View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarBank@NUS

QUERYING LARGE VIRTUAL MODELS FOR INTERACTIVE WALKTHROUGH

Shou Lidan

NATIONAL UNIVERSITY OF SINGAPORE

2002

QUERYING LARGE VIRTUAL MODELS FOR INTERACTIVE WALKTHROUGH

Shou Lidan
(M.Eng., Zhejiang Univ.)

A THESIS SUBMITTED

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY *of* SINGAPORE

2002

Acknowledgements

I would like to acknowledge the enthusiastic supervision of my Ph.D. supervisors, Prof. Tan Kian-Lee and Dr. Huang Zhiyong, during my work. This thesis would not, indeed, have been materialized without their guidance. They have provided so much advice, support, and encouragement with enthusiasm, inspiration, insight, and great efforts, during the long term of my study.

To the members of the thesis committee, Prof. Ooi Beng Chin, Prof. Lee Mong Li, Dr. Stéphane Bressan, I would like to thank them for their advice, comments, and suggestions. Their contributions to the improvements of this work are essential and greatly appreciated.

I would also like to express my deep thankfulness to the School of Computing, National University of Singapore, for supporting my research with the Research Scholarship, and the University Graduate Fellowship. I appreciate its recognition of my thesis contributions by awarding me the Dean's Graduate Award.

I sincerely appreciate the assistance and helpful comments from the other members of my research team. They are Mr. Chionh Chern Hooi and Mr. Ruan Yixin. It has always been an enjoyable experience to work with them.

To all the members of the EC and Database Laboratories, I would like to acknowledge them for our friendship and help in all aspects of the study and life here. They have made the working place more lively and lovely.

Lastly, and most importantly, I wish to thank my family and in-laws for their love, understanding and support. To them I dedicate this thesis.

Contents

Acknowledgements **i**

Contents **ii**

List of Figures **vi**

List of Tables **x**

Summary **xi**

1 Introduction **1**

1.1 Motivations 3

1.1.1 Spatial Techniques 4

1.1.2 Visibility Techniques 5

1.2 Research Contributions 7

1.3 Organization Of The Thesis 11

2 Background and Related Work **14**

2.1 Walkthrough of Virtual Environment 14

2.2 Spatial Access Methods and Data Structures 18

2.2.1 Spatial Access Methods 18

2.2.2 Spatial Techniques in 3D Computer Graphics 24

2.3 Computing Visibility of 3D Objects 26

2.3.1 Back-face Culling 26

2.3.2 Occlusion Culling 28

2.3.3 Point Based Visibility 31

2.3.4 From-region Visibility 36

2.3.5 Computing Visibility Using Graphics Hardware 42

2.4 Multi-resolution Representations 44

2.5 Summary 45

3 Walkthrough System Architecture For Large VE **47**

3.1 The Scene Graph Structure 48

3.2 Generic System Architecture for Disk-Based VE 50

3.2.1 Components of Disk-Based Walkthrough System 51

3.2.2 How Things Work 55

3.3 Data Generating 56

3.4 Summary 58

4 Virtual Walkthrough Using Spatial Techniques **59**

4.1 Overview of Our Techniques 61

4.2 Optimizing I/O Performance 64

4.2.1 Spatial Index of The Data Set 64

4.2.2 Complement Search Algorithm of R-tree 66

4.2.3 Regular Grids vs. ‘R-tree + CSearch’ 74

4.2.4 Distance-priority-based Replacement Policy 76

4.2.5 Prefetching Algorithm 80

4.3 Optimizing GPU Performance 82

4.4 Experimental Results 85

4.4.1 Tuning the parameters in REVIEW 89

4.4.2 Performance improvements of REVIEW 92
4.5 Summary 97

5 Virtual Walkthrough Using Visibility Techniques 99

5.1 Overview of Our Techniques 102
5.2 The Logical Structure of HDoV-Tree 103
5.2.1 Degree of Visibility 103
5.2.2 HDoV-Tree Structure 107
5.3 Search Algorithm 110
5.4 Storage Schemes for HDoV-Tree 116
5.4.1 The horizontal storage scheme 117
5.4.2 The vertical storage scheme 118
5.4.3 The indexed-vertical storage scheme 121
5.5 Computing DoV 122
5.5.1 Testing Conventional Visibility 123
5.5.2 DoV of Individual Objects – An Image-Space Approach . . . 124
5.5.3 DoV of Internal Nodes 129
5.6 LODs in HDoV-Tree 130
5.7 Caching The HDoV-tree Nodes 131
5.7.1 The DoV Cache Replacement Policy 132
5.7.2 The DoV-Time Cache Replacement Policy 133
5.8 Performance Results 133
5.8.1 Implementation and Experimental Setup 133
5.8.2 The storage cost of the storage schemes 136
5.8.3 Experiment 1: On caching 137
5.8.4 Experiment 2: On visibility queries 142
5.8.5 Experiment 3: on interactive walkthrough 146

5.9 Summary 154

6 Memory Based HDoV Scene Tree **155**

6.1 HDoV Scene Tree 156

6.1.1 The Basic Traversal Algorithm 157

6.1.2 Polygon Budget Traversal Algorithm 157

6.2 Implementation 159

6.2.1 Precomputation 161

6.2.2 Run-time Visualization 162

6.3 Performance Results 162

6.4 Summary 168

7 Conclusion **172**

7.1 Summary Of Thesis Contributions 172

7.2 Future Work 177

Bibliography **179**

List of Figures

2.1 Back-face culling by testing dot product 27
2.2 A simple example of occlusions at a given viewpoint 29
2.3 Occlusion in 2D [18]. Separating and supporting planes of an occluder A and an occludee T. 32

3.1 A Simple Example of Scene Graph 49
3.2 A Generic System Architecture For Disk-Resident Virtual Walkthrough 51

4.1 An example of R-tree 65
4.2 An example to motivate complement search. 66
4.3 An example of the nth complement query. 67
4.4 The TentativeComplementSearch algorithm. 68
4.5 The CSearch algorithm. 69
4.6 Complement overlap relations. 70
4.7 Comparing CSearch with TentativeComplementSearch algorithm . . 72
4.8 Objects to be kept in memory. 73
4.9 Example to illustrate the number of swaps needed for rigid spatial partitions and R-tree indexing. 74

4.10 Example to illustrate the amount of data to be loaded into memory for rigid spatial partitions and R-tree + CSearch. 76
4.11 Prefetching objects 81
4.12 Culling the object buffer using the view frustum. 83
4.13 Spatial relations between a convex shape H and a rectangle R. . . . 84
4.14 Intersection checking with vertices or R. 85
4.15 The view culling algorithm. 86
4.16 Screen shots of a large cityscape. 87
4.17 The effect of the prefetching factor k to system performance 90
4.18 Cache performance with various cache sizes 91
4.19 Average frame time for traditional and optimized systems 93
4.20 Variances of average frame time 94
4.21 Rendering time for each frame 95
4.22 Average search time 96

5.1 Comparison of Visual Quality For LODs at Different Degree-of-Visibility. Note the difference of bunny between (e) and (g), while (a) and (c) are almost identical. 105
5.2 Dynamic update of the HDoV-tree. 108
5.3 A Hierarchical Degree-of-Visibility Tree. 109
5.4 The HDoV-tree traversal algorithm 113
5.5 Examples of when to use internal LOD in the HDoV-tree 115
5.6 Horizontal Storage Scheme. *VPage i,j* represents the V-page of node j in cell i. 117
5.7 Vertical Storage Scheme. 119
5.8 Indexed Vertical Storage Scheme. 121
5.9 Querying occluders around viewing region 124

5.10 Spherical projection vs. cubic projection 125
5.11 Computing spherical projection for ΔA 126
5.12 Computing the spherical projection of ΔA (2D). 126
5.13 Steps to build a HDoV-tree. 135
5.14 Bird's eye view of the default dataset 136
5.15 Cache Hit Rate For *Continuous Queries* Using $\eta = 0.00001$ 138
5.16 Cache Hit Rate For *Continuous Queries* Using $\eta = 0.0001$ 138
5.17 Cache Hit Rate For *Continuous Queries* Using $\eta = 0.0005$ 139
5.18 Cache Hit Rate For *Random Queries* Using $\eta = 0.00001$ 141
5.19 Cache Hit Rate For *Random Queries* Using $\eta = 0.0001$ 141
5.20 Cache Hit Rate For *Random Queries* Using $\eta = 0.0005$ 142
5.21 Search time with different η values 143
5.22 Performance results on disk I/Os 145
5.23 Scalability of the visibility query performance 147
5.24 Comparison of frame time 149
5.25 Comparison of frame time between VISUAL($\eta = 0.001$) and VISUAL($\eta = 0.0003$) 150
5.26 Comparison of Visual Fidelity. Far objects are lost in (b) due to the spatial method. The visual fidelity of VISUAL system is very good even if the threshold η is as large as 0.001 151
5.27 Search Performance in Different Walkthrough Sessions 153

6.1 A Memory HDoV Scene Tree 156
6.2 The HDoV refine algorithm in Polygon Budget Mode 160

6.3 Top view of nodes being rendered in VSC mode. The white lines show the position of the view frustum. The yellow boxes bound internal nodes where the traversal terminates; the red boxes bound leaf nodes (individual objects); the green boxes bound nodes culled by VSC culling or view frustum culling. 164
6.4 Number of polygons rendered in each frame (VSC mode vs. CONV mode). Note the *logarithm y-coordinate*. 165
6.5 Frame time of the same walkthrough (VSC mode vs. CONV mode). 165
6.6 Snapshots of VSC and CONV modes. Note the very-low details of the active internal nodes on top-left corner of (a), which are represented by the yellow boxes in (b). (b) and (d) show the difference in the number of nodes being rendered. Although (a) looks as if containing more objects, it actually contains much fewer geometries than (c). 169
6.7 Number of polygons rendered in Polygon Budget Mode. 170
6.8 Snapshots of PB modes. Note the low detail of the building on the left in (a), the missing bunny in (a) and (b), and the missing building in the center of the screen in (a,b,c). 171

List of Tables

3.1 List of base models in the synthetic data sets 57

4.1 Disk I/Os Per Query 96

4.2 Results of view frustum culling 97

5.1 Some notations. 117

5.2 Storage space required by various schemes. Data in the table include V-page data and V-page-index structures only. 137

5.3 Results of frame time 152

6.1 Run-time performance of the same walkthrough path by different DoV thresholds. 166

Summary

Real-time walkthrough of Virtual Environments (VE) has been considered as an important application to provide high-quality interaction between user and computer. Existing techniques in the computer graphics areas such as *visibility culling*, *Level-of-Details*, etc. have been proven to be effective in dealing with complex scenes, provided that the scene can be loaded into main memory as an entirety. For very large datasets which cannot be completely memory-resident, new techniques have to be designed in order to (1) facilitate storing, manipulating, and retrieving of data on secondary storage; (2) provide optimization support for real-time rendering. This thesis addresses these two issues for real-time walkthrough of large Virtual Environments. The techniques proposed in the thesis are divided into two major parts: the spatial techniques and the visibility techniques.

For the spatial solution of real-time virtual walkthrough application, we identify three main bottlenecks of a very large VE that is stored on disk. We propose a method based on spatial techniques to solve the problems. The method maintains the data in two representations: the disk resident data which is organized in a spatial database and the scene graph structure which is optimized for graphical rendering. We choose the R-tree as an example of the spatial index structure for organizing the data on disk. The search interface of the spatial database is optimized to capture the search patterns of the virtual walkthrough. Algorithms for caching and prefetching are also based on the walkthrough semantics. The in-

memory data are filtered using a frustum culling algorithm to gain higher speed. A prototype walkthrough system, named REVIEW, was implemented using the above techniques. Performance of the REVIEW system was studied and compared with the conventional R-tree approach. The experimental results show that the system performance can be tuned by several parameters. The results also clearly show the benefits of applying the novel search interface, as well as the other optimization techniques.

For the visibility techniques, we propose a novel data structure called hierarchical Degree-of-Visibility tree (HDoV). This data structure, based on the Degree-of-Visibility, captures the visibility data and the hierarchy of level-of-details in a spatial structure. The HDoV-tree is distinguished from spatial data structures such as R-tree in several ways. First, the HDoV-tree is a template structure which needs to be updated for different viewpoints. Second, traversing the HDoV-tree is based on the Degree-of-Visibility rather than spatial information. Third, the HDoV-tree is tunable. We present three different storage schemes for the implementation of the HDoV-tree on secondary storage. We also propose two replacement policies for the cache of the HDoV-tree nodes to save disk accesses. Extensive performance studies were conducted to reveal the precomputational and run-time attributes of the HDoV tree. The experiments show that the HDoV-tree provides tunable and faster query performance compared to conventional method which is based on a linear-list approach. The HDoV-tree can also provide faster and smoother frame rates and better visual quality compared to a spatial method.

The memory version of the HDoV-tree is also studied in this thesis. For the memory HDoV-tree, we propose a new *polygon budget* traversal strategy to provide guaranteed performance in the rendering. The threshold-based traversal algorithm can also work on the memory HDoV scene tree. Performance results show that

the polygon budget algorithm provides control of the system performance while optimizing the visual quality of the output. The comparison between the results of a conventional visibility culling algorithm and the threshold-based traversal algorithm also show that the latter can control the visual quality of the rendering and improve the rendering performance.

Chapter 1

Introduction

Virtual Reality (VR), also known as *Artificial Reality* or *Cyberspace*, is an artificial environment that can be experienced through sensory stimuli created by a computer. The Virtual Reality technology has found its applications in various areas in the human society. These include military or medical training, education, design evaluation, architecture walkthrough, simulation of tasks, assistance for the handicapped, study and treatment of psychological diseases, gaming, and much more [42, 85, 61, 23].

The VR technology has become increasingly popular primarily due to the decreasing computer costs and the booming processing power which have made VR available on personal computers [56]. As an efficient tool for supporting human-computer interaction, VR offers great benefits to the end-users.

A very important advantage of applying VR to a model is that no physical model needs to be constructed in reality. As a good example of VR applications, the Institute of High Performance Computing (IHPC) of Singapore has applied VR technol-

ogy to investigate the ventilation performance as well as conduct fire and smoke control simulations for the building in *The Esplanade — Theaters on The Bay* project, even before the installation stage, thus reducing time-consuming and hefty adjustments and re-working after the installation phase (http://www.ihpc.nus.edu.sg/). The VR technology saves not only the construction materials, but also human work, and even the global environment as well!

The VR applications can also generate special effects which could not be possible in the real world. For example, a user may virtually "walk" in a 3D model of the human body to learn the anatomical structure [1]. Without the VR technology, such kind of "telepresence" would have been completely impossible. In some dangerous environments, such as a nuclear power plant with fatal radiation, virtual reality system enables an engineer to browse the environment without being physically present in the scenario.

In this thesis, we examine a typical class of VR application, namely *interactive virtual walkthrough* systems. In these systems, user interacts with the system to "move" from scene to scene in the virtual space. As such, if the system performs poorly and produces "choppy" frame rate to the user with many "pauses" in the graphics output, the quality of the visual feedback to the user may become unacceptable. Therefore, it is an essential problem for high-quality Virtual Reality systems, in particular, *virtual walkthrough* systems, to provide high performance and fast response time. Designing efficient and enjoyable virtual walkthrough systems with high and constant frame rates is still a challenge, especially for low-end hardware platforms such as PCs.

1.1 Motivations

This thesis addresses the design of effective and efficient interactive VR walk-through systems for *very large-scale* virtual environments. In a large-scale virtual environment, the vast amount of data in the model are too large to fit into the main memory in its entirety. Thus, data need to be retrieved dynamically on demand. Under such circumstances, the system I/O cost may become very high and cause serious slow-down in the frame rates. Therefore,the design of an optimized swapping engine, which selects appropriate data to be loaded into memory, is critical for the overall performance. Specific techniques have to be deployed to ensure high-quality user experience.

Basically, the techniques to address the above issues concern data partitioning /clustering and dynamic loading. With regard to the specific data clustering schemes, we can classify such techniques into two categories: spatial techniques and visibility techniques. Spatial techniques usually partition the dataset according to spatial proximity, therefore objects which are spatially close are clustered together and treated as entities. In contrast, visibility techniques honor visibility rather than spatial proximity. In visibility solutions, objects that are visible to viewpoint(s) are clustered together and handled as entities. In the following subsections, we discuss in greater details the motivations as well as the issues that are of interest to us.

1.1.1 Spatial Techniques

Spatial database techniques have been extensively studied. However, very few works have been done to incorporate spatial database techniques in real-time interactive visualization applications. In interactive visualization applications, a single delay caused by the database query may impair the visual quality significantly. In a virtual walkthrough application with spatial data on secondary storage, the major bottleneck for real-time performance is the disk I/O.

A good example of the existing work is [47], where the dataset is organized by an R-tree. When the visualization system queries the spatial data engine, one or more range queries are issued to retrieve the relevant data (usually spatially close to the user in the virtual world). If the user moves in the virtual space, new queries need to be issued to keep the memory buffer of the 3D scene up-to-date. The query regions in conventional spatial databases are usually regular shapes such as n-space rectangles or spheres, so that the range search, normally traversing a hierarchical data structure, could be fast and efficient. Those kind of queries, however, may lead to large overlaps of query regions in consecutive queries in a virtual walkthrough, causing severe performance problems. Another problem with the walkthrough on spatial data structure is that the semantics of the walkthrough path is not clearly described in nature. In other words, there are often spatio-temporal coherences to be exploited. Hence, the system needs a walkthrough-semantics interpreter to optimize the data access method.

Memory caching is a typical technique to reduce the I/Os in a disk retrieval system. We observe that a large portion of the spatial data structure and the

objects can be cached in memory in a virtual walkthrough. As an example, a user may often "turn around" or walk backwards in a virtual space to inspect something that he/she has just missed. In such cases, caching the spatial index and the objects can help saving the disk I/Os and further improving the search performance.

Manipulation of in-memory buffer of the objects seems to be crucial for the performance. The visual performance of a virtual walkthrough system depends on the complexity of the geometries being rendered. Since the data cached in the memory may not be relevant to the current viewing process, it is important to discard these data during the rendering process.

Motivated by the aforementioned observations, we are interested in answering the following questions:

- Can spatial data structures be used to organize, manipulate and retrieve the data of a *very large* Virtual Environment?
- How can we improve the search performance of the spatial database regarding the virtual walkthrough?
- How can we improve the user experience in terms of overall frame rate?
- How effective can the spatial solution be?

1.1.2 Visibility Techniques

Besides spatial techniques, another method to design efficient interactive walkthrough systems is to employ visibility techniques. The visibility techniques can address a few problems that spatial data structures may not adequately solve. For

example, the spatial data structures do not capture the visibility information, so a query to a spatial data structure may miss some visually important results which are spatially distant from the viewer. The visibility computation has received much attention from the graphics community [19]. Most of the work in the literature proposes various algorithms to compute the set of polygons which are visible from a given viewpoint or a viewing region. Computing the visibility of a scene is a hard problem, as noted by Cohen-Or et al [16]. Existing visibility algorithms can be categorized as *precomputed* or *computed on-the-fly*. For a very large dataset that cannot be loaded entirely into memory, computing on-the-fly may be too expensive and unnecessary, if the data do not change over time. Therefore, we consider *precomputed* visibility algorithms only, since the models in the datasets in our very large Virtual Environment seldom change by time.

Many visibility algorithms calculate the visible sets in a *precomputation phase*, and use the results of the precomputation in real-time visualization. Most of these algorithms assume that the whole database can be loaded into the memory. However, when the database is too large to be held in memory, new techniques have to be employed to support the swapping of visibility data, as well as the spatial data and objects.

As the visibility data are dependent on the viewpoint, the data in different viewing regions are often segmented into *viewing cells* and dealt with as entities. Manipulating the visibility data in different viewing regions is also an issue that is of interest to us.

Furthermore, using the visibility data, we can better apply the multi-resolution

(Level-of-Details) technique. We observe that in the previous work, the level-of-details assigned to an object is usually determined by the spatial distance, no matter if it is only visible by a small portion. If the level-of-details can be affected by the extent that the object is visible, we can save disk I/Os and rendering cost by using low details for visually unimportant objects. Also, we would like to explore the possibility of imbuing the visibility data into the spatial index, so that the overall data could be more nicely organized and manipulated.

We are interested in answering the following questions with regard to the above issues:

- How can we manipulate and query the visibility information as well as the object data in a large database of 3D models?
- How efficient is the approach?
- What is the storage cost of the approach?
- How does it compare with a spatial technique?

1.2 Research Contributions

The primary objective of this thesis is to devise effective and efficient methods that organize, process, and optimize a database of a very large virtual environment, so as to facilitate real-time virtual walkthrough in it. The research mainly focuses on designing data structures and algorithms that process and optimize the data structures. The main contributions are as follows:

- **System Framework for Large Database of Virtual Environment**

  We develop a generic system framework for a database of a very large virtual environment. The system framework is designed based on the object-oriented methodology. Using the system framework, we can design various data structures and algorithms in it, and conduct performance studies on the algorithms.

- **Search Algorithms for Data Structure Based on Spatial Techniques**

  We design a novel search algorithm to retrieve data from very large database of the Virtual Environment, which is stored in a secondary storage. The search algorithm, which is based on the spatial index of the data set in three-dimensional space, has the advantage of being able to capture the walk-through semantics. The results of applying the algorithm are: (1) the search performance can be dramatically improved; (2) the memory used by the system is reduced; (3) real-time virtual walkthrough can be implemented efficiently.

- **Optimization Techniques for The Spatial-index Based Access Method**

  We develop techniques such as: (1) Prefetching algorithm which activates retrieval of data from the secondary storage before they are required by the walkthrough; (2) Caching technique which takes spatial locality, index structure, and access history into consideration; (3) Spatial culling technique which restricts data sent to the graphics system in order to improve rendering performance.

