform to complete each region. Other approaches (such as divide-and-conquer) may place a smaller number of wells but will require more rounds of distance transform. In general, our algorithm gives a good initial approximation for a user to fine tune the
performance of the difference lists. Figure 5.15 shows an example of the distance transform computation.

Figure 5.15 An example of the distance transform computation. Cells not in the region are shown in solid color, while difference edges in the region are shown by short bars. \textit{Max\_Distance} is set to 3. The figures show the status of computation at: (a) the beginning of a new region; (b) the end of the first regular round; (c) the end of the regular second round; (d) the end of the extra round. Small arrows show the direction a cell follows when searching for a well.
5.6 Results

We implemented our algorithm using the Inventor 3D toolkit version 2.0 on a Silicon Graphics O2 workstation available at ACCAD (Advanced Computing Center for the Art and Designs) at the Ohio State University. We use two datasets to test the performance of our algorithm. The Office dataset (Figure 5.13) is a general floor model of 269 objects and 26K triangles, and contains offices, reception area and meeting rooms. The MassHall dataset (Figure 5.16) is constructed by making four copies of the Office datasets which are translated to form a square in the center. It consists of 1117 objects and 104K triangles. Different configurations of these datasets are further identified by their space subdivision resolutions and are named in accordance with the abbreviations listed in Table 5.1, which are used throughout this section. The non-uniform subdivision resolutions in the Office dataset are due to the inherent aspect ratio of the floor plan.

<table>
<thead>
<tr>
<th>Abbrev.</th>
<th>OF15</th>
<th>OC30</th>
<th>OF60</th>
<th>OF90</th>
<th>OF120</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subdiv. Res.</td>
<td>17 x 15</td>
<td>30 x 25</td>
<td>59 x 50</td>
<td>95 x 81</td>
<td>126 x 108</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abbrev.</th>
<th>MH30</th>
<th>MH60</th>
<th>MH90</th>
<th>MH120</th>
<th>MH150</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subdiv. Res.</td>
<td>30 x 30</td>
<td>59 x 59</td>
<td>90 x 90</td>
<td>118 x 118</td>
<td>153 x 153</td>
</tr>
</tbody>
</table>

Table 5.1 Abbreviations for various configurations of the Office (OF) and MassHall (MH) datasets.

5.6.1 Visibility Preprocessing

Table 5.2 shows statistics of the preprocessing stage for each dataset/subdivision combination. Reported times include the computation of both space subdivision and
Figure 5.16 The floor plan of the MassHall dataset. Objects are shown in bounding boxes.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Total Time (sec)</th>
<th>Wall cells/ Avg walls</th>
<th>Data cells/ Avg data</th>
<th>Avg. PVS</th>
<th>Longest PVS</th>
<th>Cull Rate</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>OF15</td>
<td>1</td>
<td>127/2.76</td>
<td>110/1.94</td>
<td>35.31</td>
<td>84</td>
<td>86.8%</td>
<td>32K</td>
</tr>
<tr>
<td>OF30</td>
<td>2</td>
<td>225/2.24</td>
<td>273/1.42</td>
<td>29.78</td>
<td>80</td>
<td>88.9%</td>
<td>78K</td>
</tr>
<tr>
<td>OF60</td>
<td>19</td>
<td>486/1.79</td>
<td>830/1.17</td>
<td>24.98</td>
<td>76</td>
<td>90.7%</td>
<td>256K</td>
</tr>
<tr>
<td>OF90</td>
<td>125</td>
<td>1174/1.13</td>
<td>1881/1.09</td>
<td>21.48</td>
<td>61</td>
<td>92.0%</td>
<td>581K</td>
</tr>
<tr>
<td>OF120</td>
<td>370</td>
<td>1576/1.09</td>
<td>2705/1.06</td>
<td>20.69</td>
<td>60</td>
<td>92.3%</td>
<td>993K</td>
</tr>
<tr>
<td>MH30</td>
<td>6</td>
<td>383/3.32</td>
<td>526/2.31</td>
<td>71.30</td>
<td>170</td>
<td>93.6%</td>
<td>259K</td>
</tr>
<tr>
<td>MH60</td>
<td>39</td>
<td>915/2.21</td>
<td>1453/1.48</td>
<td>42.44</td>
<td>110</td>
<td>96.2%</td>
<td>602K</td>
</tr>
<tr>
<td>MH90</td>
<td>165</td>
<td>1525/1.83</td>
<td>2579/1.25</td>
<td>33.72</td>
<td>93</td>
<td>96.9%</td>
<td>1.12M</td>
</tr>
<tr>
<td>MH120</td>
<td>457</td>
<td>2162/1.61</td>
<td>3850/1.17</td>
<td>30.47</td>
<td>84</td>
<td>97.2%</td>
<td>1.74M</td>
</tr>
<tr>
<td>MH150</td>
<td>1250</td>
<td>3175/1.38</td>
<td>5645/1.12</td>
<td>28.12</td>
<td>81</td>
<td>97.5%</td>
<td>2.71M</td>
</tr>
</tbody>
</table>