- **A Hierarchical Degree-of-Visibility Data Structure**

  We design the novel Hierarchical Degree-of-Visibility (HDoV) data structure. The HDoV-tree has the topology of a generic hierarchical spatial subdivision, and captures the geometric and material data, as well as the visibility data, in the nodes of the tree. The HDoV-tree can be traversed using the visibility metric — the degree-of-visibility. Retrieval of the level-of-details (LOD) from the database can also be determined from the visibility metric. The retrieval algorithm uses a DoV threshold when determining the recursive search path. During the recursive traversal, if a node of the HDoV-tree has a degree-of-visibility greater than the threshold, the traversal will proceed to the child nodes, otherwise, a lower-detailed LOD might be retrieved instead. We propose three storage schemes, namely the Horizontal, Vertical, and Indexed-Vertical Schemes, for the disk implementation of the HDoV-tree and analyze the storage costs respectively. We also present an image-space approach to precomputation of the DoV values of the objects.

- **Implementations and Experiments of The Spatial-index Based Walkthrough System and The HDoV-tree Based Walkthrough System**

  We implemented the spatial-index based walkthrough system named REVIEW, and the HDoV-tree based system named VISUAL. We conducted extensive experiments on the two systems to investigate the performance. Experiments of the REVIEW system show that: (1) The spatial-index can be used to organize the data in a very large VE; (2) The complement search algorithm can effectively remove the overlaps between spatial query regions;

(3) The prefetching algorithm can be tuned by a parameter and achieve the best performance, at a specific value; (4) The caching replacement policy helps to improve the search performance and is better than the conventional LRU policy.

The experiments of the search algorithm of the HDoV-tree show that: (1) Designing effective and efficient storage structure of the HDoV-tree is important for saving storage space and achieving high performance; (2) The DoV-threshold retrieval algorithm outperforms the conventional retrieval algorithm in terms of search performance, as well as the I/O cost; (3) The scalability of the search performance is good.

The interactive walkthrough experiments show that: (1) The HDoV-tree outperforms the spatial-technique based REVIEW in terms of frame rate, variance of frame time, and visual quality; (2) The search performance of the HDoV-tree is tunable given different DoV threshold values; The cache experiments reveal that the proposed cache algorithms are very promising when compared with the conventional LRU algorithm, in particular for small and medium cache size.

- **The HDoV-tree in Main Memory is Tested**

  For memory resident scene tree structure, the number of polygons being processed in each frame can be used as a metric for the system resources. When the HDoV-tree is loaded into main memory in its entirety, we can execute another traversal algorithm, known as the *polygon budget* algorithm, to guarantee the system performance. The performance study shows that the polygon

budget algorithm provides control to the system performance while optimizing the visual quality of the output. With a pre-defined polygon budget, the system can achieve performance-guaranteed interactive walkthrough.

1.3 Organization Of The Thesis

The remainder of this thesis is organized as follows. In chapter 2, we review some of the related work in the literature and describe the research background of the thesis. This chapter consists of four parts. The first part contains some previous work in real-time virtual walkthrough. The second discusses the approaches and backgrounds in some spatial data structures and algorithms. The third introduces backgrounds in visibility and some visibility algorithms. The fourth part reviews some previous work in multi-resolution models.

In Chapter 3, we depict a generic system architecture for virtual walkthrough of a very large graphical database. We highlight the generic scene graph structure which is the main memory structure used by the graphical rendering process. We introduce the components of a generic walkthrough system and describe how these components with different functions interact with each other. We also discuss the method that we use to generate the data sets for the experiments.

Chapter 4 describes our approach to apply spatial techniques to a large database of virtual environment. We present the search algorithm and optimization techniques based on the R-tree. We introduce a novel complement search algorithm, a cache replacement policy and a prefetching scheme. We study the performance of our method, fine-tune its parameters, and compare it with a conventional search

algorithm. Performance results of the experiments are also presented in this chapter.

Chapter 5 describes a novel data structure, called the Hierarchical Degree-of-Visibility tree structure. We discuss the problem with spatial techniques and present the logical structure of the HDoV-tree. We introduce a novel metric called the degree-of-visibility and depict the HDoV-tree structure following it. We also present a search algorithm of the HDoV-tree which is controlled by a DoV threshold. Next, we propose three possible storage schemes of the HDoV-tree. In particular, we compare the storage space and search performance of each storage scheme. The approach to computing the Degree-of-Visibility and generating the HDoV-tree structure is discussed in detail. We also consider the problem of caching the nodes of the HDoV-tree in memory with some novel caching replacement policies. Finally, we present the experimental results of the performance of this novel data structure compared with conventional visibility approach and the spatial approach in terms of search performance and virtual walkthrough.

In Chapter 6 we propose the memory based Hierarchical Degree-of-Visibility tree, or the HDoV scene tree. The threshold-controllable traversal algorithm is further complemented by a polygon budget algorithm which can provide performance-guaranteed walkthrough by allocating polygon budget among the nodes of the scene tree. We also study the performance of the algorithms in the memory HDoV scene tree and report the results.

Finally, we conclude the thesis in chapter 7. We summarize the main contributions of this thesis and propose some promising directions for future work.

Some of the work described in this thesis has resulted in a few technical articles in database [78, 79, 81, 66] and graphics [77, 80] conferences. Paper [78], [79], and [77] present our work in spatial techniques, while [81] and [80] discuss the visibility techniques in this thesis.

Chapter 2

Background and Related Work

In this chapter, we provide an overview of the background of our research, as well as some previous work related to real-time virtual walkthrough. Firstly we will examine the background of virtual walkthrough systems in section 2.1. Secondly we will discuss several approaches and backgrounds about some spatial data structures and their algorithms in section 2.2. We also review previous work in visibility computation in section 2.3. Furthermore, we review related work in multi-resolution representations of virtual models in section 2.4. Finally, we summarize this chapter in section 2.5.

2.1 Walkthrough of Virtual Environment

Generally, a virtual walkthrough system refers to a real-time visual navigation system which can be used to "walk" from one place to another continuously. Users can explore and then have a good knowledge of the scene by walking through the

scene interactively. Given the scene of a virtual world, such as a city, a terrain, or a forest, the user of a walkthrough system should be able to navigate in it with the control of an input device, for example a mouse or a keyboard.

Virtual walkthrough has already been deployed in many applications of industry or academy, for example, in an architectural walkthrough or simulation of the space exploration [26, 91]. In both cases, the user doesn't have to be physically "there". For such VE to generate acceptable effects to the users, the virtual walkthrough system that provides the interaction must output high and constant frame rates [25].

The model that a user perceives during a virtual walkthrough, or more generically, a real-time visualization, is often referred to as a *scene*.

The scene, which is usually represented in a *polygonal model*, is rendered by the graphics engine as viewed from a specific *viewpoint*, as if the user's eyes are located at it. More importantly, in 3D graphics, the three-dimensional region that a user can see is bounded by six clip planes, known as the *left, right, top, bottom, near*, and *far* clip planes, enclosing a frustum-shaped space called *view frustum* [89].

Given a large scene represented in polygonal models with geometries, colors, and textures, it is very common to capture it in a spatial hierarchy with hierarchical bounding volumes. Such a hierarchy can be in the form of a k-D tree, an octree or quadtree (for 2D case) [70], or other structures. Sometimes, as some of the scene objects can share the same geometry or material attributes, nodes in the spatial hierarchy can share a child node to save storage space and rendering time. Such spatial hierarchy is also known as a *scene graph*.

There have been many successful implementations of virtual walkthrough systems, which run at interactive frame rates with large and complex scenes.

An example is the Soda Hall Walkthrough project in U.C. Berkeley [26]. This work described techniques for managing large amounts of data during an interactive walkthrough of an architectural model. The model was subdivided using a variant of the k-D tree. Culling was applied to cell-to-cell, cell-to-objects, eye-to-cell, and eye-to-object visibilities. The researchers reported that they were able to cull away an average of 97% of the building model and reduce rendering time by an average factor of 39 in each frame. As the computation is based on a cell-to-cell and cell-to-objects visibility, the technique is restricted to internal walkthrough of architectural models.

Aliaga et al. [2] developed an interactive massive model rendering system to navigate in very complex 3D models at interactive rates. The fundamental idea in this system is to render objects that are far from a viewpoint using fast image-based techniques and to render all objects that are near the viewpoint as geometry using level of details and visibility culling. The system successfully accelerated walkthrough of a 13 million triangle model of a large coal-fired power plant and of a 1.7 million triangle architectural model. This research is an original work in image based rendering for far geometries. We shall use polygonal simplification instead of image based rendering in our work.

Chim et al. [11] discussed caching and pre-fetching of virtual objects in distributed virtual environment. A multi-resolution modeling (which has also been presented in detail in [52]), as well as a caching scheme were proposed in [11]. Three

pre-fetching schemes: mean, window, and EWMA were discussed and compared. Unfortunately, these results were based on simulations, so the effectiveness of the method has not been practically demonstrated.

Many other systems deployed various techniques such as: the *extended projections* by Durand et al [20], *streaming of complex 3D scenes* in a network environment by Teler et al [84], the *HLODs* for dynamic environment by Erikson et al [22], the GIS index for VR database by Pajarola et al [60], the image caches and ray casting technique by Wimmer et al [90], the *occluder fusion* technique by Wonka et al [92], the *hierarchical image caching* technique by Shade et al [76], the temporally coherent visibility technique by Heo et al [38], the *visibility octree* by Saona-Vázquez et al [71] etc.

The main techniques deployed in these systems are listed as follows:

- Spatial Techniques
- Visibility Techniques
- Levels of Details
- Image Based Rendering
- Spatio-temporal Coherence

In the following sections, we will only look at the first three techniques in detail.

2.2 Spatial Access Methods and Data Structures

The *multi-dimensional access methods* have attracted a lot of attention from the database community for over fifteen years. A lot of work in multi-dimensional access methods is devoted to *spatial access methods*, which differ from *point access methods* in that the first handle the so-called *extended objects*, such as rectangles or polyhedra [27]. Most spatial access methods evolved from point access methods by using one of the following four techniques [27]:

1. transformation (object mapping);
2. overlapping regions (object bounding);
3. clipping (object duplication);
4. multiple layers.

In the remainder of this section. We review some spatial access methods. And we also look at some spatial techniques in graphics.

2.2.1 Spatial Access Methods

Spatial access methods have already been widely studied in various contexts [69, 8]. The transformation technique usually transforms objects into a different representation. For example, an object can be transformed into a higher-dimensional point, or be transformed into a set of one-dimensional intervals. The transformation technique has been adopted in the hB-tree [55] and the P-tree [44].

The overlapping-regions technique enables data buckets to correspond to overlapping subspaces, therefore allowing extended objects to be assigned directly to a single bucket region. The R-tree [37], and the R*-tree [5], as well as the JP-tree [44], the *spatial* kD-tree [58], the GBD-tree [57], and the PLOP-hashing [74, 49] are examples of this technique. In the overlapping-regions technique, as the overlap among regions may pass the same search criteria, point queries and region queries may require access to multiple search paths and, therefore, harm the search performance. This problem has been noted in the R-tree [37]. Several techniques have been proposed to minimize the overlap [65, 7, 3].

Unlike the overlapping-regions technique, the clipping technique only allows disjoint buckets. If one object spans over more than one bucket, it is clipped into the corresponding buckets. Therefore, data of the object are either segmented or referenced in separated pages. This may result in worse search performance and more chances of bucket overflows [59]. There are other problems with the clipping technique [36, 35]. The reader is referred to the above-mentioned papers for details. Examples of spatial data access methods using clipping technique include the R$^+$-tree [75], the Cell-tree [34] etc.

The multi-layer technique is similar to the overlapping-regions technique. It differs from the overlapping-regions technique [27] in that: (1) the layers are in a hierarchy; (2) each layer has its own partition scheme of the whole universe; (3) data regions in the same layer are disjoint; (4) the data regions do not adapt to the spatial extensions of the corresponding data objects. The potential problems with the multi-layer technique include but are not limited to the following: (1) a

query may need to access all layers; (2) spatially close objects may be arranged in separated layers. Examples of multi-layer technique are the *multi-layer grid file* [82] and the *R-file* [43].

The R-tree

R-tree [37], as a typical spatial index structure, has been under research for around two decades. It allows a fast query for all objects overlapping a d-dimensional rectangular region. An R-tree is a B^{+}-tree like structure, which stores multidimensional rectangles as complete objects without clipping them or transforming them to higher dimensional points or splitting them into multi-layers.

In the R-tree structure, a non-leaf node contains entries of the form

$$entry = (cptr, MBR)$$

where *cptr* is the address of a child node in the R-tree and MBR is the minimal bounding rectangle of all rectangles, which are entries in that child node. A leaf node contains entries of the form

$$entry = (Oid, MBR)$$

where Oid refers to a record in the database, describing a spatial object and MBR is the enclosing rectangle of that spatial object. Leaf nodes containing entries of the form $(dataobject, MBR)$ are also possible. A more detailed description of the R-tree can be found in Chapter 4.

The R-tree is a dynamic structure, where operations such as insertions and deletions can be intermixed with queries and no periodic global reorganization is required.

Guttman's R-tree defined the following parameters: M is the maximum number of entries that will fit in one node; m is a parameter specifying the minimum number of entries in a node ($2 \leq m \leq M/2$). The R-tree satisfies the following properties:

- The root has at least two children, unless it is a leaf.
- Every non-leaf node has between m and M children, unless it is the root.
- Every leaf node contains between m and M entries, unless it is the root.
- All leaves appear on the same level.

The R*-tree

The R*-tree [5] is based on the R-tree while optimized in the following aspects:

- The split algorithm of the R*-tree. The node-splitting algorithm [37] takes area, margin, and overlap of the directory rectangles into consideration when performing split operation.
- A novel forced reinsertion scheme. If a new data rectangle is inserted, each first overflow treatment on each level will cause a reinsertion of p entries. This may cause a split in the node if all entries are reinserted in the same location. Otherwise splits may occur in one or more other nodes, but in many situations splits are completely avoided.

The average insertion cost of the R*-tree is lower than for the R-tree. Though the R*-tree outperforms its predecessor, the cost for the implementation of the R*-tree is only slightly higher than the R-tree.

The Linear Node Splitting Algorithm

In order to minimize the search time of R-tree, the overlapping areas of the bounding rectangles have to be minimized as much as possible. Ang et al. presented the linear node splitting algorithm [3] to undertake the optimization.

In order to minimize the overlap between two new nodes split from the original node, all bounding rectangles are "pushed" as far apart as possible towards the boundary of the MBR of the original node. In a d-dimensional space, $2d$ rectangle lists, each of which corresponds to a border of the d-dimensional MBR, are maintained to store the rectangles that are nearer to the border than to its opposite. The split direction is then determined as on the dimension whose two rectangle lists (1) have closest number of rectangles, or (2) have smallest overlaps, or (3) have smallest total coverage.

Experiments conducted by Ang et al. show that the R-tree created using the linear node splitting algorithm outperforms conventional R-tree in range queries.

LOD-R-tree

Level-of-Details (LOD) is an important graphics technique, which uses a series of multi-resolution representations of a complex model for rendering. Different representations contain different levels of geometric or image details describing

the model. In [47], Kofler et al. introduced an R-tree combined with Level-of-Details. At the root level, one large bounding rectangle contains the entire scene, several smaller rectangles contain parts of the scene, which are again subdivided into even smaller rectangles. The novel idea in the work is that each level of the R-tree is simply a level-of-details. Traditional R-trees only contain lists of sub-rectangles down to the second lowest level. In contrast, LOD-R-trees also include graphical information in all nodes. Thus, it requires more storage for each node. The information is sufficient to visualize the geometry of one resolution bounded by the rectangle. A traversal of deeper levels of the R-tree is only necessary for regions near the viewpoint.

Searching for objects in the virtual scene is equivalent to traversing the LOD-R-tree. LODs are taken into account by processing queries for different boxes in different ways. Queries for rectangles closer to the viewpoint need to traverse to deeper levels in the R-tree, while those for farther can terminate the traversal at high levels in the tree. One problem with this approach is that there are usually large overlaps between queries issued from the same view frustum. As a result, a lot of nodes have to be accessed for more than once. To prevent data from being reloaded from disk, the nodes have to be marked by a dirty-bit. Such queries also retrieve irrelevant data.

Hilbert Space-Filling Curve Method

Mapping multi-dimensional data to one dimension with simple one-dimensional access methods is regarded as a solution to indexing multi-dimensional data [24].

Space-filling curves, which pass through every point in a space, have been utilized to map the coordinates of the points in the space to the one-dimensional sequence numbers on the curves.

The Hilbert space-filling curve has been deployed to index multi-dimensional data. J. K. Lawder and P. King [53] developed a technique to utilize the Hilbert Curve to implement a fully funtioning data storage and retrieval application. In this thesis, we will not consider utilizing the space-filling curve.

2.2.2 Spatial Techniques in 3D Computer Graphics

As we mentioned, the 3D region that a user can see in the world space is defined by a view frustum, which consists of six clip planes. A basic optimization technique that can be deployed with the view frustum is to bound scene objects with bounding volumes, and test for overlap between the bounding volumes and the view frustum. If the bounding volume of an object does not intersect the view frustum, it is considered "out of the user's eyesight", therefore, the object does not need to be rendered. This kind of object culling based on the overlap test of the view frustum and the bounding volume is called *view frustum culling* (VFC). If every object in the scene, which contains n objects, is tested against the VFC, the computational complexity is $O(n)$.

Hierarchical view frustum culling (HVFC), dating back to 70's, is an extended view frustum culling technique that applies on levels of the bounding volumes in the scene hierarchy [14]. Each node in the hierarchy has a bounding volume (BV) that encloses a part of the scene. During each rendering operation, the

scene hierarchy is traversed from the root, and if a BV is found to be outside the frustum during the traversal, then the contents of that BV need not be processed further, and performance is improved. For a scene hierarchy such as an octree or R-tree, hierarchical view frustum culling requires much less time than a linear culling process ($O(\log n)$).

Reducing the time for each view frustum culling operation is the key to improving the overall performance of HVFC. There are a few view frustum culling algorithms in the literature [4, 9, 31, 32, 39]. Assarsson and Möller [4] classify them into two common approaches:

- Perspective Transformation

  To perspectively transform the bounding volume of a node to be tested and the view frustum, to the perspective coordinate system and perform the testing in the new space. The advantage is when the bounding volumes are axis aligned bounding boxes (AABB), this results in testing two AABBs against each other, which is very simple. The problem with this common approach is that all vertices have to be transformed to the perspective space, requiring many floating point operations.

- Plane Testing

  The other approach is to test the bounding volume against the six planes defining the view frustum. The advantage of this approach is early trivial rejection of BV if it is outside one of the six planes or inside all six planes. If these fast tests fail, more expensive intersection tests need to be made. In [4], instead of doing so, the algorithm recursively continues testing the planes of

the sub-boxes in order to get higher performance.

One issue worth mentioning is that the bounding volume in VFC can also be a sphere, in which case the overlapping test could be faster as compared to bounding rectangles. This method has been applied in some computer games [87].

Nowadays, almost all high-level graphics APIs, such as OpenGL Optimizer, Java 3D, deploy some kind of view frustum culling to improve rendering performance.

2.3 Computing Visibility of 3D Objects

Some people categorize the view frustum culling (VFC) algorithms into the category of visibility algorithms, as objects behind the far-clip plane of the view frustum are regarded as "invisible". Existing visibility techniques include *back-face culling*, *view frustum culling* and *occlusion culling*. Since view frustum culling is basically a spatial technique, we shall focus on the other two techniques in this section. We also observe that previous work in visibility techniques seldom addressed the problem of visibility data storage and retrieval, and it did not consider the run-time performance when disk I/Os are accounted for.

2.3.1 Back-face Culling

Back-face culling algorithm avoids rendering geometry that faces away from the viewer. This culling operation can be left to the graphics hardware with no impact on the final graphics output. As figure 2.1 shows, back-facing polygons can be

identified by a simple dot product between the normal vector of the polygon and the vector of the viewing direction.

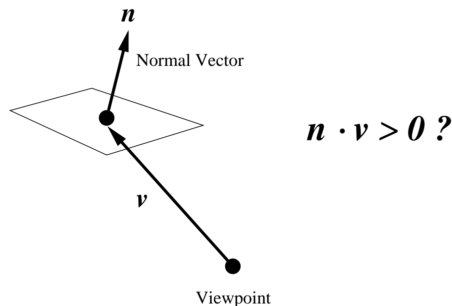


Figure 2.1: Back-face culling by testing dot product

A naïve back-face culling method checks every polygon one by one, which has O(n) computational complexity. In [51], a hierarchy of clusters of polygons, which is based on the normals and positions, is proposed. The algorithm partitions the space into regions with respect to each cluster. During run-time, the algorithm can reject entire clusters based on hierarchical comparison. Frame-to-frame coherence is also exploited to track the viewpoint. The algorithm is able to achieve sub-linear speed-up. However, a problem with this approach is that polygons must be reorganized into groups according to their normals. This makes the method unpractical with the existence of other criteria, for example, when polygons are required to be organized according to spatial location. Zhang et al. [93] propose a method to reduce the back-face test to one logical operation per polygon. The *normal mask*, where each bit is associated with a cluster of normals in a normal-space partitioning, is introduced. A polygon's normal is approximated by the cluster of normals in which it falls, while the cluster's normal mask is stored with the polygon in a preprocessing step. The backface test finally reduces to a single

logical AND operation between the polygon's normal mask and the backface mask.

The reader is referred to these papers for more details. Detailed discussion of back-face culling is beyond the scope of this thesis.

2.3.2 Occlusion Culling

This section provides a brief overview in occlusion culling techniques. For a more detailed description of various algorithms, the reader is referred to [19] or [16].

Occlusion culling has proven to be an important technique to reduce the number of polygons to be rendered and, therefore, to improve the performance of visualization. The visibility computation attempts to find the visible set, or the subset of polygons in the scene that are visible from a given viewpoint or a viewing region. Figure 2.2 shows the 2D case of a very simple scene with two occluders and a few occludees. Objects that are partially hidden and visible are usually considered as visible naturally. Suppose the scene, $\mathcal{S}$, is composed of n triangles $\mathcal{S} = \{P_1, P_2, \ldots, P_n\}$. Given a view frustum with viewpoint at p, a triangle P_i is said to be *visible* if there exists a line segment between p and a point $x \in P_i$, denoted as $\overline{px}$, which intersects no other triangles in $\mathcal{S}$.

For large scenes, the number of visible polygons is usually much smaller than the total size of the input [16]. For example, in a walkthrough of an urban area or forest, at almost any position the user can usually see only a few buildings or trees, which consists of a small portion of the whole scene model. Anther example is the walkthrough of a building with walls and doors separating rooms and blocks. At any moment, the user's viewpoint can always be in a single room or out of

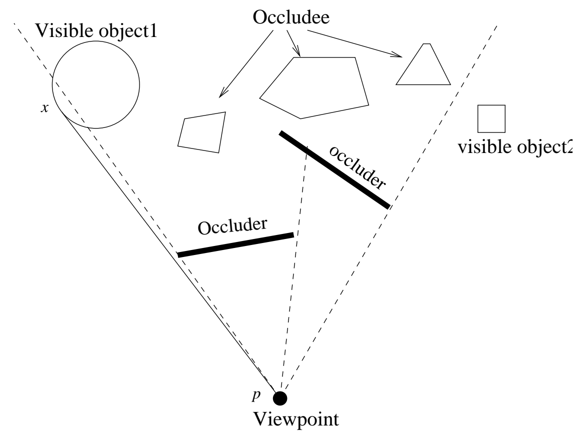


Figure 2.2: A simple example of occlusions at a given viewpoint

the whole building. When it is inside a room, the things that the user can see are those inside the room and those visible through doors and windows, which are often referred to as *portals* [26]. On the other hand, if the viewpoint is out of the building, there is no care about how the internal parts are structured.

Visibility is a hard problem since a small change in the viewpoint might cause large changes in the visibility [16]. Therefore, unlike spatial methods, there is no guaranteed spatial locality to be exploited in visibility computation.

The problem of occlusion culling is to avoid rendering polygons that are occluded by others in the scene. Consider a large scene containing millions of polygons, but only a small portion of them is visible from a single viewpoint. For such scenes, we would prefer the occlusion culling algorithms to run in time which is proportional to the number of the visible polygons, instead of the total number of polygons in the scene. This feature of the algorithm is called *output sensitive.*

Many visibility algorithms compute *conservative visibility* instead of exact visibility, as computing the latter may be too expensive. Conservative visibility set is the object set that includes all visible objects and probably some invisible ones.

A conservative visibility algorithm may mistakenly classify some invisible objects as visible, while it will never miss a really visible object. The invisible part of the output of a conservative visibility algorithm can be removed by a hidden-surface-removal algorithm in the subsequent graphics processing. The output from a conservative algorithm is also known as a *potential visible set* (PVS).

In contrast to conservative set, approximate visibility set is a set that includes most of the visible objects and probably some invisible ones. An approximate visibility algorithm cannot guarantee that all visible objects are included in the result set. Approximate algorithms are less common than conservative ones. In [46], a prioritized-layered projection (PLP) is introduced for optimizing the rendering of high depth-complexity scenes. By rendering an estimation of the visible set for each frame, the system computes on demand a priority order for the polygons that maximizes the likelihood of rendering visible polygons before occluded ones for any given scene.

Visibility algorithms can be categorized into *point based* methods and *from-region* ones. Point based algorithms solve the visibility on a given viewpoint, while the from-region algorithms compute visible part of the scene when viewed from a region. Many point based visibility methods can treat occluder fusion. However, point based methods are usually deployed in run-time rendering, therefore incurring a lot of computational overhead [20]. In recent years, a lot of *from-region* visibility algorithms have been developed. These algorithms compute visibility on regions, where the user's viewpoint can move around. When the viewpoint is inside a region, the same PVS will be used. The from-region algorithms capture some temporal

coherence of user's motion, therefore, the cost spent on each rendering frame is expected to be small. We introduce some well-known techniques applied in various point based and from-region visibility algorithms.

2.3.3 Point Based Visibility

Coorg et al's Algorithm

Coorg et al. [18] proposed a technique to compute object space visibility. This algorithm is based on several ideas. First, the algorithm requires a simple (and fast) visibility test that identifies whether some region of the model is completely or partially occluded by a set of occluders. Second, a cheap preprocessing step that identifies nearby large occluders for all viewpoints is described. Finally, a hierarchical visibility algorithm repeatedly applies the visibility test to determine the status of tree nodes in a spatial hierarchy.

As figure 2.3 shows [18], the supporting and separating planes of A and T can be used to detect which of these cases holds. First, the planes are oriented toward the occluder to form half-spaces. We say that a viewpoint satisfies a plane if (and only if) it is inside the plane's positive half-space; this relation can be checked by performing an inner product of the viewpoint with the plane equation. Full occlusion occurs when all of the supporting planes are satisfied; that is, when the viewpoint is in the intersection of the supporting half-spaces (region 3). Partial occlusion occurs when all the oriented separating planes are satisfied, but some supporting plane is not (region 2). Otherwise, there is no occlusion (region 1). The algorithm uses such simple and fast tests to track visibility events among objects.

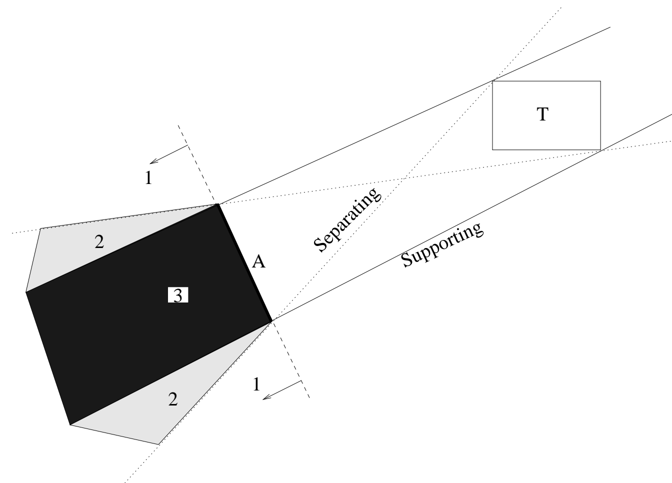


Figure 2.3: Occlusion in 2D [18]. Separating and supporting planes of an occluder A and an occludee T.

The algorithm dynamically selects a set of occluders, and uses them to determine which portions of the rest of scene cannot be seen in a hierarchy of objects. Occluders are selected based on the so-called area-angle, which is similar to the solid angle.

The algorithm works well with scenes containing large occluders, such as urban and architectural walkthrough. But the requirement for large occluders limits the genericity of the algorithm.

Shadow Volume BSP tree

Chin and Feiner introduced the Shadow Volume BSP tree algorithm for point light sources [12]. It can be used to compute shadows for polyhedral scenes. The algorithm uses a BSP tree to order the input polygons in increasing order of depth from the light source (viewpoint in the visibility case). In the process of building the BSP tree, the scene polygons are split and marked as lit or shadowed. This algorithm is suitable for visualization of static scenes.

Chrysanthou et al. [13] presented an algorithm using SVBSP tree for shadow calculation in dynamic polyhedral scenes. The tree is built incrementally by inserting the light-facing polygons into an initially empty tree. Subsequent polygons are filtered into the tree by comparing them at each level against the plane at the root of the tree and recursively inserting them into the appropriate subtree. During the building of the tree, each inserted polygon constructs a list of pointers to the locations it occupies on the tree. When a dynamic object moves, its polygons and their shadow planes in the tree are found using the location lists and are marked. After all relevant polygons have been marked, the SVBSP tree is traversed and all marked nodes are removed. The dynamic objects can then be reinserted into the SVBSP tree. The run-time execution of the algorithm is in near real-time.

The Shadow Volume BSP tree describes the scene elegantly with each node representing a plane in the scene. The short-coming of using SVBSP trees is the very expensive floating point computation.

Hierarchical Z-buffer

The Hierarchical Z-buffer (HZB) method [32] is an extension to the *de facto* standard, the Z-buffer algorithm [89]. This algorithm works on two hierarchical data structures, an object-space octree and an image-space *Z-pyramid*.

The polygons in the scene are organized in the object-space octree. A node in the octree is said to be hidden if its associated cube is hidden with respect to a Z-buffer. With this definition, it is possible to combine the Z-buffer with an octree spatial subdivision. If a cube is hidden with respect to a Z-buffer, then all

polygons fully contained in the cube are also hidden. Therefore, if the faces of an octree cube is scan-converted and found that each pixel of the cube is behind the current Z-value in the Z-buffer, all the geometry contained in that node is hidden.

However, scan-converting a large octree cube which may occupy large number of pixels is expensive. To reduce the cost of scanning large numbers of pixels, the Z-pyramid can be used. The idea of the Z-pyramid is to use the original Z-buffer as the finest level (bottom level) in the pyramid and then combine four Z-values at each level into one at the next coarser level by choosing the farthest Z-value from the observer. Such choice of Z-value on the next coarser level is conservative as any cube which is determined to be hidden on a coarser level will always be hidden if tested on the next finer level.

Each time the Z-buffer is modified, the new Z-values are propagated through to coarser levels (upper levels), until the new Z-value is no farther than the old value stored in the pixel.

An additional temporal coherence list is also used to exploit the temporal coherence in interactive visualization. This list stores the visible cubes from the previous frame. By rendering the geometry on the list, and using the resulting Z-buffer to form the initial Z-pyramid, the subsequent Z-pyramid test will be much more effective than when started from scratch.

The Hierarchical Z-buffer computes the visible polygons in the two hierarchical data structures elegantly. However, reading the Z-buffer and the bottom-up update of the Z-pyramid could be expensive without the support of a specific hardware.

Hierarchical Occlusion Map (HOM)

Zhang et al. [94] presented the hierarchical occlusion map technique for visibility culling of scenes with high depth complexity. The idea of the work is similar to the Hierarchical Z-buffer. The algorithm uses an object space bounding volume hierarchy and a hierarchy of image space occlusion maps. Occlusion maps represent the aggregate of projections of the occluders onto the image plane. For each frame, the algorithm selects a small set of objects from the model as occluders and renders them to form an initial occlusion map, from which a hierarchy of occlusion maps is built.

The algorithm first renders the occluders into an image, which forms the lowest-level and highest resolution occlusion map. This image represents an image-space fusion of all occluders in the object space. The occlusion map hierarchy is built by recursively filtering from the highest-resolution map down to some minimal resolution. The highest resolution need not match that of the image of the model database. Hierarchy construction of the HOM can be accelerated by graphics hardware that supports bilinear interpolation of texture maps or the CPU, whichever is faster.

Occlusion maps represent the fusion of small and possibly disjoint occluders. The opacity values in a low resolution occlusion map give an estimate of the opacity values in higher-resolution maps. Occlusion maps are based on the average operator rather than the minimum or maximum operators. This property allows for a conservative early termination of the overlap test.

Visibility is tested by performing overlap checking first. If the screen projected

bounding box of an object overlaps pixels of the HOM which are not opaque, the node cannot be culled. If the pixels are opaque, then the object is projected on a region of the image that is covered, and depth comparison with a single Z-plane or a software *depth estimation buffer* can be performed.

As compared to Hierarchical Z-buffer, the HOM method does not require a Z-buffer. This makes it useful for low-end systems. The construction of HOM has wide support from graphics hardware (texture mapping with bilinear interpolation). HOM supports conservative early termination in the hierarchical test by using a transparency threshold and approximate occlusion culling by using an opacity threshold.

2.3.4 From-region Visibility

Virtual Occluders

Virtual occluders were introduced by Koltun et al. [48]. A virtual occluder is a simple convex object that is guaranteed to be fully occluded from any viewpoints in the viewing cell. It works as an effective occluder which enables aggregate occlusion of a large area. Virtual occluders enable effective region-to-region culling technique. In [48], an object space method is given to synthesize such virtual occluders by aggregating the visibility of a set of individual occluders.

First, a few objects are identified as seed objects. For each seed object, a cluster of spatially close objects is found, and the effective occlusion of these objects can be represented by a single virtual occluder. Initially, each cluster only contains the seed object. More objects that satisfy a criterion are added to the clusters

and the effective umbra of the clusters is augmented iteratively. In Koltun's work, the virtual occluder is placed just behind the furthest object in the cluster, and is completely contained in the aggregate umbra of the cluster. In the real-time rendering, the PVS of the viewing cell is computed just before the walkthrough enters the viewing cell. It is done by hierarchically testing the scene-graph nodes against the virtual occluders. As the number of virtual occluders is very small, the test is very fast.

The occlusion generated by virtual occluders can be very effective treating 2.5D or 2D scenes such as buildings and urban areas. A 3D scene can be divided into several 2.5D slices along the height dimension, and 2D occluders can be constructed in each slice. These 2D virtual occluders are then extended back to 3D. However, this restricts the algorithm to be used for generic scenes and occluders.

Volumetric Visibility

By using a hierarchy of voxel structure, Schaufler et al. [72] developed a conservative from-region algorithm. This algorithm treats the space as a discrete voxel structure and uses the opaque interior of objects as occluders. Such choice of occluders facilitates the extension into adjacent opaque regions of space.

Given a viewing cell, Schaufler et al.'s algorithm takes the following steps: (1) Rasterize the boundary of scene objects into the discretization of space and determine which voxels of space are inside an object and therefore opaque. (2) Traverse the discretization of space and find an opaque voxel that is not already hidden. Group this blocker with neighboring opaque voxels to get an effective

blocker. (3) Construct a shaft that encompasses the region of space hidden by this blocker as seen from the viewcell. (4) Use the shaft to classify the voxels into partially or completely outside the shaft and fully occluded. Take note of occluded voxels. Blocker extension is performed into hidden space, regardless of whether this space is empty or opaque. Opaque blockers are used in the order from large to small and from front to back.

A shaft is constructed around the occluded region from the supporting planes between the viewing cell and the blocker. Once it has been constructed, a recursive traversal of the spatial data structure flags hidden voxels as occluded in the highest tree node possible.

To determine the visibility status of objects in the original scene description, the bounding boxes they occupy are inserted into the tree and checked if all the voxels they overlap are hidden.

The algorithm has been tested over visibility preprocessing for real-time walk-through and reduction in the number of shadow rays in a ray-tracer [72].

The main contribution of this paper, as the authors claimed, is a conservative scene discretization that decouples the effectiveness of visibility calculations from how the scene was modeled (or, in other words, with no assumptions of large or convex occluders), and the introduction of blocker extension as a means of both finding efficient blockers and performing effective occluder fusion.

Extended Projection

Based on point-based visibility algorithms such as the Hierarchical Z-buffer (HZB) [33] or the Hierarchical Occlusion Maps (HOM) [94], Durand et al. presented a technique called extended projection to extend point-based methods to from-region visibility [20]. Occluders and occludees are projected onto a plane, and an occludee is marked as hidden if its projection is completely covered by the cumulative projection of the occluders. The notion of *Extended Projection* is defined separately for occluders and occludees. The extended projection of an occluder onto a plane, with respect to a cell, is the intersection of the views from any point within the cell; while the extended projection of an occludee is defined as the union of all views from any points of the viewing cell. An improved extended projection for occludees for specific, but not uncommon configurations, is also presented.

A hierarchical pixel-map representation of extended projections, called extended depth map, is proposed in [20]. Extended projections of multiple occluders can be aggregated. Concave occluders that intersect the projection plane are sliced. Occluders can be re-projected onto new planes. The position of the projection plane is crucial for the effectiveness of extended projections. The re-projection operator is used to define the occlusion-sweep, where parallel planes can be swept through the scene leaving the cell. By performing occlusion sweeping, occlusions generated by multiple concave or small occluders can be aggregated to generate large occlusions.

Occluders are selected using a solid angle heuristic, similar to that presented in [94]. To improve the efficiency of occlusion tests, an adaptive scheme, which

selects more occluders in the direction where many occludees are still identified are visible, is implemented.

Run-time rendering is performed on a SGI Performer scene graph. Visibility data are encoded into delta-PVS, where only the difference between two adjacent cells is stored. Nodes in the scene graph are tagged as visible or inactive when the scene graph is updated each time the observer enters a new cell.

The merit of this algorithm is that it performs occlusion aggregation and occlusion sweep, therefore, small occlusion generated by small occluders or convex occluders can be aggregated to form large occlusion.

Occluder Shrinking

In [92], Wonka et al. presented a technique that is based on occluder fusion. The basic idea of this technique is as follows: Shrinking an occluder with a small radius ϵ produces a smaller umbra. If an object is occluded by the shrunk occluder, it will still be occluded by the original occluder from a very smaller region centered at the viewpoint. As a result of this observation, the visibility from a very small region can be converted to a problem of point visibility at a viewpoint, and the visibility from a large region can be solved by picking sample points on the region.

For each sample point, the algorithm rasterizes occluder shadows into an image plane called *cull map* using hardware Z-buffer. Large number of occluders can be quickly filtered through a hierarchical approach so that large portions of the occluded occluders can be quickly rejected. If an object (or a potential occluder) is inside the cull map, it is tested against the Z-value of the current cull map and

is rejected if it is invisible. If an object is outside the cull map, a *view channel* between the viewing cell and the bounding box of the object is constructed and tested against the edges of the cull map. The object is occluded if the computed view channel is completely contained in the horizon.

This algorithm was proved to be very effective when processing outdoor 2.5D building walkthrough. For the dataset used by the authors in the paper, over 99% percent of the scene was pruned from about 200 visible occluders. However, the cull map technique can only be used on 2.5D scenes. This restriction makes the technique limited to urban scenes, and may not be suitable for other generic scenes.

Another work applying occluder shrinking was done by Wang et al. [88]. In this work, the occluder shrinking is applied in image space to exploit temporal coherence for on-line visibility computing. In this case, occlusion computing is performed only if needed. Objects in the scene are associated with a *time stamp*. Visibility computation is needed when: (1) the real-time reaches the value of the time stamp; (2) a dramatic change happens in the viewing direction; (3) an occluder is deleted from the scene. The occluder shrinking is performed on a hierarchy of occlusion masks. The scene tree is traversed to mark the occluded nodes with the computed time stamp. As the occlusion culling is performed online, the system is able to deal with some dynamic objects.

Compared to [92], this work is less restrictive on the selection of occluders and scene structure, as it calculates visibility online. However, this could also affect run-time system performance.

2.3.5 Computing Visibility Using Graphics Hardware

Most modern graphics cards can accelerate rendering by using some sorts of hardware acceleration in various stages of the rendering pipeline. For example, back-face culling and view frustum culling can all be left to the graphics hardware. However, doing so would inevitably increase the amount of data sent to the graphics card, which is usually very expensive. A good visualization system usually culls geometries in the early stages of the rendering, before sending the rest to the hardware. Some hardware has provided occlusion culling acceleration. A well-known hardware feature is available on some HP machines. The `GL_OCCLUSION_TEST_HP` mode allows an application to quickly determine the visibility of some set of geometry based on the visibility of the bounding box of the geometry [41]. This token enables HP's occlusion-testing extension. If any geometry is rendered while occlusion culling is enabled, and if that geometry would be visible (i.e., rendering it would affect the Z-buffer), the occlusion test state bit is set, indicating that the rendered object is visible. This is typically done to increase performance: if every pixel of a bounding box is "behind" the current Z-buffer values for those pixels (i.e., the bounding box is entirely occluded), anything drawn within that bounding box would also be behind the current Z values, and therefore one can cull it (i.e., avoid processing that geometry through the pipeline). Note that this enable has no effect on the current render mode, or any other OpenGL state.

The following are the steps to use this hardware-accelerated OpenGL feature.

- disable updates to color and depth buffer (optional)

```
glDepthMask(GL_FALSE);

glColorMask(GL_FALSE,GL_FALSE,GL_FALSE,GL_FALSE);
```

- enable occlusion test

```
glEnable(GL_OCCLUSION_TEST_HP);
```

- render bounding geometry

```
gl rendering calls ;
```

- disable occlusion test

```
glDisable(GL_OCCLUSION_TEST_HP);
```

- enable updates to color and depth buffer

```
glDepthMask(GL_TRUE);

glColorMask(GL_TRUE,GL_TRUE,GL_TRUE,GL_TRUE);
```

- read occlusion test result

```
glGetBooleanv(GL_OCCLUSION_TEST_RESULT_HP, &result);
```

- if (`result`) render internal geometry

  else do not render

2.4 Multi-resolution Representations

Many computer applications in computer graphics and other fields implement interactive visualization for complex geometric models. Usually, these original models such as the ones used in virtual reality applications require computation and storage far exceeding the capacity of modern graphics hardware. Using multiple representations that have different resolutions for the same model in different contexts is very effective in such cases. There are many existing multi-resolution techniques designed for interactive visualization of complex polygonal or image-based models.

Early in 1976, Clark [14] presented a technique to use simpler versions of the geometry for objects that have less visual importance. These simpler versions of an object are Levels of Details (LOD). There are various techniques to generate LODs, such as *polygon simplification*, *texture-based simplification* etc [10]. We will only look at the polygon simplification algorithms here. More detailed review about LODs and polygon simplification algorithms can be found in [50, 29]. We will not discuss texture-based simplification in this thesis.