Table 5.2 Visibility preprocessing statistics for the Office and MassHall datasets.
visibility using the incremental visibility algorithm. Note that on average it takes less than 7 milliseconds to compute an omnidirectional PVS for a cell in the OF60 and MH30 configurations, which implies that visibility computation can be performed on-the-fly in the walkthrough stage using a low resolution subdivision. The algorithm also shows an excellent cull rate for both datasets (86.8% to 92.3% for the Office dataset and 93.6% to 97.5% for the MassHall dataset). Thus, on average, only 6.7% to 13.2% of the Office model and 2.5% to 6.4% of the MassHall model needs to be rendered from a given cell location. Even though the efficiency of culling is dataset dependent and varies from location to location in model space, the high cull rates achieved in these test cases demonstrate our algorithm’s capability of handling visibility computation in building environments.

The average number of objects in each cell’s PVS reduces as the subdivision resolution increases. This is expected since a higher subdivision resolution allows for increased precision for both wall occlusions and object locations, therefore resulting in more accurate visibility computations. The smaller average PVS sizes imply a faster rendering time during walkthrough in for the higher-resolution configurations. However, since more cells need to store the visibility information, the PVS storage requirement also rises dramatically. For example, while the MH150 configuration shows a reduction of 60% in the average PVS size comparing to MH30, it requires more than 10 times of the PVS storage than MH30! Without an efficient storage scheme, the reduced PVS size may
not justify the increased memory cost in some situations. This problem is addressed in the next section when we present our results for the difference list scheme.

### 5.6.2 Difference Lists

<table>
<thead>
<tr>
<th>Cells</th>
<th>MaxDist</th>
<th>Total time (sec)</th>
<th>Diff edge percentage</th>
<th>Regions</th>
<th>Wells</th>
<th>Avg. PVS</th>
<th>Storage Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>OF15</td>
<td>5</td>
<td>1</td>
<td>40%</td>
<td>90</td>
<td>103</td>
<td>16.01</td>
<td>55%</td>
</tr>
<tr>
<td>OF30</td>
<td>10</td>
<td>1</td>
<td>62%</td>
<td>108</td>
<td>130</td>
<td>7.52</td>
<td>75%</td>
</tr>
<tr>
<td>OF60</td>
<td>10</td>
<td>2</td>
<td>80%</td>
<td>100</td>
<td>287</td>
<td>4.40</td>
<td>82%</td>
</tr>
<tr>
<td>OF90</td>
<td>20</td>
<td>3</td>
<td>86%</td>
<td>164</td>
<td>403</td>
<td>2.83</td>
<td>87%</td>
</tr>
<tr>
<td>OF120</td>
<td>20</td>
<td>6</td>
<td>91%</td>
<td>153</td>
<td>630</td>
<td>2.35</td>
<td>89%</td>
</tr>
<tr>
<td>MH30</td>
<td>10</td>
<td>1</td>
<td>33%</td>
<td>403</td>
<td>407</td>
<td>38.03</td>
<td>47%</td>
</tr>
<tr>
<td>MH60</td>
<td>10</td>
<td>3</td>
<td>58%</td>
<td>566</td>
<td>706</td>
<td>12.78</td>
<td>70%</td>
</tr>
<tr>
<td>MH90</td>
<td>20</td>
<td>10</td>
<td>71%</td>
<td>589</td>
<td>775</td>
<td>6.86</td>
<td>80%</td>
</tr>
<tr>
<td>MH120</td>
<td>20</td>
<td>18</td>
<td>79%</td>
<td>557</td>
<td>970</td>
<td>5.17</td>
<td>83%</td>
</tr>
<tr>
<td>MH150</td>
<td>20</td>
<td>37</td>
<td>84%</td>
<td>537</td>
<td>1241</td>
<td>4.19</td>
<td>85%</td>
</tr>
</tbody>
</table>