Polygon simplification algorithms are usually used to generate levels of details for a complex object. According to Erikson et al. [22], polygon simplification algorithms can be typically categorized into these classes: vertex removal, edge collapse, face collapse, vertex clustering and vertex merging. We only consider some of these in the following.

Schroeder et al. developed a *vertex removal* algorithm which iteratively picks a vertex to remove, as well as all adjacent faces [73]. This algorithm maintains the topology of the model being simplified. But it is restricted to manifold surfaces.

Vertex clustering algorithms are usually faster. However, as an inherited result of vertex clustering, the number of faces in the result is difficult to control, and the quality of the output is often not high [64, 45]. There are a few *edge collapse* algorithms in the literature [40, 63]. Hoppe's Progressive Meshes algorithm [40] can generate good results, but the algorithm takes long time to execute. A compromise between fast methods with low-quality results and slow ones with high-quality output is the iterative *vertex merging* method [28, 21]. The method in [28] merges two vertices to simplify the model based on quadratic error metrics. The algorithm was later extended into n-space to support color-preserving and texture-preserving simplification [30].

2.5 Summary

In this chapter, we have gone through the backgrounds and previous work on virtual walkthrough, as well as some related topics. Various techniques aiming at improving the performance of virtual walkthrough have been proposed in the literature. Basically, the previous work can be categorized as spatial techniques, visibility techniques, multi-resolution representations etc.

Existing spatial access methods (SAM) include transformation technique [55], overlapping-regions technique [37, 5], clipping [75, 34], and multi-layers technique [82, 43]. However, most of these approaches are not optimized for virtual walkthrough applications. Due to this problem, most of the spatial access methods are not directly usable for manipulating the data in a VE database.

Many visibility algorithms exploit hierarchical spatial data structures, hardware

acceleration, and spatio-temporal coherence to speed up visibility computation. Some of them compute visible set in run-time while others perform precomputation. In our case, since the database is in secondary storage, dynamically traversing the database and computing the visibility is prohibitive. So we essentially adopt precomputed visibility techniques.

Apart from the above two techniques, level-of-details has been proven to be very effective in dealing with large polygon sets. It is widely accepted as a standard approach to attain faster rendering speed without compromising much visual quality. The LODs can be generated in *run-time* or *precomputation*. We shall consider static LODs generated in precomputation only, as the latter is less demanding on CPU.

Chapter 3

Walkthrough System Architecture For Large VE

As an application stemming from techniques in the graphics areas, virtual walk-through applications mostly assume that the VE can fit into the main memory. However, this assumption is no longer reasonable, given the requirements of a real application. First, a realistic VE typically consists of thousands of virtual objects, each of which is represented by hundreds of polygons, and may take up gigabytes of storage space. Second, as users' expectation increases, we can expect more complicated models that capture fine details of the actual environment as closely as possible to be designed. This will lead to an explosion in the size of the model, even if it is a simple environment. Clearly, when the VE is too large to fit into the main memory, it becomes crucial to manage the main memory space effectively, otherwise, the frame rates can become unacceptable. Similarly, for the memory-resident objects, we need to restrict the amount of data fed into the graphics engine

to guarantee good performance.

A natural extension to memory-resident walkthrough system for very large scenes is to store the scene representation on hard disk, and access the scene data in the run-time. Due to the long access-time of disks, however, it could be extremely expensive to retrieve scene data from hard disk. In this chapter, we propose a generic system architecture to enable real-time walkthrough of a very large scene which is stored on hard disk. The techniques in subsequent chapters are all based on different incarnations of this generic system framework.

In Section 3.1, we discuss the scene graph structure; in Section 3.2, we propose the generic system architecture for disk-resident virtual walkthrough; and in Section 3.3 we also look at the data sets to be used in the implementations and experiments.

3.1 The Scene Graph Structure

A virtual world in memory is usually organized in a hierarchical scene graph structure [62], which has levels of nodes representing geometries, materials, transformations and other properties. Rendering of the scene is equivalent to the traversal of the scene graph. Each geometry node of the scene graph is associated with a bounding volume, which can be used to test if it overlaps the view frustum. If it does not, the traversal will not render the node, and will proceed to other nodes. This process, often referred to as *View Frustum Culling*, is a standard culling process and has been applied in most of the high-level graphics libraries, such as Java 3D, Iris Performer, OpenGL Optimizer etc. Figure 3.1 shows a simple example of the scene graph structure. Each node in the scene graph has a few pointers

pointing to child nodes (in ellipse) or attributes (in boxes). Note that nodes can share attributes or child nodes to save memory space and graphics processing.

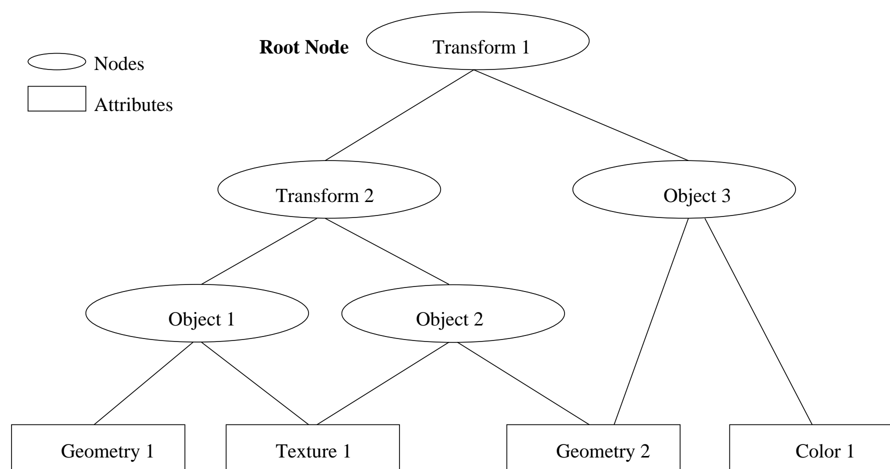


Figure 3.1: A Simple Example of Scene Graph

In some scene graphs, if no nodes share attributes and/or child nodes, the scene graph is also a *scene tree* structure, as each node except the root node has a unique parent node. In this thesis, we will consider the scene tree structure only.

Visibility culling is an important technique to prune hidden objects. In scenes like cities or forests, a large part of the scene is occluded by some large *occluders* with respect to the viewpoint. Such scenes are said to be *densely occluded*. We have already reviewed some visibility culling algorithms in chapter 2. A conventional method of performing visibility culling is to reserve a visibility bit for each object in the scene graph, and set it to "1" if it is visible or "0" otherwise. When the recursive traversal reaches the object, the visibility bit is tested to determine whether to render it or not.

Another typical technique to reduce the load on the GPU (Graphics Processing Unit) and I/O is to apply multi-resolution representations to geometrically com-

plex objects. Objects that are far away from the view point are represented with low resolutions, while those that are near are represented with high ones. Static LODs are often implemented in scene graphs by associating a number of geometry attributes to a node and switching the current LOD to an appropriate geometry.

3.2 Generic System Architecture for Disk-Based VE

In traditional database applications, data stored in secondary storage can be manipulated directly once they are loaded into memory. This is not the case for a walkthrough system for a large VE. Data in a walkthrough system for a very large VE have two different representations – one for external storage, and the other for internal (main memory) manipulation. This is necessary in order to optimize the performance. On one hand, virtual objects are organized in the secondary storage based on their spatial locality so that objects that are near to one another can be loaded into memory with minimal I/O cost. On the other hand, existing graphics engines are optimized to manipulate virtual objects in memory in certain format (e.g., scene graph in our case). The overhead incurred is the transformation between the two representations. Traditionally, spatial objects that overlap the view frustum can be retrieved from the disk and be sent to the graphics engine. It is, however, not efficient to retrieve all objects overlapping the view frustum from disk in every rendering frame. Therefore, there are three potential bottlenecks in the system:

1. I/O bottleneck: loading the data (index and virtual objects) into memory.
2. CPU bottleneck: transforming the data from disk-based format to in-memory format.
3. GPU (Graphics Processing Unit) bottleneck: loading the graphics pipeline with data to be rendered and viewed.

We will focus on the first and last problems. Our current solution to the second problem is straightforward: we only transform those disk-based data that are most likely to be accessed.

3.2.1 Components of Disk-Based Walkthrough System

A disk-based virtual walkthrough system has the ability to dynamically query the graphics database and adjust the scene graph. Figure 3.2 shows a generic system architecture for disk-based virtual walkthrough.

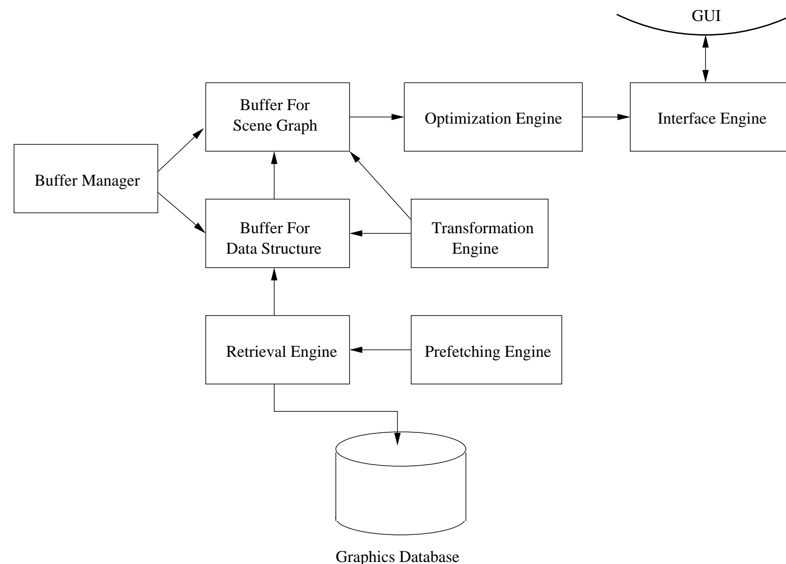


Figure 3.2: A Generic System Architecture For Disk-Resident Virtual Walkthrough

The framework consists of a few components as follows:

- *Retrieval Engine*

  The retrieval engine is in charge of loading appropriate models from the secondary storage. When designing a high-performance retrieval engine, a few issues need to be considered in order to select appropriate data to be loaded into memory in each query:

  1. The retrieval strategy must be designed so that the likelihood of duplicated retrieval of the same data in consecutive queries could be minimized. A good design can prevent or minimize the duplication among consecutive queries.

  2. Data retrieved in the current operation must have a great probability to be consumed in the current rendering frame or in the near future. Sometimes data being loaded are not immediately demanded by the rendering process but are very likely to be used shortly. Such loading process is known as *prefetching* of data. As the queries issued during a walkthrough session is often marked by a specific pattern, the prefetching process must be based on walkthrough semantics.

  Besides these issues, other problems such as data compression, parallel processing, etc. can also be considered. However, in our work we shall focus on the aforementioned two issues when designing the retrieval engine.

- *Prefetching Engine*

  The function of the prefetching engine is self-explanatory. It predicts the

future position and orientation of the user and activates the prefetching process in the *retrieval engine* to load in data that are likely to be used very soon. This component is not necessary for all the walkthrough systems. The existence of a prefetching engine may lead to higher performance, as more required data can be ready in memory after prefetches.

- *Buffer For Data Structure*

  The buffer for data structure is the memory structure reserved for the disk-resident data structures (index) used in the database queries. The retrieval engine accesses the buffer to obtain the file address of nodes in the data structure (index) or the disk-format graphical objects. Designing efficient data structures is one of the most important tasks in a disk-based virtual walkthrough system.

- *Buffer For Scene Graph*

  As we have mentioned, the *scene graph* is a standard data structure for interactive rendering. The objects loaded by the retrieval engine are kept in memory and organized in a scene graph structure, which is optimized for rendering process. Data in the scene graph may be added, deleted, or modified according to specific walkthrough semantics. Sometimes the *buffer for data structure* may refer to the scene graph in order to check the validity of a specific node in the scene graph. There are numerous versions of various scene graph libraries provided by software vendors or individuals. In chapter 4, we use the OpenGL Optimizer™ scene graph; while in chapter 5, we develop our own scene graph structure for better control over the data conversion in

the HDoV-tree.

- *Buffer Manager*

  The buffer manager manages the memory buffers for the data structure and the scene graph. Designing efficient memory caching (buffering) techniques for the data structures and the scene graph nodes is very important given the long access time of disk-resident data.

- *Transformation Engine*

  The transformation engine performs the two-way "translation" between the disk-formatted data and the objects that are directly usable by the other components. When a disk index item is loaded by the retrieval engine, the transformation engine converts it into a memory index item. Also when an object model is loaded, it converts the model buffer into a node of the scene graph. When new objects are inserted into the database, it is the transformation engine that converts the data into appropriate format and saves them on the secondary storage.

- *Optimization Engine*

  The optimization engine conducts a lot of processing with the scene graph structure. When the scene graph is fed into this engine, there are various operations carried out with it. These operations may include, but are not limited to:

  - View Frustum Culling
  - Occlusion Culling

  - LOD Switching

  These operations are crucial for the performance of the graphics sub-system. Since the optimization engine runs CPU-bound tasks, reducing the processing time is very important for the design of the optimization engine.

- *Interface Engine*

  The interface engine reads the data that are filtered by the optimization engine, and sends graphics commands in the form of "graphics primitives" to the graphics hardware. Meanwhile, the interface engine also accepts user commands from the graphical user interface (GUI), such as "move forward" or "turn around", and makes changes to the user parameters which control the database access and graphics rendering.

3.2.2 How Things Work

In the previous section, we proposed the generic structure of a disk-based virtual walkthrough system. In run-time, these components work together to achieve high-performance virtual walkthrough. When the user makes some virtual "movements" via the input device, the interface engine is informed of these events. It checks the current 3D position of the user's viewpoint and determines if a database query is needed. If so, the user parameters (such as the position in 3D space and the orientation) are computed and the retrieval engine is called to issue a query to the database. The prefetching engine makes prediction to the user parameters, and prefetches data from the secondary storage. The retrieval engine passes what it

has recently loaded from secondary storage to the transformation engine, which then performs the conversion from the disk-formatted data to scene graph nodes or index data. The output of the conversion are stored in the respective buffers (for index or for scene graph). In each rendering frame, the optimization engine reduces the number of nodes or polygons to be rendered by pruning branches of the scene graph that do not overlap the current viewing frustum, setting lower LODs for some visually unimportant models, or culling occluded models.

Theoretically, data retrieval, conversion and graphics rendering could be implemented in separate processes (or threads) to improve CPU utilization. However, as the rendering process tends to occupy almost 99 percent of CPU time, on a single CPU machine, multi-threading does not help to improve the system performance. In our system(s), the data retrieval and conversion may occur in between two consecutive rendering frames. This is a general setting which is valid for all the systems to be proposed in the following chapters.

3.3 Data Generating

In order to evaluate the system architectures and various techniques on the system platform, we need large and complex models which can be navigated around. We would like the data sets to meet the following requirements: (1) The data sets must be large enough so that they cannot be loaded into the main memory of a normal PC; (2) The minimum unit in the queries of the data sets is an "object", which is small enough so that there are usually thousands of "objects" in such data sets; (3) Each of the "objects", usually represented in polygonal surface models, should

Name	**Number of polygons**
Bunny	69451
House 1	1812
House 2	552
House 3	240
Tower 1	540
Tower 2	984
Tower 3	1702
Tower 4	1348
Tower 5	1048

Table 3.1: List of base models in the synthetic data sets

be complex enough. By complexity of objects, we refer to the number of polygons contained in each object. The number of polygons of an object is also proportional to the storage space that it occupies, and the processing time that it takes.

Unfortunately, there are few publicly available models that are free of charge. Those that are available on the Internet are always either too simple or highly dependent on commercial viewers or CAD tools. Therefore, we had to generate our own data sets by importing some complex geometric models and making duplications. The base models that we use to generate the synthetic data sets are many building models and the Stanford bunny model [1] (at *http://www-graphics.stanford.edu/data/ 3Dscanrep/*). Table 3.1 shows the number of polygons in each of the base models in the synthetic data sets.

Besides the above issues, we also require the data set to be suitable for the real-time walkthrough applications and visualization. We choose synthetic data of city scape to be our data sets, as city walkthrough is one of the most obvious applications of VE in real life. In synthetic city data sets, we can perform real

[1]We would like to thank the Stanford Computer Graphics Laboratory for making the model available online.

virtual-walkthrough, and can change the size of the data sets with relatively fewer efforts. Intuitively, spatial-based querying and visibility-based querying can both be handled effectively in a city model, as many buildings and objects are far away to each other, or occluded by each other in a city.

The systems that we build for large database of virtual walkthrough have a precomputational phase, which may take hours on high-end PCs. For instance, in chapter 5, we need to precompute the visibility, a task that is fairly time-consuming. Therefore, we generate just a number of datasets of different sizes to investigate the scalability of the proposed techniques.

3.4 Summary

In this chapter, we have presented a generic system architecture for a large VE. We have presented the various components and the respective functions of the architecture. We have also discussed how different parts of the system work together to perform real-time walkthrough. Finally, we have discussed datasets that we used in our systems, as well as the issues on data generating.

Chapter 4

Virtual Walkthrough Using Spatial Techniques

Users expect high and constant rendering frame rates when they interactively navigate in a Virtual Environment. However, as discussed in the previous chapter, when the VE is too large to fit into the main memory, the frame rates may become unacceptable. The slow-down in frame rates are mainly caused by the following factors: (1) Disk-resident data need to be swapped into memory on demand. The retrieval could be expensive in CPU time and I/O, and therefore cause a drop in frame rate. (2) Data loaded from the secondary storage are not directly usable by the graphics sub-system to be painted onto the screen. They need to be converted to renderer-friendly format, i.e. scene graph format. (3) A lot of data stored in main memory are not always relevant to the viewer.

There is a lot of prior work in the spatial database area. We explore the possibility of reusing existing spatial database techniques to address the problem of

interactive walkthrough of a large VE. We would also like to minimize the efforts to be made to implement such a walkthrough system based on existing data structures. In this chapter, we study the problem of supporting a Virtual Walkthrough system with a disk-resident database using spatial techniques. In this thesis, we use R-tree as the application platform. We observe that the conventional search algorithm of the R-tree is not optimized for the virtual walkthrough semantics, where a lot of overlaps exist between consecutive query regions. We also note that the query process can be expedited by buffering some of the tree nodes in main memory. For example, the root node of the R-tree can always be buffered to improve search performance as it is accessed in each query. Another issue to be studied is the query strategy in application-level — if we are going to issue a query, how large should the query box be? And where should the query region be? We also look at the technique to improve the rendering speed for data in memory.

The rest of this chapter is organized as follows: First, in section 4.1 we give the overview of our solutions to the problems (on page 50) in a disk-based walkthrough system. We propose a walkthrough system, namely the REVIEW (REal-time VIrtual Environment Walkthrough) system. We discuss various techniques which optimize system performance in section 4.2 and 4.3. In section 4.4, we study the performance results of REVIEW. Finally, we summarize this chapter in section 4.5.

4.1 Overview of Our Techniques

In our solution, there are two distinct data representations of the VE in the system. In the secondary storage, data are organized based on their spatial location in an R-tree [37] index. In an R-tree index, the leaf nodes contain entries of the form (MBR, ptr) where MBR is the minimum bounding box of the virtual object being indexed, and *ptr* is a pointer to the object being indexed. Note that *ptr* is an address when the node and objects are in memory, or a file name when the node and objects are stored in a file or on disk. The non-leaf nodes contain entries of the form (MBR, ptr) where MBR is the bounding box of all the bounding boxes of the entries of the lower level nodes and *ptr* is the pointer to the lower level node in the R-tree. The rationale to apply R-tree index to the scene data will be described in section 4.2.3. In our implementation, we have also optimized the R-tree using the linear node splitting algorithm proposed in [3].

In the main memory, loaded data are transformed (using the *Transformation Engine*) into a *scene graph*. The scene graph, as discussed in chapter 3, is a hierarchical structure that captures the virtual objects and their features such as locality, colors, textures and lightings. To better manage the main memory, it is also organized into two distinct pools – one for manipulating the data loaded from external storage, and the other for the scene graph. The scene graph is then fed into the graphics engine for display.

To minimize I/O cost, we employ three optimization strategies. First, we propose a complement search algorithm that retrieves only the non-overlapped regions. Secondly, we have designed a Buffer Manager that manages the main memory allo-

cated to the R-tree index. We exploit the access patterns of a walkthrough to design a novel cache replacement policy for the R-tree nodes, namely *distance-priority-LRU* policy. Essentially, the policy keeps those nodes that are close to the current viewpoint in memory, while replacing those nodes whose bounding boxes are distant from the current viewpoint. Finally, we also deployed a prefetching technique to predict the position of the view cell that the user will be in. A Prefetching Engine is designed to handle this task.

For the memory-resident object data, we have also designed a view frustum search algorithm. The algorithm prunes objects that do not overlap the current view frustum before the rendering phase. The scheme also guarantees that only potentially visible objects are sent to the graphics engine.

With regard to the problems discussed in section 3.2 (on page 50), our basic strategy to the first and last problems is to associate the user's viewpoint with two different convex cells:

- The first cell, called *frustum cell*, is a sufficiently small one that contains the view frustum. It serves as a search region to determine the in-memory objects that should be sent to the graphics engine. In this way, irrelevant data in the memory can be pruned and only the potentially visible data are passed to the graphics engine. This is realized by a View Frustum Search Engine, a component of the *Optimization Engine*.
- The second cell, called *disk cell*, is used to determine the objects that should be loaded into memory. For simplicity, both cell types are axis-aligned boxes. The disk cell contains the frustum cell and is larger than it. Whenever the

user moves such that its frustum cell falls out of the corresponding disk cell, objects belonging to a new disk cell will have to be fetched. Clearly, two consecutive disk cells often have significant overlaps.

This approach has two main advantages. First, in an interactive walkthrough process, query frustums in consecutive rendering frames usually have significant overlaps, as the user's viewpoint moves smoothly. Motions of the user are usually combinations of translations and rotations. By using a cell whose size is larger than the view frustum, we can store the previous results in memory and retrieve data merely for non-overlapped areas in the next rendering frames. Second, if the frustums of the next frames are totally bounded in the original box, there is no need to access data from secondary storage. This can also lead to higher frame rate. The disk cell is used by the *Retrieval Engine* when data from secondary storage are accessed.

We have implemented a prototype walkthrough system called REVIEW (REal-time VIrtual Environment Walkthrough) [78]. We have also evaluated its performance on a 1 GB synthetic data-set generated to simulate a large cityscape.

Designing effective and efficient methods for the various components is our main research focus. We shall look at the novel algorithms that we have proposed for the various components in the system.

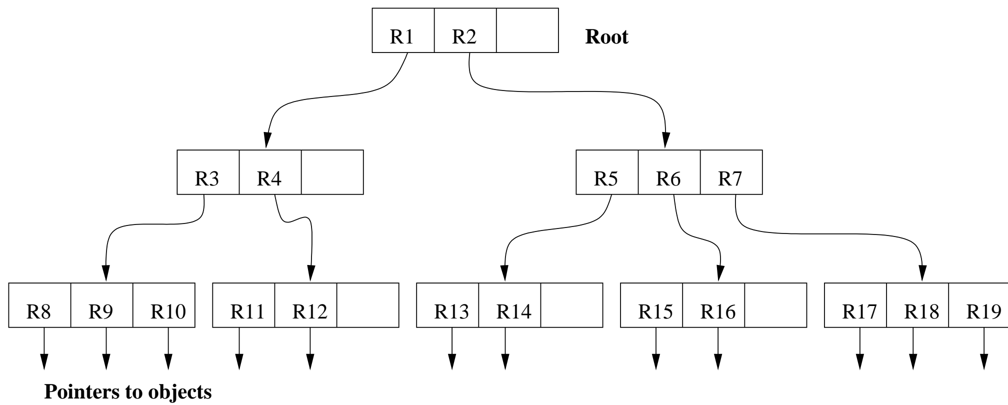
4.2 Optimizing I/O Performance

In this section, we shall examine several techniques that we have deployed to overcome the I/O bottleneck, namely an efficient search algorithm, an effective buffer replacement policy and an intelligent prefetching scheme.

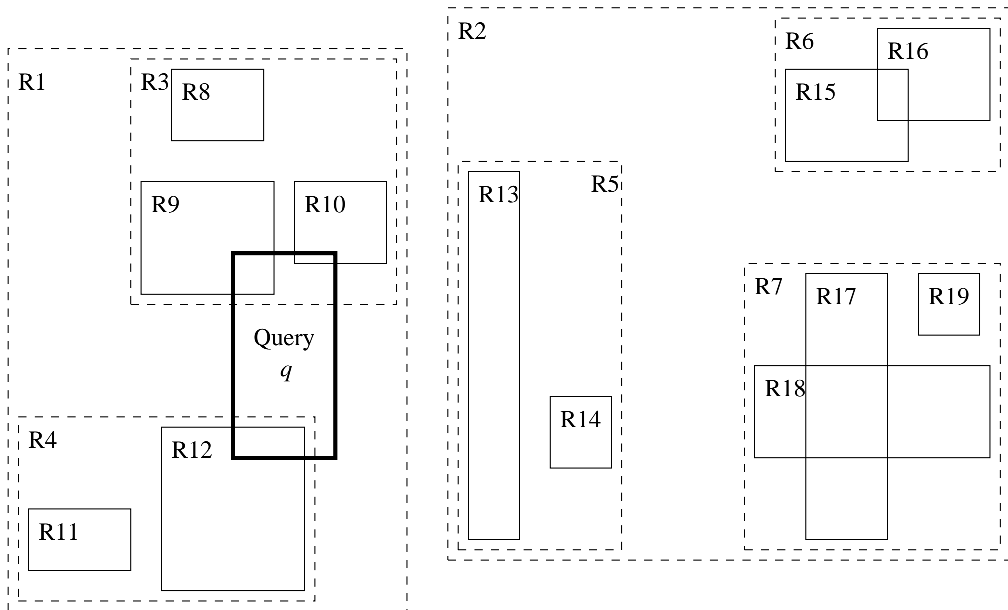
4.2.1 Spatial Index of The Data Set

Choosing an appropriate spatial indexing structure for the disk-resident scene data is crucial for the run-time performance of the walkthrough. K-D tree, octree, BSP tree, R-tree, etc. are well-known data structures for organizing 3-dimensional data. However, not all of these are suitable for a disk-resident data set. A very important feature of a disk index structure is *whether it is height-balanced*. A balanced tree is one where no leaf is much farther away from the root than any other leaves. Therefore, a balanced tree achieves a more balanced performance when searching for leaf nodes. Although other balanced spatial structures can be used, an R-tree, as a generalized B^+-tree, also optimizes costly disk accesses that are of concern when dealing with large data sets. As disk I/O is one of the bottlenecks in our system, we choose to employ an R-tree index for the disk-resident data set, so as to improve the I/O performance.

By definition [37], R-tree is a height-balanced tree similar to B^+-tree with index records in its leaf nodes containing pointers to data objects. Operations of R-tree include rectangular search algorithm, insertion algorithm and deletion algorithm. As figure 4.1 shows, (a) is a very simple example of an R-tree, (b) shows the 2-dimension layout of the bounding boxes of the nodes in this R-tree. Suppose



(a) A simple R-tree



(b) 2D layout of R-tree nodes

Figure 4.1: An example of R-tree

that each of the leaf level bounding boxes bounds a virtual object in the data set, then box R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, and R19 bound a total of 12 objects. Given a query region q as shown in figure 4.1(b), the search algorithm firstly checks the rectangular overlap at the root level. As R1 intersects q, the algorithm therefore proceeds to the children of R1 for further

overlap checking. Meanwhile, since R2 does not intersect q, indicating that no object in the sub-tree of R2 spatially overlaps q, so the whole sub-tree under entry R2 can be pruned. When coming to R3 and R4, the search algorithm finds that both rectangles intersect q, so both sub-trees need to be searched. At the leaf level, the result set R9, R10, R12 can be found, and the corresponding three objects can be retrieved. The objects bounded by R8 and R11 are not retrieved because R8 and R11 do not intersect the query box.

4.2.2 Complement Search Algorithm of R-tree

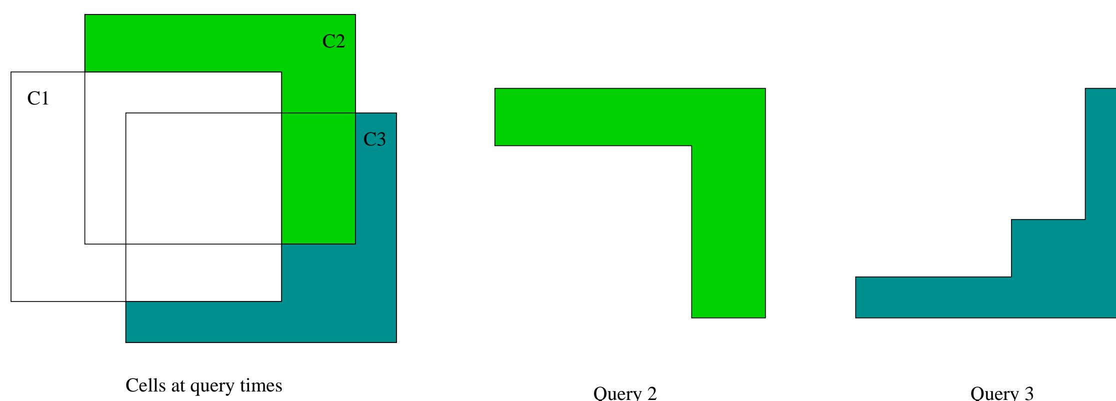


Figure 4.2: An example to motivate complement search.

As mentioned, a user's viewpoint is associated with a *disk cell*. Whenever the user moves out of its current disk cell, objects belonging to a new cell will have to be accessed, as illustrated in figure 4.2. Here, the user's frustum cell is initially within cell C1. When the user's frustum cell moves out of C1, data belonging to cell C2 have to be loaded. Intuitively, it doesn't make sense to load all objects belonging to cell C2 into memory, since there is a significant amount of overlap between cell C1 and C2. In fact, ideally, we should only load objects in the shaded

region of C2. Similarly, if C3 becomes the current cell, then only objects in the shaded region of C3 need to be accessed.

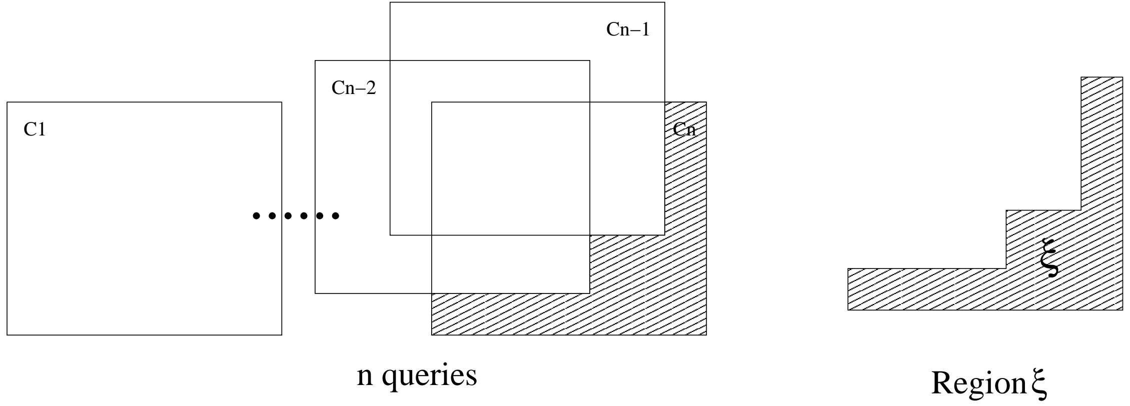


Figure 4.3: An example of the nth complement query.

As an example to describe the problem, figure 4.3 shows n consecutive query regions, $C_i, (i = 1, 2, \cdots, n)$. Region ξ, shaded in lines pattern in the figure, is the target region in which we need to find out the to-be-loaded objects. More precisely, we want to query for objects that overlap region ξ and have not been queried in previous queries, namely

$$Q_n = \{o_j | o_j \cap \xi \neq \phi \wedge o_j \notin \bigcup_{i=1}^{n-1} Q_i\} \ .$$

If there exists a method to test the overlapping between region ξ and a rectangular box, we can intuitively describe the algorithm as the pseudo-code shown in figure 4.4. Lines 3, 4, and 5 indicate that if the bounding box of the entry overlaps ξ, the traversal will proceed to its child node. However, as we will see, this tentative algorithm has some problems.

As figure 4.3 shows, the non-overlapped areas of cells are usually concave ge-

Algorithm TentativeComplementSearch (T, ξ, H)

1. if T is not a leaf node /* search subtrees */
2. for each entry E of node T do
3. if (BB(E) **Overlaps** ξ)
4. Invoke **TentativeComplementSearch** on the sub-tree
5. associated with entry E
6. else /* search leaf node */
7. for each entry E of node T do
8. if (BB(E) **Overlaps** ξ)
9. if BB(E) **Overlaps** none of $C_1, \ldots, C_i$ in H
10. E is a qualifying record

Figure 4.4: The TentativeComplementSearch algorithm.

ometries, so it is difficult to describe such a region in each retrieval operation. It is also difficult to search for objects overlapping such an area (line 3 and 8 in figure 4.4), which is usually a concave shape, in an R-tree, as the original search algorithm employs only box-shaped regions. One possible solution to this problem is to construct a BSP-tree [86] containing all the faces of region ξ, and test the overlapping of the bounding boxes in R-tree against it. But this would consume much CPU time due to the numerous numeric operations required by BSP tree. Therefore, the TentativeComplementSearch algorithm is impractical to be implemented.

In this section, we shall propose a novel search method for R-tree that retrieves only objects in the non-overlapped regions at very low cost. We refer to the search algorithm being proposed as the CSearch(Complement Search) algorithm. Essen-

tially, the algorithm requires us to maintain a history of cells H = $\{C_1, C_2, \ldots, C_i\}$. Given that we want to load objects belonging to a new cell C, the problem becomes one of retrieving objects whose bounding boxes overlap C but do not overlap any of the cells in H. Referring to Figure 4.2 again, if C_3 is the current cell whose objects we want to retrieve, then objects that overlap C_3 but not C_1 and C_2 are the ones that we are interested in.

Algorithm CSearch (T, C, H)

```
1.   if T is not a leaf node /* search subtrees */
2.      for each entry E of node T do
3.         if (BB(E) Overlaps C)
4.            if BB(E) COverlaps all of C_1, ..., C_i in H
5.               Invoke CSearch on the sub-tree
6.               associated with entry E
7.   else /* search leaf node */
8.      for each entry E of node T do
9.         if (BB(E) Overlaps C)
10.           if BB(E) Overlaps none of C_1, ..., C_i in H
11.           E is a qualifying record
```

Figure 4.5: The CSearch algorithm.

Figure 4.5 gives the algorithmic description for the complement search. One of the main operations is the **COverlap** (Complement Overlap) operation between two regions. We define the complement overlap between these two regions as follows: given a cell A, the space not contained in A is the complement of A, which is denoted as $\bar{A}$. If a bounding box BB (of a virtual object or a group of objects)

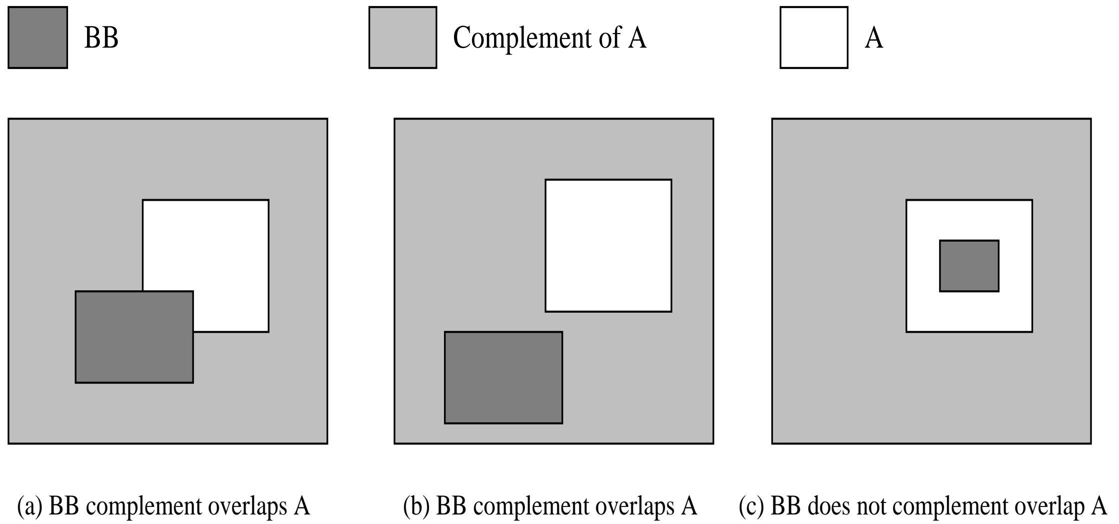


Figure 4.6: Complement overlap relations.

overlaps (or intersects) $\bar{A}$, then we say that BB complement overlaps A. In Figure 4.6, the bounding boxes of (a) and (b) complement overlap A, but that of (c) does not. In Figure 4.5, we use T to denote an R-tree node, and use E to denote an entry of the R-tree node.

The algorithm CSearch is conservative when accessing non-leaf nodes to guarantee that no objects in the new cell would possibly be lost. As the pseudo-code shows, it is obvious that complement-overlap checking will only happen when the bounding box overlaps the current cell C, thus CSearch will not access more R-tree nodes than the original algorithm. The algorithm will terminate the recursive search at the R-tree nodes with bounding boxes that are completely contained in all the $i + 1$ cells (i history cells + current cell). When the size of the cell is large enough as compared to the average size of scene objects, and the overlapping between two cells of consecutive query operations is also large, CSearch can stop searching at high level nodes in the R-tree, saving a large percentage of disk

accesses. At the same time, CSearch retrieves all data objects inside the current cell in one traversal of the R-tree, without accessing those that already have been retrieved in the past frames, minimizing the result set of objects that have not been accessed.

The algorithm of complement-overlap being applied in the CSearch algorithm is simple. According to its definition, complement-overlap equals to “not completely contained in”. If two points P1 and P2 are inside a cell, all points on the line segment between P1 and P2 are also inside the cell. As the boxes and cells are convex, if all vertices of a box are contained in a cell, all the points in the box are also in it. If there exists one vertex outside the cell, COverlap is true, otherwise, it is false.

Figure 4.7 shows the difference between TentativeComplementSearch and CSearch algorithm. As the figure shows, TentativeComplementSearch algorithm attempts to find nodes that overlap region ξ, while the CSearch algorithm performs the overlap checking in another way. The latter tests if a bounding box overlaps ξ by testing if it overlaps the “inside half space” of cell C, and the “outside half space” of the history cells. By doing so, the CSearch algorithm is able to avoid the expensive overlap checking for complement region ξ.

If we denote the cells that a user accesses as $C_1, C_2, C_3, \ldots$, based on the CSearch algorithm, the retrieval engine will issue the following queries to the database (R-tree): C_1, $C_2 - C_1$, $C_3 - (C_1 \cup C_2)$, $C_4 - (C_3 \cup C_2 \cup C_1)$, $\ldots$, $C_{i+1} - (C_i \cup \ldots \cup C_2 \cup C_1)$ and so on. As a comparison, a traditional method would issue queries C_1, C_2, C_3, $\ldots$, to the database. For a complement search like $C_{i+1} - (C_i \cup \ldots \cup C_2 \cup C_1)$,

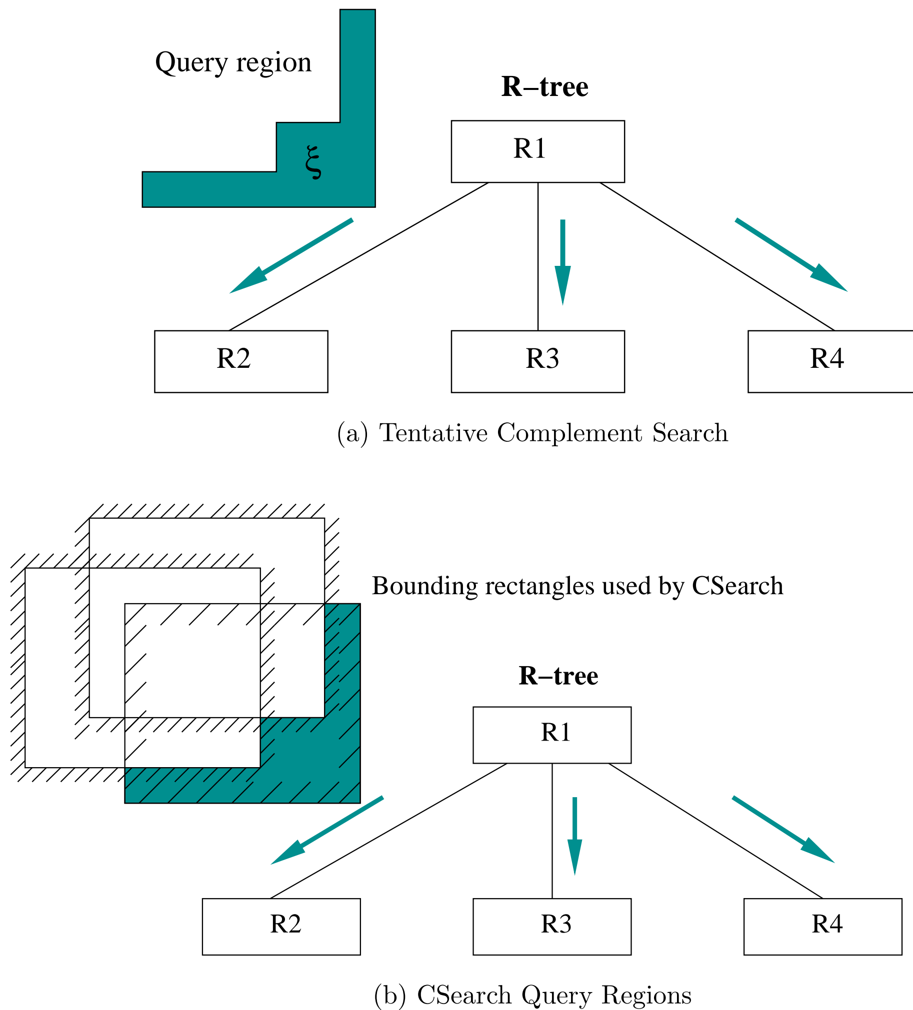


Figure 4.7: Comparing CSearch with TentativeComplementSearch algorithm

we can remove any cells from $\{C_1, C_2, \ldots, C_i\}$, if the bounding boxes of all their objects do not overlap C_{i+1}. Such cells have no effect on the query result because objects overlapping them cannot overlap C_{i+1}. Thus, before sending the query to the database, a filtering operation can be conducted on the cell list, so those cells not interfering the current cell do not need to be considered in the CSearch algorithm. In our prototype walkthrough system, the number of cells in the history to be maintained is fewer than twenty in most cases. With such short cell lists, the

CPU cost on the extra COverlap and Overlap testing is negligible.