Table 5.3 Difference list statistics for the Office and MassHall datasets. The Diff_Threshold is 0.1.

Table 5.3 shows the result of applying the difference list scheme to the preprocessed visibility information shown in Table 5.2. The table shows that between 33% to 91% of the difference edges are computed when Diff_Threshold is set to 0.1, which create a number of disjoint regions (from 90 to 537) and designated wells (from 103 to 1241). In all cases, the average PVS size per cell is greatly reduced, because only the PVS differences are saved at most cells. We are especially interested in the high resolution cases, where the PVS storage reduces to the ratios 89% and 85% for OF120 and MH150 datasets, respectively. This translates to approximately 109 KBytes and 406 KBytes of
PVS storage, respectively, as compared to 32 KBytes for the OF15 dataset and 259 KBytes for the MH30 dataset in the complete list situation. The memory overhead is now relatively smaller for the high resolution cases since the difference list approach reduces the storage cost to a more reasonable range.

![Graph showing PVS storage reduction for dataset MH150](image)

Figure 5.17 PVS storage reduction for dataset MH150 showing various settings of `Max_Distances` and `Diff_Threshold`. The average PVS size is normalized to size when the difference list scheme is not used.

Generally speaking, a higher `Diff_Threshold` implies that more difference edges will be computed, which in turn reduce the number of regions and possibly the number of wells. However, this arrangement does not necessarily reduce memory consumption, as illustrated in Figure 5.17, where the normalized average of the PVS size is graphed when setting `Diff_Threshold` is set between 0 (only two identical cells create a difference edge)
and 1.0 (computes all difference edges). Although this approach has reduced storage requirement when compared with the complete PVS approach, the minimal storage requirement is at a threshold value of 0.1 for the MH150 dataset. This value is dataset dependent and in general should not be greater than 0.5.

The illustration shown in Figure 5.18 is a visualization of the information generated by the difference list scheme. Figure 5.18(a) shows the well locations as generated by our previous heuristic algorithm. The "diagonal line" shaped distribution illustrates the conversion to wells from cells at a fixed distance within each region, especially when the region is a fully connected territory with a regular shape. Figure 5.18(b) shows the region distributions and shapes. With an appropriate \texttt{Diff\_Threshold} value, we can approximately join cells in accordance with the room partition, since different scenes existing on either side of a wall means that cells do not generate difference edges between them. When compared with the floor plan shown in Figure 5.16, most regions indeed conform to the room partition of the original model. An exception is the meeting room in the bottom right Office model, which merges into the region grown from the center square. Further investigation reveals that the wall (which consists of two parallel polylines) that separates the two spaces in that area is entirely contained by the same column of cells. These wall cells serve as a "bridge" between both worlds and can potentially join the two worlds into the same region. Note that these misaligned merges do not affect the efficiency of our difference list scheme. Figure 5.18(c) shows the distribution of PVS sizes among the cells. As expected, the majority of cells record less
Figure 5.18 Visualization of the difference list scheme applied on the MHI150 dataset. The Max_Distance is 20 and Diff_Threshold is 0.1. (a) Region distribution. Regions are color-coded by their region ID. The floor plan of the Masshall is shown here for comparing the regions with the original room partitions. (b) Well distribution (shown as white dots). (c) PVS size distribution. All wells store complete lists, while other cells store only the differences. The color bar shows the mapping scale. Black area represents cells with no stored objects.
than 4 objects in their difference lists, while only a few other cells keep a complete list for later restoration.

### 5.6.3 Walkthroughs

<table>
<thead>
<tr>
<th>Visibility Setting</th>
<th>Dataset</th>
<th>$T_{\text{comp}}$</th>
<th>$T_{\text{restore}}$</th>
<th>$T_{\text{render}}$</th>
<th>Frame Rate (fps)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Visibility Info.</td>
<td>Office</td>
<td>N/A</td>
<td>N/A</td>
<td>0.450</td>
<td>2.2</td>
<td>1.00</td>
</tr>
<tr>
<td>Complete PVS</td>
<td>OF120</td>
<td>N/A</td>
<td>N/A</td>
<td>0.085</td>
<td>11.8</td>
<td>5.36</td>
</tr>
<tr>
<td>Difference Lists</td>
<td>OF120</td>
<td>N/A</td>
<td>0.0005</td>
<td>0.086</td>
<td>11.6</td>
<td>5.27</td>
</tr>
<tr>
<td>Interactive PVS</td>
<td>OF60</td>
<td>0.010</td>
<td>N/A</td>
<td>0.116</td>
<td>7.9</td>
<td>3.59</td>
</tr>
<tr>
<td>No Visibility Info.</td>
<td>MassHall</td>
<td>N/A</td>
<td>N/A</td>
<td>1.470</td>
<td>0.7</td>
<td>1.00</td>
</tr>
<tr>
<td>Complete PVS</td>
<td>MH150</td>
<td>N/A</td>
<td>N/A</td>
<td>0.108</td>
<td>9.3</td>
<td>13.68</td>
</tr>
<tr>
<td>Difference Lists</td>
<td>MH150</td>
<td>N/A</td>
<td>0.0008</td>
<td>0.108</td>
<td>9.2</td>
<td>13.53</td>
</tr>
<tr>
<td>Interactive PVS</td>
<td>MH60</td>
<td>0.023</td>
<td>N/A</td>
<td>0.222</td>
<td>4.1</td>
<td>6.00</td>
</tr>
</tbody>
</table>

Table 5.4 Walkthrough performance (all times are in seconds).

While our original goal is to fully make use of preprocessed visibility information, our efficient algorithm leads itself to interactive visibility computation. We do this by lowering the subdivision resolution of the models and apply visibility computation on-the-fly to interactively decide PVS from the current cell. This is equivalent to a run-time object-level cull operation for the rendering pipeline. Currently our implementation simply computes the omnidirectional visibility from any cell location in which the observer resides in. An optimization can be made to compute visibility only for the view direction and incrementally updates when the observer moves or turns. Other
optimizations may include caching computed results in each cell or predicting user
movement to "look ahead" the visibility computation.

Table 5.4 shows performance results of a random walkthrough of the test models
using various visibility settings. We show $T_{\text{comp}}$ (time to compute visibility on-the-fly),
$T_{\text{restore}}$ (time to restore a complete PVS from a well and the difference lists), $T_{\text{render}}$ (time
to render a frame), and also compute the frame rate and speedup. Our experiment shows
that the performance of the difference list-based visibility method is almost identical to the
complete PVS visibility method, and the interactive visibility computation instantly
boosts the walkthrough performance with quick culling of a significant portion of the
model. For the test models, our algorithm has speedups between 5.27 to 13.68 for
preprocessed visibility and between 3.59 to 6.0 for interactive visibility. Since in our
algorithm rendering performance is associated with the complexity of a local confine, we
expect an even higher speedup for models with higher global complexity and greater
depth distances.