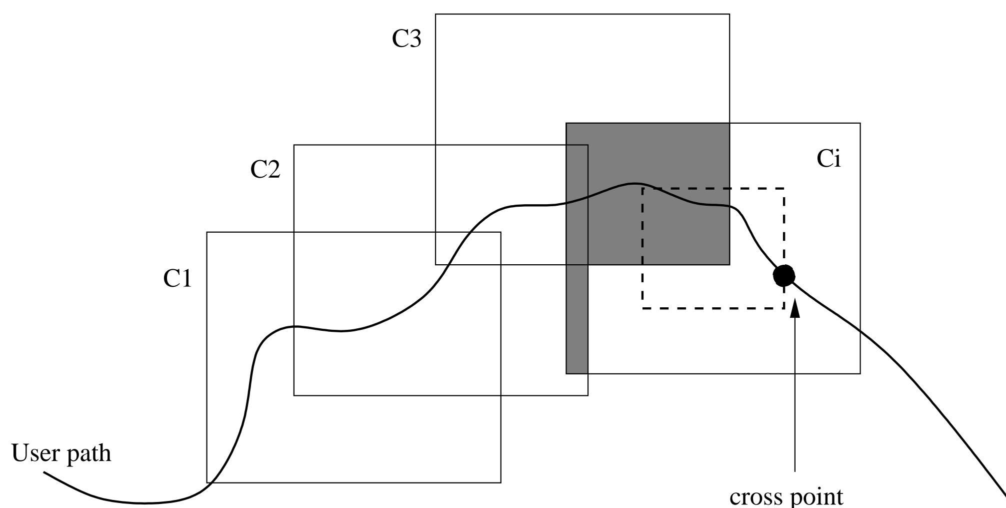


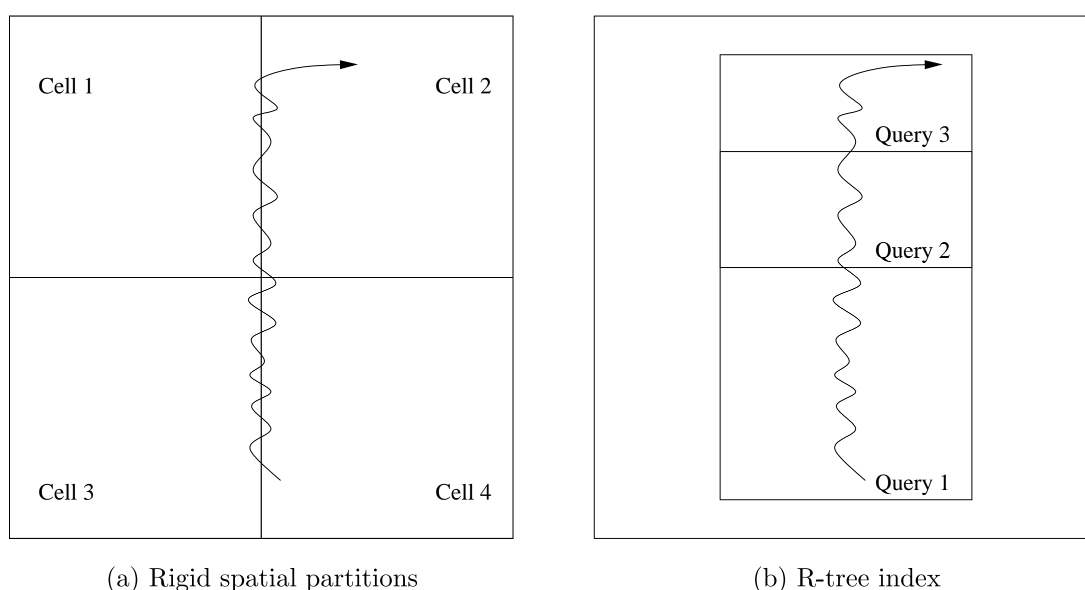
Figure 4.8: Objects to be kept in memory.

As a user "steps" out of a cell boundary, a complement search returns a new result set. The complement search algorithm guarantees that the result sets will have no overlap. However, as the object buffer in the main memory gets filled up, old objects should be freed to make space for objects of new cells. Once the buffer is full, we need to remove the previous results and to keep only objects of the current cell in the buffer. That is, as shown in Figure 4.8, only objects in C_i need to be retained in the buffer. Unfortunately, since $C_1, C_2, \ldots, C_{i-1}$ may overlap with C_i, to remove the results of these cells may also remove objects in the overlapped regions (shown as the shaded area in the figure). A direct method to deal with this problem is to delete from the buffer the objects that do not overlap the latest cell, while maintaining the objects overlapping it. As shown in Figure 4.8, objects overlapping the latest cell C_i are kept in memory. After this operation, the object buffer contains only the objects of cell C_i, of which the user is currently walking out.

4.2.3 Regular Grids vs. 'R-tree + CSearch'

People may argue that regular grids can be used to organize the data on the disk and be queried for walkthrough. To fully demonstrate the advantage of using R-tree and CSearch over regular cells, we compare the two partitioning schemes with a few examples in this section.

In a static partitioning approach, we can split the VE space into rigid spatial partitions or grids/cells of the same size. However, in the dynamic approach, instead of splitting the VE, we use an R-tree index to index all objects in the VE and create cells dynamically at runtime.



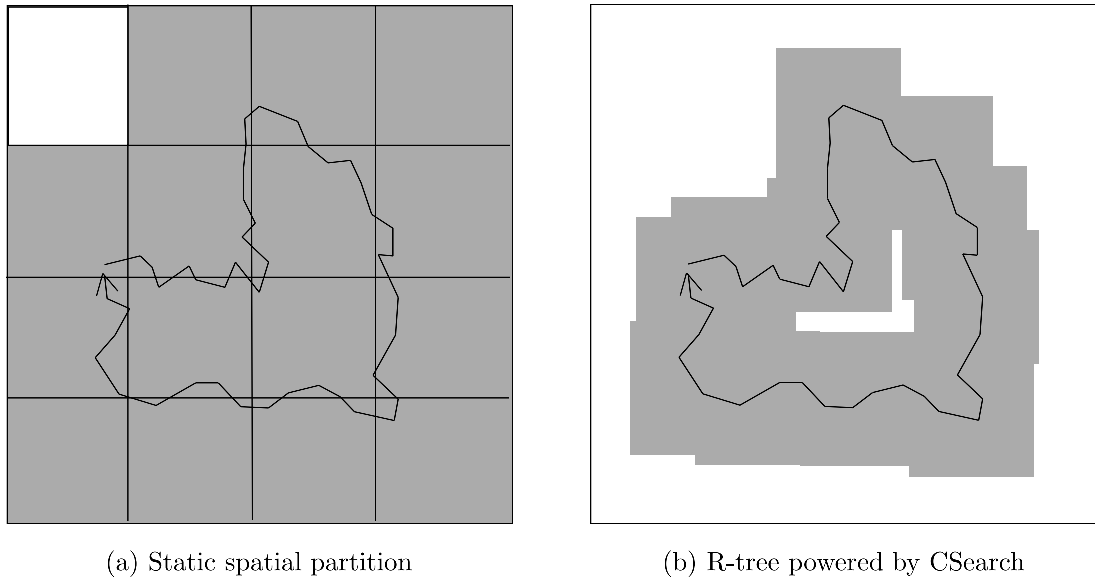
(a) Rigid spatial partitions (b) R-tree index

Figure 4.9: Example to illustrate the number of swaps needed for rigid spatial partitions and R-tree indexing.

Figure 4.9 shows the top view of a very simple scene, in which a user walks through along a curved path. Suppose the system has the memory which could only store the data of one cell, a swap has to be performed every time the user

crosses a cell boundary. In figure 4.9(a), many swaps have to be performed when the user crosses cell boundaries for many times. Once the environment is rigidly subdivided, cell boundaries are always present at fixed positions in the environment and excessive swapping (and performance degradation) always occurs at these positions when the user crosses the cell boundaries frequently. Performance therefore becomes dependent on the physical position of the user path. Using an R-tree partitioning scheme and the CSearch algorithm, there are absolutely no fixed subdivisions of the environment. As such, we can issue queries centered to the user's current position to load necessary parts of the environment into main memory. As shown in Figure 4.9(b), under a dynamic partitioning scheme, it is possible to take only 1 swap in order to perform the same walkthrough. The swap occurs at Query 2. No swapping is required in Query 3 because we have enough main memory to store the results of both Query 2 and 3.

Suppose now we have enough memory to load all cells of the environment, the dynamic approach will require less memory to load the environment. For a given user path, the shaded region in Figure 4.10(a) shows the area that is needed to load into memory for a rigid cell subdivision method. In comparison, the shaded area in Figure 4.10(b) shows the area needed to load into memory for the R-tree indexing plus **CSearch** algorithm approach. From both figures, we see that the dynamic spatial partitioning approach requires less memory space to load the environment for the given user path. The figure apparently indicates that more I/O operations are required by the grid partitioning approach. Another problem with the rigid cell subdivision method is it also requires the splitting of objects if they lie across



(a) Static spatial partition

(b) R-tree powered by CSearch

Figure 4.10: Example to illustrate the amount of data to be loaded into memory for rigid spatial partitions and R-tree + CSearch.

cell boundaries.

From the comparisons, we found the R-tree index powered by CSearch algorithm is more suitable for the organization of a large disk-resident scene.

4.2.4 Distance-priority-based Replacement Policy

Because of the limited memory size (compared to the large VE), index nodes that are cached in the memory may have to be replaced. We propose a new priority-based replacement policy that is based on walkthrough semantics.

In the R-tree, since a search process starts from the root node (at level 0), it is obviously beneficial to keep the root node in memory all the time. As to nodes on lower levels, we assign an integer priority number P to those being cached, according to the following rules:

- Lower-level nodes have larger P values
- Nodes at the same level in the R-tree have same P values.

We note that a smaller P value indicates higher priority.

Our cache replacement policy is based on the *block-distance* between two axis-aligned boxes,

$$A = \{[X^a_{min}, X^a_{max}], [Y^a_{min}, Y^a_{max}], [Z^a_{min}, Z^a_{max}]\}$$

and

$$B = \{[X^b_{min}, X^b_{max}], [Y^b_{min}, Y^b_{max}], [Z^b_{min}, Z^b_{max}]\}$$

which is defined as follows:

$$DIST(A,B) = \begin{cases} 0 & \text{if A overlaps B} \\ max\{dist_x, dist_y, dist_z\} & otherwise \end{cases}$$

where

$$\begin{aligned} dist_x &= min\{|p_a - p_b|, p_a \in [X^a_{min}, X^a_{max}], p_b \in [X^b_{min}, X^b_{max}]\}, \\ dist_y &= min\{|q_a - q_b|, q_a \in [Y^a_{min}, Y^a_{max}], q_b \in [Y^b_{min}, Y^b_{max}]\}, \\ dist_z &= min\{|r_a - r_b|, r_a \in [Z^a_{min}, Z^a_{max}], r_b \in [Z^b_{min}, Z^b_{max}]\}. \end{aligned}$$

It is well known that the effectiveness of a cache replacement policy depends largely on the access pattern. In a walkthrough application, a user normally walks across the whole scene and turns left or right or backward sometimes. Since the R-tree index is organized to represent the spatial subdivision structure, when a user

walks out of a high-level bounding box at some time and is already quite distant from it, he(she) is not likely to access it or its descendants in the near future. Based on this observation, the replacement can be made in consideration of the following information:

1. The QN value of each cache entry represents how long the entry resides in the cache. After each search process, increment by 1 the value QN of each entry in the cache that is available for replacement.

2. If there is a free entry in the cache, load the required node into the free entry, set its priority value to be the priority number of its level P, and set the QN value of its entry to 0.

3. If there is no free entry in the cache, find an entry with the largest priority value. If multiple entries of the same largest value exist, choose the entry which is least recently used and whose node's bounding box has greatest block-distance away from the current query cell. Replace it with the required node, then set this node's QN value to 0 and priority value to be the priority number of its level, P.

In accordance with the above points, we define a function

$$f = \mathcal{F}(P, QN, DIST(Q, BB)),$$

where P is the initial priority, QN is an integer number representing how long the entry resides in the cache, and $DIST(Q, BB)$ is the block-distance of the current

query box (or cell) Q and the bounding box BB of the node. The f value is computed for each entry in the cache considered to be replaced. The function is defined in such a way that an entry with the highest f value will be replaced. There are many ways to define the function f to meet the above conditions. For simplicity in our implementation, we use the following definition:

$$f = \alpha \cdot pp + \beta \cdot pd + \gamma \cdot pl,$$

where pp, pd and pl are normalized values of the priority p, *DIST(Q, BB)*, and QN respectively. α, β, and γ are weight factors and $\alpha + \beta + \gamma = 1$.

This policy, namely distance-priority-LRU policy, guarantees that:

1. High-level nodes have a higher tendency to remain in the cache.

2. Nodes which have not been accessed for a long time or distant from the current viewpoint will have a higher preference to be replaced;[1]

3. For nodes on the same level, the later a node is accessed, the more likely it is to reside in memory.

Considering the circumstance of walkthrough a large virtual environment, this distance-priority-LRU policy is expected to be superior to the traditional LRU scheme since a node that is currently distant from the user is not likely to be accessed in the near future. On the contrary, if a user takes a circular path and moves near to an area which was accessed long time ago, the distance-priority-LRU

[1] We note that this is in contrast with existing schemes that typically give higher level nodes higher priority.

algorithm will detect that the node is near to the user and should be kept in the cache. However, under the LRU policy, this node will be assigned a low priority, since it has not been accessed for a long time, and may be removed from the cache.

4.2.5 Prefetching Algorithm

When a user initializes a walkthrough session, the system will access the database by querying the R-tree index with a disk cell containing the initial view frustum. If the cell tightly bounds the view frustum, once the user moves, the view frustum will move out of the cell almost immediately, making it necessary to query the database with a new cell. To avoid the problem, we should use a disk cell whose dimensions are larger than those of the view frustum. On the contrary, if the dimensions of the cell are much larger than those of the view frustum, two new problems may arise. First, the result may be too large to be stored in main memory. Second, the retrieval may be slow, as more R-tree nodes (or disk pages) need to be accessed. To optimize the overall performance, we should tune the size of the cell to an appropriate one, so as to strike a balance between the query frequency and the size of the results.

When approaching the boundary of a cell, the user is likely to move out of the cell soon. Thus, before the user goes out of the cell, we need to prefetch data of another cell using a different thread. As shown in Figure 4.11, C_i is the current cell. C_2 is the prefetched cell for C_1 and C_3 is the prefetched cell for C_2. As C_1 and C_2 have large overlap, complement search algorithm can be applied during the prefetching of C_2. C_3 can also be prefetched complementing C_1 and C_2.

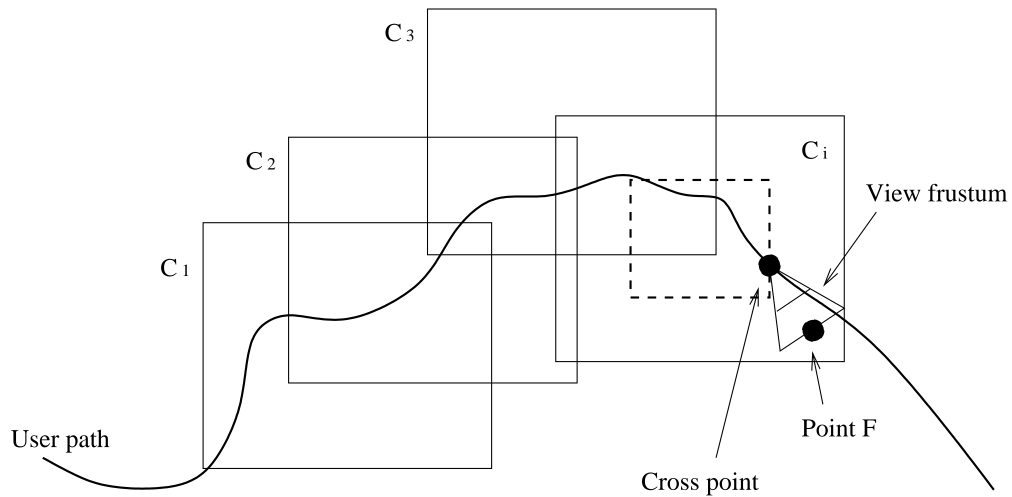


Figure 4.11: Prefetching objects

The main challenge with prefetching is to predict the position of the next cell, i.e. cell C_{i+1}, for C_i. As the user interactively navigates in the virtual environment, turning left and right frequently, it is very difficult, or even impossible to know where the user is going to visit in the next moment. However, by observation, the user will more likely move in the direction of the current view. So it is reasonable to set the central point, $F(x, y, z)$, of the far clip plane of the view frustum as the center of the new cell. An inertial term $I = k \times velocity$ is added to F to produce the final result:

$$F' = F + I = F + k \times velocity$$

where $velocity$ is the current velocity vector of the user and k is an adjusting factor. If a user moves quickly in the direction of velocity, the predicted center is further. Otherwise, if the velocity is low, it is more likely that the user may turn to other directions, so the prediction should be more conservative and nearer to the view point. If the user turns away from the predicted direction, the view frustum

should still be within the predicted cell. However, if the user's direction is not correctly predicted, the user will move out of the predicted cell in a short time. As a consequence, the frequency of queries increases under such circumstance.

Fortunately, as we have implemented the complement search query interface, the new cell used to query may have an overlap with the current cell which, in turn, implies that only a small number of objects need to be retrieved. If the user walks back into the old cells again, it is not necessary to issue a new query. This feature outperforms the walkthrough systems based on the static cells, where a user would experience a more serious slow-down, caused by the swapping of the two neighboring cells, when the user crosses the cell boundary back and forth frequently.

4.3 Optimizing GPU Performance

Object data returned from queries to the disk cell are stored in an object buffer in the main memory. However, it is not practical to send all these data to the GPU (Graphics Processing Unit) or graphics pipeline, as the object buffer may contain a large number of objects that are not visible in the current frame. Culling the object buffer using the view frustum (see Figure 4.12) will improve the performance of the GPU because fewer objects will be transmitted and rendered.

The culling process can take the advantage of using the bounding boxes of the cells and the objects. If the current view frustum does not overlap the bounding box of the cell, the objects in this cell will not overlap the frustum.

Let H be a frustum cell that bounds the view frustum. Essentially, what we

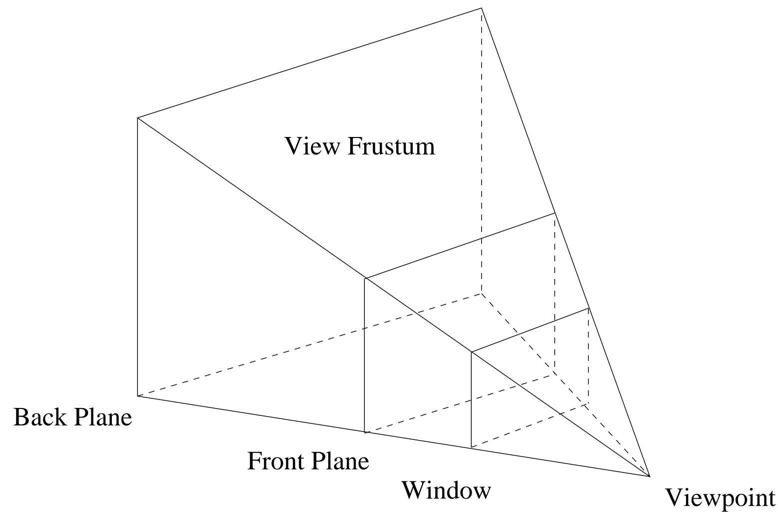


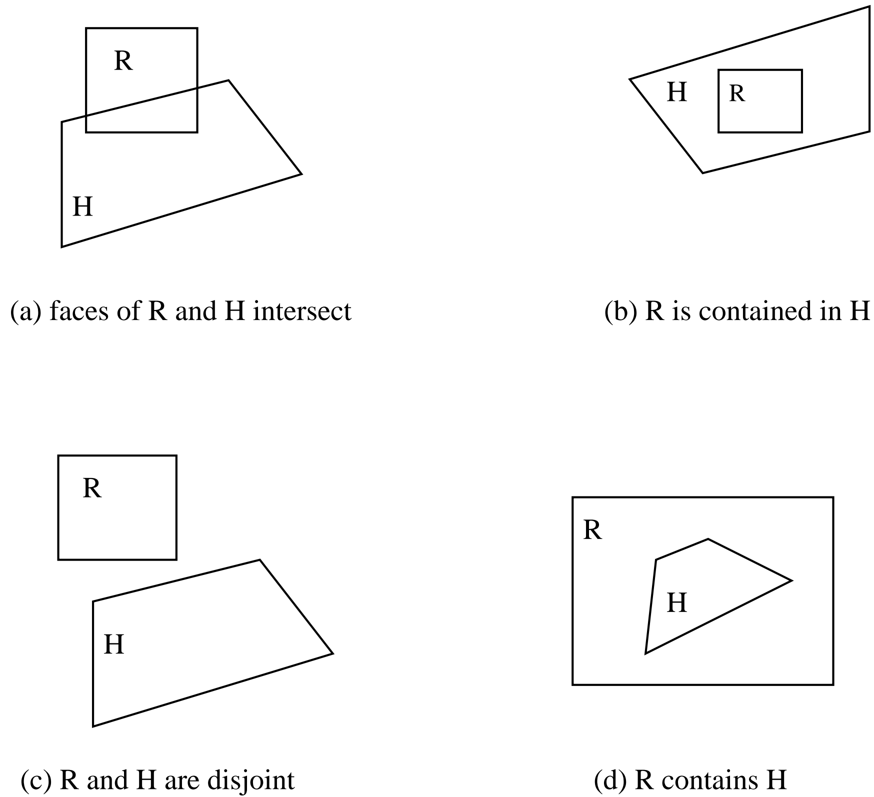
Figure 4.12: Culling the object buffer using the view frustum.

want is to send only objects in H to the graphics pipeline. To determine whether a 3-dimensional box R overlaps H, we need to consider the four possibilities shown in Figure 4.13(a)-(d) respectively:

1. $R \cap H \neq \phi, R \not\subset H, H \not\subset R$
2. $R \subseteq H$
3. $R \cap H = \phi$
4. $H \subseteq R$

First, we need to check the face-face intersection between H and R. If one intersection is found, it means that the two boxes overlap. Otherwise, there are three possibilities, corresponding to cases (2), (3) or (4) above. If there is one vertex of the R (H) inside H (R), the two boxes overlap. Otherwise, there is no overlapping.

For face-face intersection, since each face of H splits the whole 3D space into



(a) faces of R and H intersect

(b) R is contained in H

(c) R and H are disjoint

(d) R contains H

Figure 4.13: Spatial relations between a convex shape H and a rectangle R.

two half spaces, we can first check whether all vertices of R (H) are in the same *positive half space* where the normal of the face points to. If they are, as shown in Figure 4.14, R (H) does not intersect any face of H (R). When the size of each object is much smaller compared to the size of the H, most bounding boxes in the lower level of R-tree conform to this condition. So the algorithm can be accelerated.

The algorithm is shown in Figure 4.15. Given a frustum cell H having s polygons, denoted as p_1, p_2, ..., p_s, and t vertices, denoted as v_1, v_2, ..., v_t, and an n-dimensional rectangle R, with vertices r_1, r_2,..., r_8, and faces f_1, f_2, ..., f_6, it finds if H overlaps R.

This algorithm accurately checks if a bounding box overlaps the frustum cell. For object data stored in memory, we first check whether the frustum cell overlaps

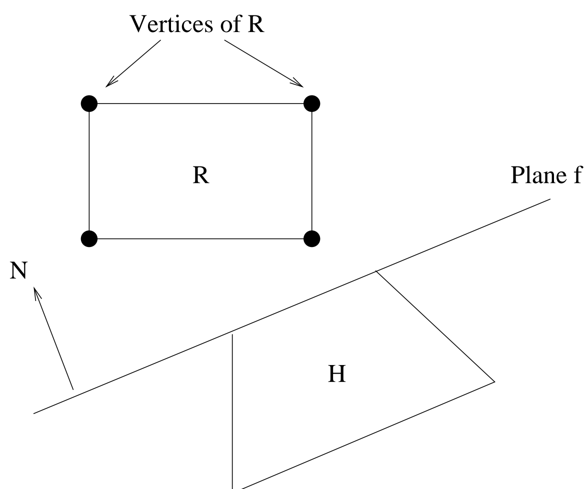


Figure 4.14: Intersection checking with vertices or R.

the cell-level bounding boxes. If it is true, the search process will go on with all objects in the result set corresponding to that cell. Otherwise, the whole cell does not need to be rendered. As a result, a large portion of objects in memory can be filtered, saving a lot of burden on graphics engine.

4.4 Experimental Results

We implemented a prototype system called REVIEW (Real-time VIrtual Environment Walkthrough) that employs the proposed techniques. The system was built upon a Silicon Graphics Octane workstation running IRIX 6.5, with 400 MB memory. Since the memory size is large, we set an upper limit of memory size to 20MB for the system.

We generated a synthetic dataset to simulate a large cityscape. There are about 900,000 virtual objects/buildings (Figure 4.16(a)), requiring about 200 MB of hard

Algorithm ViewCulling

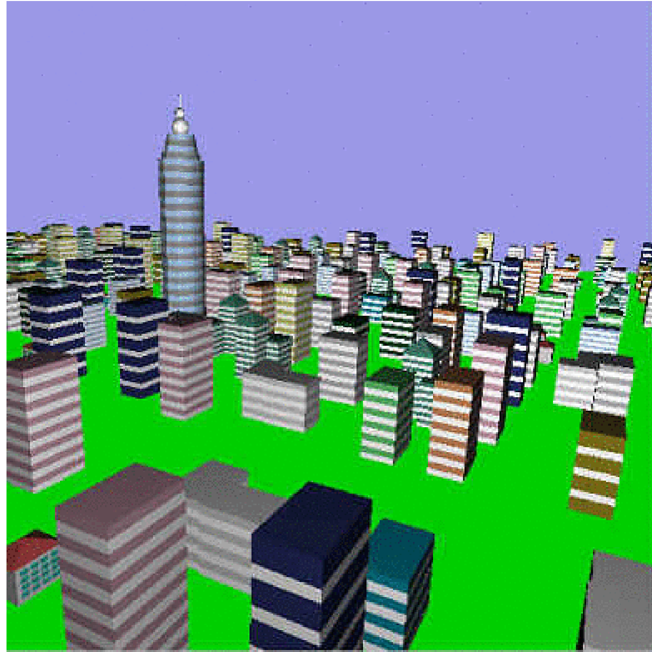
```
1.   for (each face p_i of H)
2.        if (all vertices of R are on the positive side of p_i)
3.             Return FALSE
4.   if (any vertices of R is inside H)
5.        Return TRUE;
6.   else if (any vertices of H is inside R)
7.        Return TRUE;
8.        else
9.             for (each face p_i of H)
10.                 if (p_i intersects one of f_1, f_2, ..., f_6)
11.                      Return TRUE
12.            return FALSE
```

Figure 4.15: The view culling algorithm.

disk space (inclusive of R-tree index), and more than 1 GB of memory space if fully loaded into main memory (in scene graph format). The distance between objects is 10 to 30 meters long, which is quite similar to realistic cases of a city. As figure 4.16(b) shows, the visual output of the REVIEW system during interactive walkthrough is quite good.

The parameters of the view frustum include following: The eyesight of a user, or the *depth* of the view frustum, is set to be one kilometer long, which is consistent with the real walkthrough. The horizontal and vertical *field-of-views* are both set to 45 degrees.

To test the effectiveness of various techniques depicted in this chapter, we ran



(a) Bird's eye view of the data set

(b) A snapshot of the walkthrough

Figure 4.16: Screen shots of a large cityscape.

experiments under different system configurations. For clarity of the description, we will use the following abbreviations:

Optimal A full-fledged REVIEW system configuration, in which *complement search*, *index caching*, and *prefetching* techniques are applied.

NC A REVIEW system configuration without index caching, in which only *complement search* and *prefetching* are applied.

NP A REVIEW system configuration without prefetching, in which only *complement search* and *index caching* are applied.

As reference, we also implemented a version that makes use of traditional R-tree query search, i.e., the search is based on box-shaped queries without any optimization. We shall refer to this scheme as BOX.

In REVIEW, the disk retrieval cell is larger than the frustum cell. We represent this by the concept of a *scale factor* (SF). If the frustum cell size is S, then disk cell is set to SF$\times S$.

We conducted two groups of experiments. The first group of experiments allows us to fine-tune our configurations to find the optimal prefetch factor, cache size and cache policy. The second group illustrates the performance improvements of REVIEW to the traditional system. The systems are tested with the following default settings, unless stated otherwise:

1. The default prefetching factor k, except NP, is 0.8

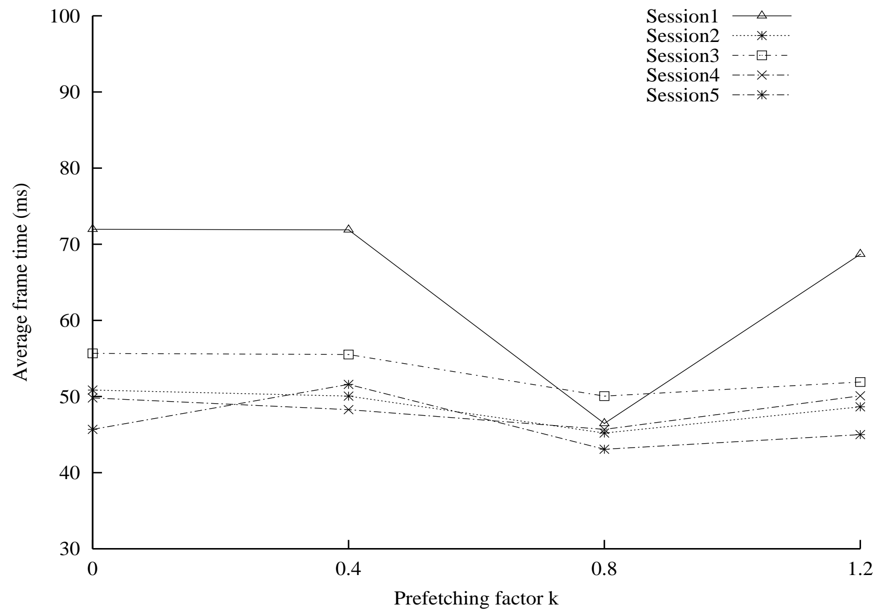
2. The default index cache size is 1MB

3. The default index cache replacement policy is *distance-priority-LRU*, with the weight factors set to $\alpha = \beta = \gamma = 0.333$.

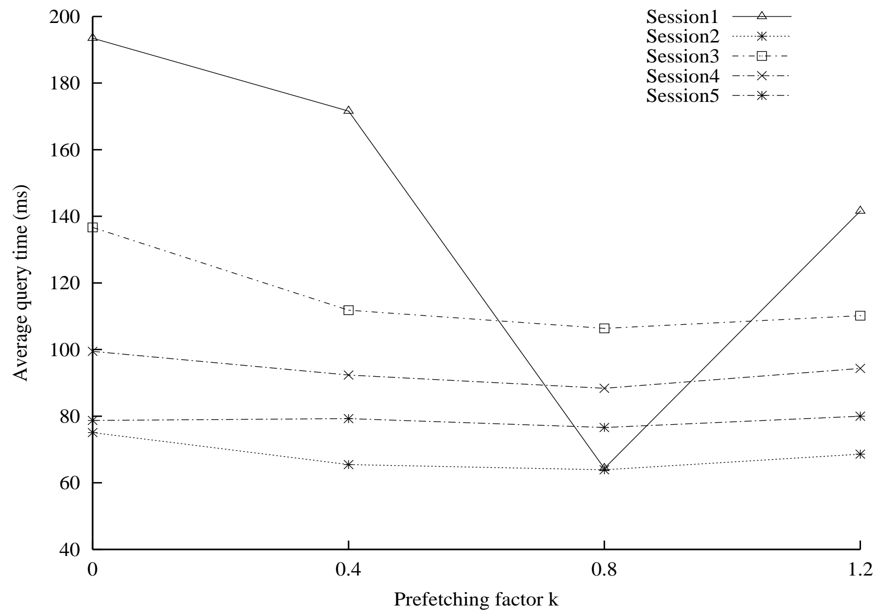
4.4.1 Tuning the parameters in REVIEW

We note that there are several parameters in REVIEW that have to be tuned. First, in the prefetching algorithm, its effectiveness depends largely on the semantics of the user walkthrough and the algorithm itself. To fine-tune the prefetching factor k in the prefetching algorithm, we used several user sessions to find an optimal k value. Figure 4.17 shows the results. These user sessions have different motion patterns. The fast, slow, turning, backward, and normal patterns were tested in the five sessions respectively. In the figure, each curve represents an individual walkthrough session.

Frame time is defined as the cycle time between two consecutive rendering operations. The time for database query, memory data manipulation, rendering and other overheads are all included in frame time. A real-time walkthrough requires the frame time to be shorter than 50ms, i.e., the frame rate higher than 20 frames per second. (This is due to physiology of the human eye.) In Figure 4.17(a), all sessions have a minimal average frame time around point where $k = 0.8$. As different sessions have quite different motion patterns, the effect of k is also quite different. In session 1, since the user moves rapidly, it is crucial to make an accurate prediction of the position of the new cell. If the k is too small, user will move out of the new cell more frequently, generating more prefetches. On the contrary, if the prediction is too far away from the current position, as less overlap can be



(a) Average frame time vs. k



(b) Average query time vs. k

Figure 4.17: The effect of the prefetching factor k to system performance

obtained between two consecutive cells, the query time will increase and so will the frame time. Average query time are shown in Figure 4.17(b). The query time also has a minimum value around the point where $k = 0.8$.

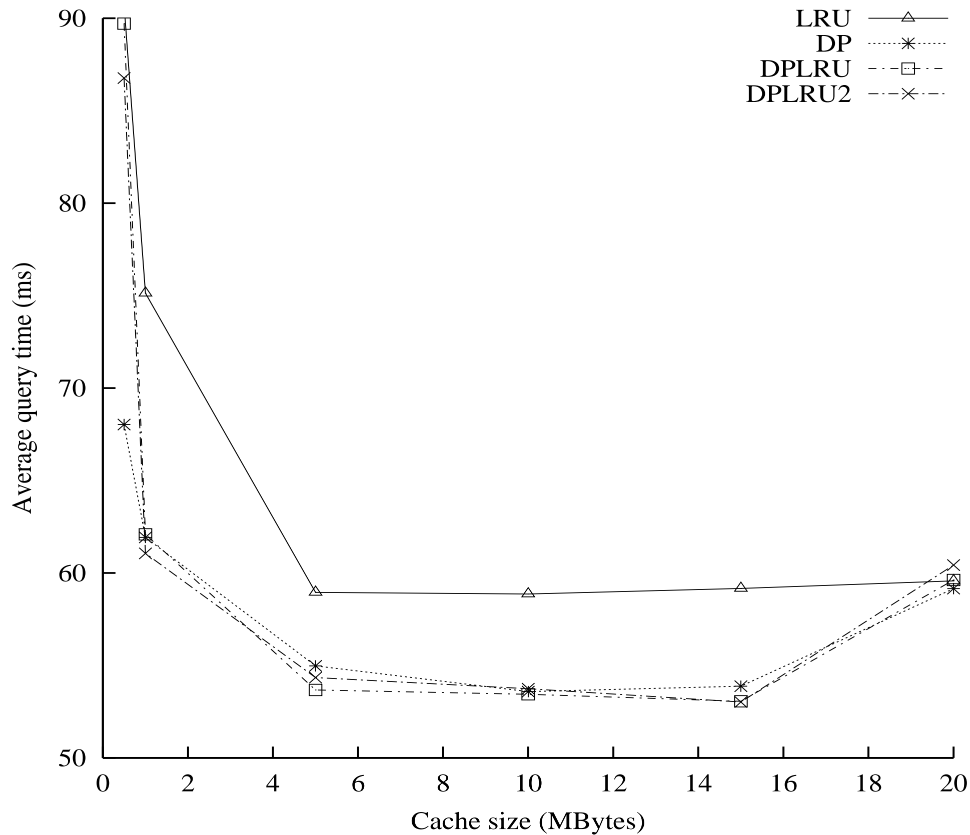


Figure 4.18: Cache performance with various cache sizes

Next, we need to tune the parameters used in the replacement policy. Figure 4.18 illustrates the results of index cache performance with different cache replacement policies under various cache sizes. The figure shows that the *distance-priority-LRU* scheme performs better than LRU when cache size is smaller than 20 MB. For cache size between 5 to 15 MB, the best setting is the DPLRU, where $\alpha = \beta = \gamma = 0.333$. As the cache is implemented with software, it takes more

than *O(1)* time for finding a cache entry, as opposed to hardware implementation. Therefore, the overhead caused by the software implementation of the cache offsets the performance improvement at large cache sizes. This explains the increase in query time at cache sizes larger than 15 MB.

4.4.2 Performance improvements of REVIEW

Results of rendering frames

The metrics for measuring the quality of a walkthrough are the frame time and the smoothness of the walkthrough. The smoothness of the walkthrough can be represented by how much each frame time varies from the average frame time. A walkthrough with a small average frame time and a small variance is considered of good quality. Both the average frame time and the frame time variance of the REVIEW system are smaller than those of the BOX system. In addition, the frame time of REVIEW meets the requirement of real-time walkthrough.

The user positions and orientations are recorded during different walkthrough sessions. In the experiment, one recorded user session is used as the user path for all systems.

As shown in Figure 4.19, for different cell sizes, the average frame time of the traditional system is much longer than the optimized system. For the traditional system, as the cell size decreases, more queries have to be issued to the database. Moreover, as the overlaps of the cells used in the query are not considered, the average frame time increases as the cell size decreases. In contrast, for the configurations which implement complement query interface, i.e. *Optimal*, *NC*, and

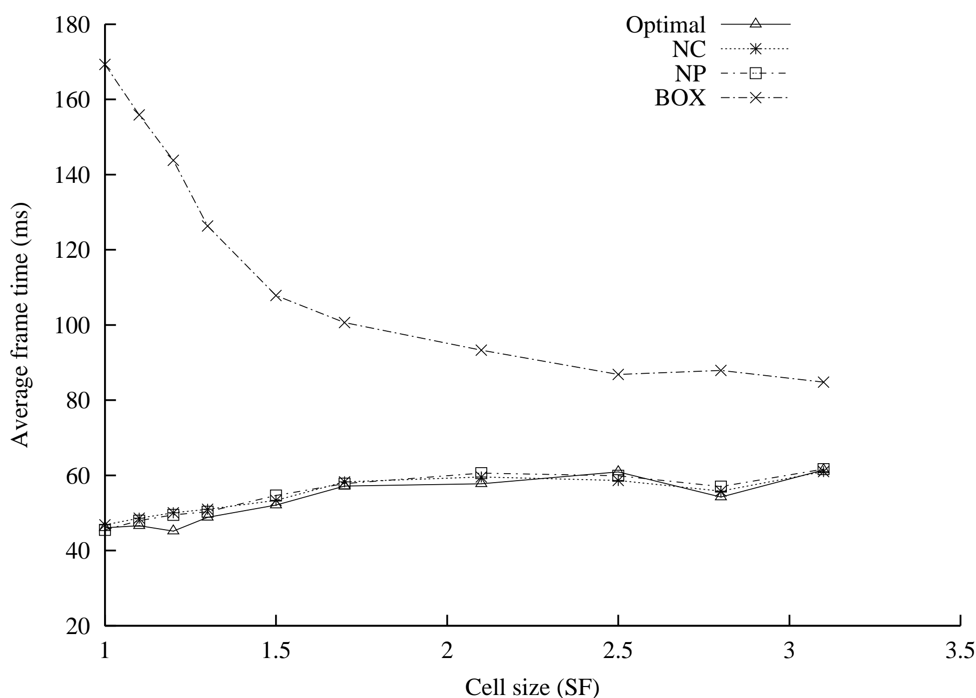


Figure 4.19: Average frame time for traditional and optimized systems

NP, the queries will only return data in non-overlapped areas. Therefore, as the cell size decreases, the average frame time does not increase. This indicates that the REVIEW system is less sensitive to changes in cell sizes than the BOX system. From the figure, we can also see that the rendering frame rate of REVIEW is higher than that of the BOX. As the cell size increases, the average frame time of the BOX system decreases. This does not mean that the walkthrough quality of the BOX system increases. The reason for the decrease in frame time is that fewer queries are issued to the database.

As the query boxes become larger, the search time per query is also larger. A user will experience a serious "pause" during each query. Hence, the walkthrough effect is not better. This is confirmed in Figure 4.20. In the figure, the variance of

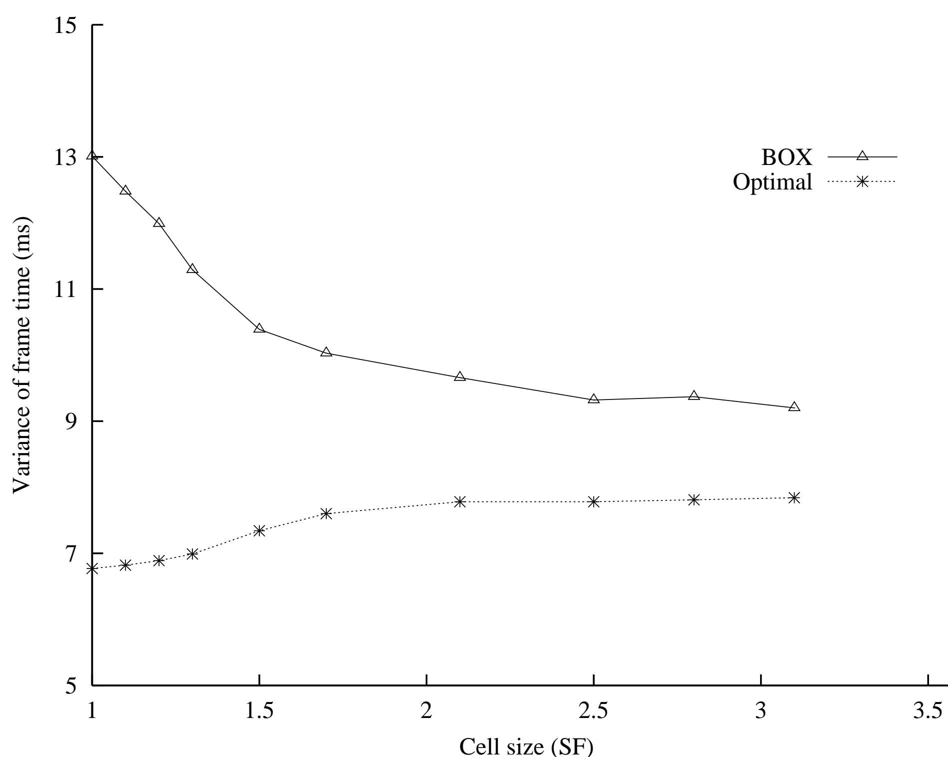


Figure 4.20: Variances of average frame time

the average frame time of the BOX is larger than that of the Optimal configuration. This indicates that the BOX system's frame time varies more than Optimal configuration frame time, giving a choppy visual effect. In contrast, the Optimal configuration frame time has lower variation and gives a more constant frame rate. The results also show that caching and prefetching have less impact on the average frame rate than the complement search algorithm.