5.7 Discussion

Most previous visibility algorithms for architectural walkthroughs
[3][4][64][2][83][97] utilize the original partitions defined by wall segments and solve the
visibility problem using the portal sequences connecting different spaces. While these
approaches are more general than ours for a 3D building environment, they require the
definition of all portals in model space, either by embedding the information in the model
[64], or by running a subdivision and searching algorithm to automatically identify them [97]. A manual approach to embed the portal information is not intuitive, and may introduce inconsistencies between portal sequences and the actual 3D space. An automatic approach, on the other hand, requires a lengthy preprocessing of the dataset and may restrict wall orientations to be axially parallel for efficiency reason. Compared to these methods, our algorithm is new in that it superimposes a second subdivision grids to allow for a more flexible way of computing the visibility information. The user sets the resolution of the subdivision grid, and in doing so, may trade off accuracy with processing time. This allows for either preprocessing or interactive computation of visibility information. The expensive process of creating and tracing 3D portals is replaced by a fast cell-to-cell visibility test on a 2D floor plan. As was shown in the previous section, the rendering performance in architectural environments instantly improves with even a relatively low resolution subdivision. A general problem for a regular grid subdivision-based visibility algorithm is the storage requirement for all information of each grid cell. Our difference list scheme virtually eliminates this problem by using a very efficient scheme to compress the visibility information. In general, our algorithm is simple, fast, easy to understand and implement, and provides excellent cull rates without complicated 3D computations.

Since we use a 2D floor plan as a basis for visibility computation, 3D portals on a wall, regardless of their sizes and positions, essentially opens up holes on the 2D wall segments. Therefore, our algorithm is not suitable for walls with many small portals, such
as small windows or ventilation holes, since a 2D projection of these portals will practically render these wall segments useless. In addition, the subdivision scheme applied on the floor plan will rarely align properly with the room partitions. Therefore, it will tend to cause more overestimation at cells which contain wall segments that separate two rooms, since both sides of a wall are visible to these cells. However, during the actual walkthrough, we do not expect gross overestimations of these cells to affect rendering performance too much, since they are visited less frequently than other regular cells.

Our approach's dependence on the properties of architectural models (such as 2D floor plans and wall occlusions) implies that it may not be suitable for some general cases, especially in environments without proper orientations or an identifiable group of occluders. In these environments, general visibility algorithms such as Greene [38][39], or Coorg and Teller [99][100], can be used. These algorithms do not assume any special characteristics from a model, as they can use any object of any orientation as an occluder. However, their performance may depend on the order of the dynamic selections of occluders, which determines the effectiveness of the calculated occlusion. Besides, a general visibility algorithm usually requires more run-time overheads, such as processing time and storage for auxiliary data structures, than a specialized algorithm. With a few assumptions of the model, a specialized visibility algorithm like ours can easily achieve better results in its target applications than a general algorithm, while at the same using less resources.
It is possible to extend our visibility computation framework to render 3D volumetric datasets, by using an approach similar to Yagel and Ray [118]. One major problem when extending the algorithm to 3D is the memory required to save the preprocessed information (much more than the 2D case.) We believe that by carefully tuning the difference list algorithm, the memory problem will be manageable. The difference list approach may further be extended to achieve an effect of cell fusion [118], which stores just a single complete PVS for each “fused” region. The implementation results in Section 5.6.2 show that the difference list algorithm is capable of identifying regions with similar visibility information. Therefore, we can perform a space subdivision, process the PVS, apply the difference list algorithm to merge regions, and end up with just a few regions, each with a single, complete PVS.
CHAPTER 6

Conclusion

In this dissertation we have designed two new techniques that tackle the problem of visualizing large datasets. Our first contribution is a new parallel ray casting algorithm, implemented on a distributed-memory multiprocessor system using message-passing mechanism. Parallel ray casting algorithms usually suffer from a high communication cost, due to the fact that the traversal of rays usually leads to an irregular object-space access pattern. Unlike other previous approaches, our algorithm combines the rendering efficiency of an image-order approach (such as early ray termination and adaptive sampling) with the deterministic communication order of an object-order approach. This algorithm totally eliminates the communication bottleneck experienced by most parallel ray casting algorithms, by imposing a front-to-back access order for all object communications. A new fetch list mechanism and a complementary cache management system are designed to take advantage such an order, resulting in great cache efficiency (no cache conflicts or thrashing) and a high degree of latency hiding. To achieve these benefits, the algorithm requires only a small fraction (several hundred kilobytes) of system memory as cache to operate optimally. This is an important characteristic, since most of the processor’s memory may be needed for the large sized dataset.
Our new parallel algorithm is very efficient, capable of rendering 21 frames per second for a volume of $256^3$ voxels and one frames per second for a volume of $512^3$ voxels on the Cray T3E using 128 processors. It is also a very scalable algorithm, reaching 80% of parallel efficiency for a volume of $512^3$ voxels on 128 processors. The algorithm can also be extended to render multiple images from a static scene simultaneously. Such a "multiframe" extension can further improve the rendering efficiency by taking advantage of the frame-to-frame cache coherence across multiple image frames. Our results show an improvement from 5.8% to 40% when applying a five-frame multiframe rendering on various datasets on the Cray T3E.

The second innovation in this dissertation improves the efficiency in rendering large architectural models with visibility computation. Our method is especially suitable for architectural models because it takes advantage of various properties of buildings (i.e., connected, vertically-standing wall segments.) It is a conservative, but fast, object-space visibility determination algorithm, which efficiently culls away large portions of the model from any given viewpoint. Our algorithm uses a 2D floor plan to represent a 3D architectural model, especially for the partitioning structure of rooms. A 2D regular grid is superimposed on the floor plan by applying a space subdivision, and visibility information is computed based on subdivided regions by using walls as major occluders. Our algorithm is designed to be used as a preprocessing step to compute and save the visibility information in advance. To reduce the storage requirement for storing the visibility information, a difference list scheme is developed, which employs a mechanism to save
only the differential visibility information between neighboring regions, and provides a quick restoration (with user-adjustable parameters) to the complete visibility during runtime. With the difference list scheme, visibility information can be stored at only a small percentage (about 11% to 15%) of the full set's memory cost, allowing a space subdivision of higher resolutions to be used in the computation (and thus more accurate visibility information). Our results demonstrate a speedup in rendering performance between 3.59 to 5.36 for a simple dataset (Office) and between 6.00 to 13.68 for a more complex dataset (MassHall).

A few future research directions are listed below. One promising problem is to design a visibility-assisted parallel volume renderer. Our current parallel ray casting implementation does not require such assistance, since the ray casting itself is one form of visibility processing. However, for object-parallel methods, the lack of early ray termination results is a waste in computation power. One possible solution is to store a scaled-down cell volume on each processor, and encode information, such as the average opacity or min/max opacity, within each cell volume. Each processor applies a 3D visibility computation to determine the "visible" candidate cells, and then redistribute or fetch cells for local rendering. This is similar to the hybrid approach discussed in Yoo, et al. [121]. However, since all processors have global visibility information, a better redistribution and load balancing scheme than Yoo, et al. [121] can be achieved. Other fruitful research directions lies on improving or enhancing the current algorithms, which include the following lists.
For the ART algorithm:

- Implementing an efficient perspective-projected ART algorithm.
- Extending the ART algorithm to ray tracing.
- Enhancing the capability of the multiframe rendering to handle dynamic scenes.
- Exploring possible solutions to reduce ray memory.

For the visibility computation algorithm:

- Extending the algorithm to a general, 3D polygonal environment.
- Extending the algorithm to a 3D volumetric environment.
- Exploring other space subdivision methods to adapt the subdivision grid to the dataset.

By exploiting both parallel processing and visibility computation, the methods in this dissertation combine the power of hardware acceleration and software optimization to approach the challenging problem of visualizing large datasets. We demonstrate a feasible way to bring the interactive rendering of large datasets close to reality. As the processing power of parallel computers continues to increase, the goal of reaching real-time performance for large dataset rendering will be achievable in the foreseeable future.
Bibliography


152