Figure 4.21 shows the results on the rendering time for each frame when a user path is applied to an Optimal system and a BOX system. Both of them use the same cache size of 1 MB. The results show that the Optimal system has shorter rendering time and much smoother frame rate. Since the complement search algorithm returns smaller result set, the Optimal system also needs less

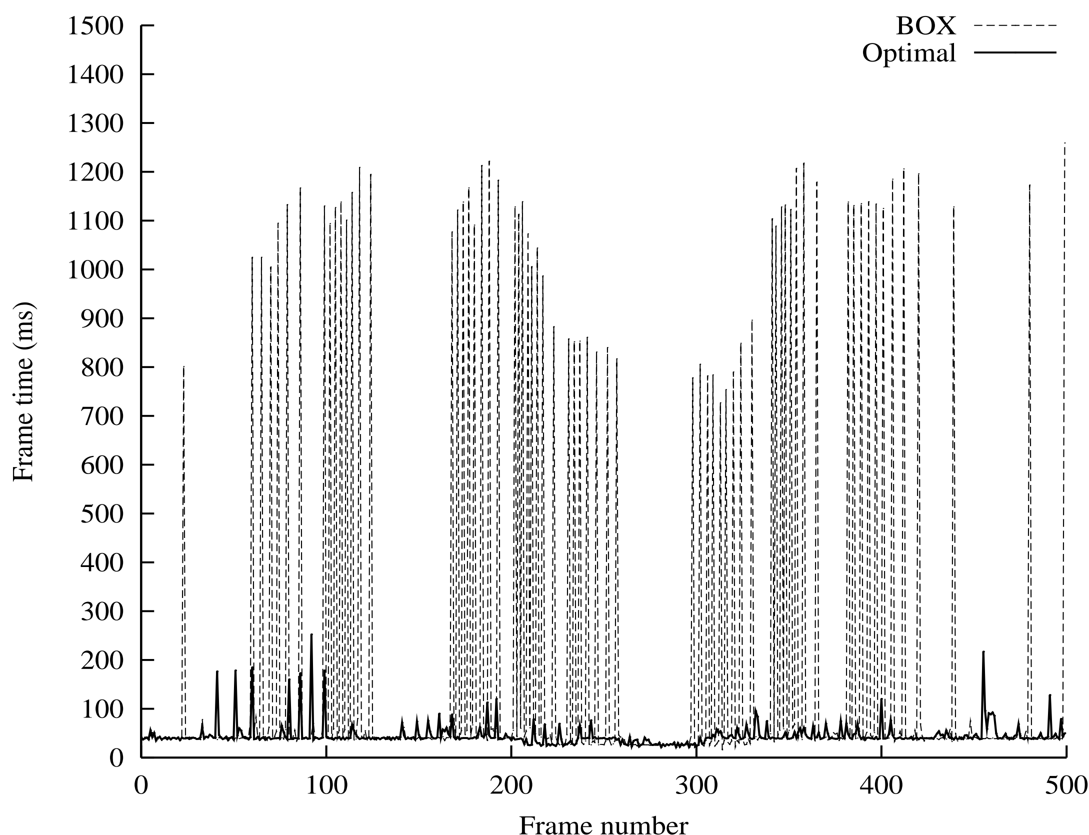


Figure 4.21: Rendering time for each frame

time to transfer the result set into the scene graph structure, so the change in rendering frame time is much smaller on each query.

Results of I/O

Figure 4.22 shows the average search time of user walkthrough sessions in different sized databases. It is apparent that the Optimal system outperforms the BOX system in databases of different sizes.

In table 4.1, the average disk accesses per query are shown for five different walkthrough sessions. The disk I/Os of the Optimal system varies from 9% to 21% of those of the BOX system. Therefore, it is apparent that the Optimal system

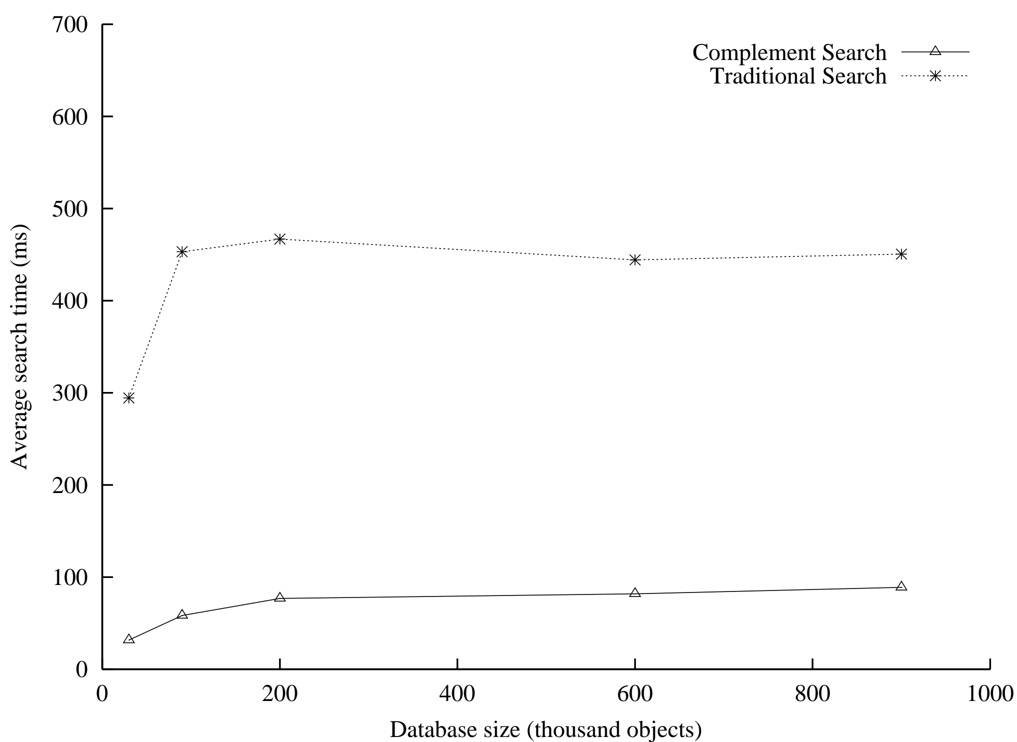


Figure 4.22: Average search time

performs better than the BOX in disk I/Os.

User session #	BOX Disk I/Os Per Query	Optimal Disk I/Os Per Query
1	1877.75	410.95
2	1900.94	169.96
3	1963.73	265.88
4	2018.24	201.89
5	1934.57	212.48

Table 4.1: Disk I/Os Per Query

Results of optimizing GPU performance

Table 4.2 shows the results of the view frustum culling algorithm discussed in section 4. The Optimal system runs this algorithm to remove irrelevant objects before sending the rest to the graphics engine. Therefore, the algorithm tries to

remove as many objects as possible to reduce the workload on graphics subsystem. The left column of the table contains various cell sizes, while the right one shows the respective average percentage of objects that are culled away before the rendering. The data in the table illustrates that a large percentage of object data in memory can be filtered and need not to be sent to the graphics engine. The percentage increases as the cell size increases. This is because when the cell size increases, more irrelevant data are retrieved into the memory, so the algorithm can find more irrelevant objects in the memory buffer.

Cell Size	Average Objects Removed (%)	Average Time Reduced (%)
1.0	91.34	34.93
1.2	92.26	46.13
1.5	92.93	50.53
2.1	93.99	61.02
2.5	94.86	65.16
3.1	95.33	90.77

Table 4.2: Results of view frustum culling

4.5 Summary

In this chapter, we have explored the problem of designing effective walkthrough system for a very large virtual environment using spatial techniques. For the I/O bottleneck, we have designed a novel search algorithm that minimizes access to secondary storage, a novel distance-priority-based cache replacement policy and a prefetching technique based on walkthrough semantics. For the bottleneck at the graphics processing unit, we have also proposed a view frustum search algorithm that loads the graphics engine with data that are visible in the view frustum. All

these techniques are based on the spatial relations among objects and the user's position/orientation. We have implemented REVIEW, a prototype walkthrough system and evaluated its performance on a very large virtual environment. Our results showed that the proposed techniques can sustain near-constant, if not constant, real-time frame rate and improve the visual effects.

Chapter 5

Virtual Walkthrough Using Visibility Techniques

The REVIEW system uses a spatial access method to retrieve data from the graphical database. With careful design, spatial access methods can perform well and generate good visual output. However, it has the inherent problem of "being spatial, rather than visual". For example, consider the case of visualization of a spatially large scene of world. If a model of a large object is located at a very distant position and does not overlap the query box of the spatial query, it might not be considered as a candidate of the result. Such cases do occur, for example, in a virtual environment of a landscape model, when a huge mountain is very distant from the viewer (user). A spatial query may not see it in the result set as it is too far away, though the mountain should be visible from anywhere. As the size of the query boxes is limited, distant and yet visible objects which do not overlap the query boxes are plainly pruned in the search. In this case the "eyesight" of the

viewer is always restricted by the spatial query box. In other words, the spatial access methods cannot adequately address the visibility problem without performance penalty. One may argue that a larger query box can be used to retrieve the distant and yet visible objects. Unfortunately, spatial based methods will never know how large the query box should be, in order to guarantee that all visible objects are included. Moreover, as we shall see, a large query box may retrieve many irrelevant objects.

We note that in complex virtual environment models, such as urban areas, architecture models, and mechanical CAD models, access methods based on spatial indexing tend to retrieve many irrelevant data. Invisible objects which fall into the query box are often included in the result, wasting disk I/O operations and GPU (Graphics Processing Unit) time. Past research in the graphics area found that culling invisible objects before the rendering operation is very effective in improving the memory-based rendering performance [16]. As a matter of fact, there has been a lot of research work in visibility computing, which tries to find the visible objects when the viewer is located at a viewpoint or within a viewing cell. Using the visibility data to render visible objects, is known as *visibility culling*. Discussing how to find visible objects in a geometric model is beyond the scope of this chapter. Assume that we have already obtained the visibility data from some visibility algorithms, we are more interested in how to store, organize, and manipulate the visibility data, so that visible objects (in respective representations) can be searched and rendered efficiently.

It is worth mentioning that the visibility algorithms are run in a precomputing

process for pre-determined viewing regions. Real-time visualization can only be performed in viewing regions which have been processed by the visibility precomputation.

In this chapter, we present a novel data structure called Hierarchical Degree-of-Visibility tree (HDoV-tree) which employes visibility techniques for virtual walkthrough. We propose a threshold-based traversal algorithm that allows the HDoV-tree to be tuned to balance visual fidelity and performance. We also propose three storage organizations of the HDoV-tree on secondary storage. We have implemented the proposed structure in a prototype walkthrough system called VISUAL, and conducted experiments to study its effectiveness. To further improve performance, we have designed two memory cache replacement policies for the tree nodes, and compare their performance against the conventional LRU method.

The rest of this chapter is organized as follows. We present the logical structure of the HDoV-tree in Section 5.2. The search method of the HDoV-tree is discussed in section 5.3. In Section 5.4, we present the storage schemes for the HDoV-tree. In Section 5.5, we discuss the method to compute the DoV values. In Section 5.6, we briefly introduce the technique to generate LODs for the HDoV-tree. And then in Section 5.7, we give the design of two cache replacement policies for the nodes of the HDoV-tree. Section 5.8 reports the results of our experimental study on our prototype system. We summarize this chapter in section 5.9.

5.1 Overview of Our Techniques

In visualization systems, one of the most frequently used operations is the *viewpoint* query, which returns all objects that are *visible* from the query viewpoint. By modeling the movement of a viewpoint, we will have a walkthrough application that continuously refreshes the set of visible objects as the viewpoint moves. To support these queries for large models, one straightforward solution is to partition the user viewpoint space into disjoint *cells*. For each cell, we associate a list of objects that are visible from any point within the cell. Thus, based on the cell corresponding to the viewpoint, only the visible objects need to be accessed. In practice, for performance reason, objects that are nearer to the viewpoint are shown in greater details while those that are further away may be approximated by their coarser representations. This minimizes I/O cost and the amount of data that the graphics engine need to render. However, there are some limitations with this simple strategy. First, the decision on the appropriate LODs to be used is ad-hoc and static, and cannot be changed at runtime. Second, the number of objects to be loaded may be unnecessarily high. This is because the LODs are for individual objects. For example, for a group of k distant objects, k LODs have to be loaded even though these objects can be treated as a single "entity" (i.e., a coarse representation obtained based on the group of objects). Finally, since the list is a single dimensional representation of the objects, there is no way to determine the spatial properties of these objects relative to one another without examining the entire list. This is important since it may be useful to load in the portion of data that the view frustum is (based on the viewing direction), followed

by those outside the view frustum.

We propose a novel data structure called *Hierarchical Degree-of-Visibility* tree (HDoV-tree) to support visibility queries. The HDoV-tree has the topology of a hierarchical spatial subdivision, and captures the geometric and material data as well as the visibility data in the nodes. Moreover, it is distinguished from spatial data structures such as R-tree in several ways. First, the HDoV-tree is *view-variant.* Given different viewing positions, the objects that a viewer can see may be different. In other words, at different viewpoint positions, the tree/nodes "capture" different visible objects. Second, traversing the HDoV-tree is based on the visibility data rather than spatial proximity. A branch along a path may be pruned based on some threshold value if the objects along the branch are hardly visible. Third, the HDoV-tree is tunable. Depending on the users' needs and the computational power of the machines, different users may see visible objects with different degree of fidelity.

5.2 The Logical Structure of HDoV-Tree

5.2.1 Degree of Visibility

When we talk about visibility, the set of visible objects is known as the *visible set* of the corresponding viewing point or viewing-region. The basic problem in visibility computation is to compute the visible set for a viewpoint or a viewing-region. The visible set can either be *precomputed* or *computed on-the-fly.* For precomputed visibility, the conservative from-region visibility is more common, as to store point

visibility for each viewpoint may be difficult.

There has been a lot of work done in recent years to precompute from-region visibility [20, 72, 17, 48, 92]. We have already discussed some of them in chapter 2.

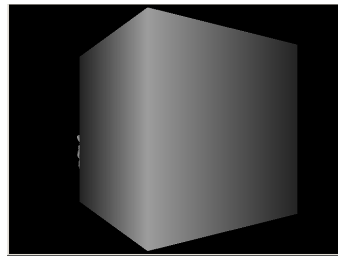
Existing visibility algorithms usually mark an object as *visible* or *invisible*. We observe that such boolean representation tends to be too 'conservative' as it will mark an object as visible even if only a very small part of it can be seen.

For example, as shown in figure 5.1, (a) depicts a fully-detailed bunny with only about 5% of its body visible to the viewer, while (c) shows a simplified model (500 faces) with the same portion visible. The difference of visual quality of (a) and (c) is almost ignorable. However, for (e) and (g), as large parts of the bunny are visible, the difference of the details is very obvious. Figure 5.1(b, d, f, and h) show the wire frames of the respective models. Conventional visibility apparently cannot tell the difference between (a) and (e), as both bunnies are "visible".

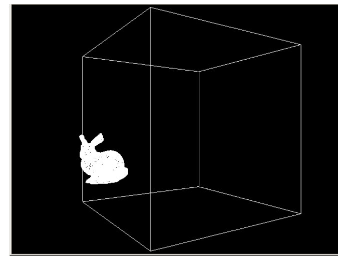
We introduce *Degree of Visibility* (DoV) to represent visibility more precisely. The DoV describes how visually important an object is, considering all possible viewing directions. More formally, we define the 3D *shadow set* of viewpoint p generated by an occluder $O \subset \mathcal{R}^3$ to be $S(p, O)$, which, in mathematical language, is the set of point s, whose interconnecting line with p, $\overline{sp}$, intersects O, while s is not in O [71]. Therefore, we have

$$S(p, O) = \{s | s \in \mathcal{R}^3, \overline{sp} \cap O \neq \phi \wedge s \notin O\}$$

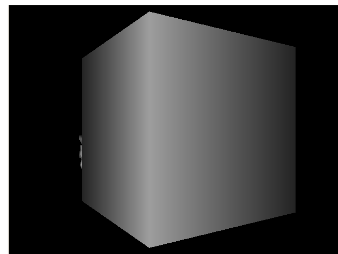
For a given viewpoint p, the visible part of a point set $X \subset \mathcal{R}^3$ can be defined



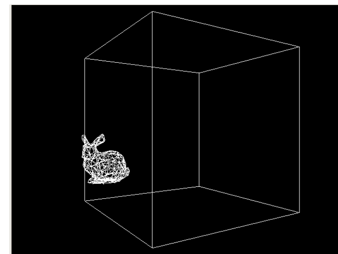
(a) 5% of the bunny(69451 faces) visible



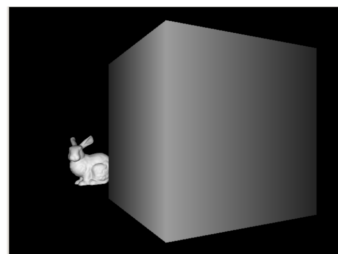
(b) 5% of the bunny(69451 faces) visible (wireframe)



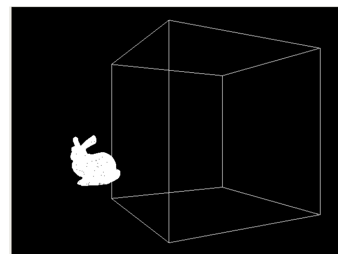
(c) 5% of the bunny(500 faces) visible



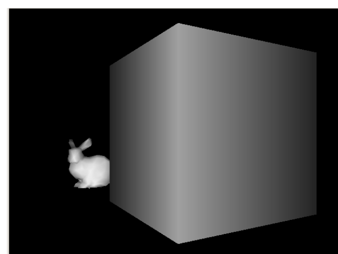
(d) 5% of the bunny(500 faces) visible (wireframe)



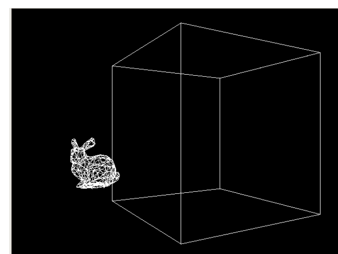
(e) 90% of the bunny(69451 faces) visible



(f) 90% of the bunny(69451 faces) visible (wireframe)



(g) 90% of the bunny(500 faces) visible



(h) 90% of the bunny(500 faces) visible (wireframe)

Figure 5.1: Comparison of Visual Quality For LODs at Different Degree-of-Visibility. Note the difference of bunny between (e) and (g), while (a) and (c) are almost identical.

as:

$$X_{visible} = X - \bigcup_i S(p, O_i) \tag{5.1}$$

We define the Degree of Visibility of a point set X with regard to a number of occluders O_i to be the ratio of the area of the projection of $X_{visible}$ onto a unit sphere (where $R = 1$) centered at p and the spherical area of the sphere. If we use $SProj_p(\xi)$ to denote the spherical projection of ξ on the unit sphere centered at p, the point DoV of set X can be defined as

$$DoV(p, X) = \frac{\int_{s \in Sphere} F_p(s, X_{visible}) \mathrm{d}A}{4\pi R^2}$$

where A is the spherical surface, and

$$F_p(s, \xi) = \begin{cases} 1 & s \in SProj_p(\xi) \\ 0 & \text{otherwise} \end{cases}$$

For region-based visibility, the DoV of an object viewed from a region $\mathbb{R}$ can be computed conservatively as

$$DoV(\mathbb{R}, X) = max(DoV(p, X)), \qquad \forall p \in \mathbb{R} \tag{5.2}$$

The definition of DoV guarantees that the DoV of an object is always in the interval [0, 1]. And it is also compatible with the traditional boolean visibility, as, if X is completely or partly visible regarding the occluder set, the DoV is greater than 0; otherwise, if it is completely hidden by the occluders, the DoV equals to 0.

We defer the description of the method of computing DoV to section 5.5.

5.2.2 HDoV-Tree Structure

We combine LOD, spatial index structure, and degree-of-visibility (DoV) into a Hierarchical Degree-of-Visibility (HDoV) tree structure. The backbone of the HDoV-tree is a spatial data structure that also stores the level-of-details (LODs) and degree-of-visibility (DoV) information. The spatial data structure essentially captures the spatial distribution of the objects in the virtual environment. However, there are several features that distinguish it from a spatial structure. First, the traversal of the HDoV-tree is based on the DoV values instead of the spatial content. Second, while the structure captures the static spatial distribution of objects, the visibility of these static objects is dynamic, i.e., object visibility depends on the positions of the viewpoints. In some sense, we can consider the HDoV-tree as a "template" that is dynamically instantiated with the visibility data of the corresponding cell of the viewpoint. Figure 5.2 illustrates this. Consider two viewpoints in cells i and j. Both the spatial content of the HDoV-trees are the same, but different sets of nodes may be visible.

In this chapter, for simplicity as well as because we are dealing with 3D objects only, we employ the R-tree [37] as the spatial structure in our implementation.

Figure 5.3 shows the *logical* structure of the HDoV-tree. By logical, we refer to an instance of the structure that corresponds to a particular cell. In the HDoV-tree, entries in the leaf nodes are of the form (VD, MBR, Ptr) where VD contains the DoV value of an object, MBR is the minimum bounding box of the object

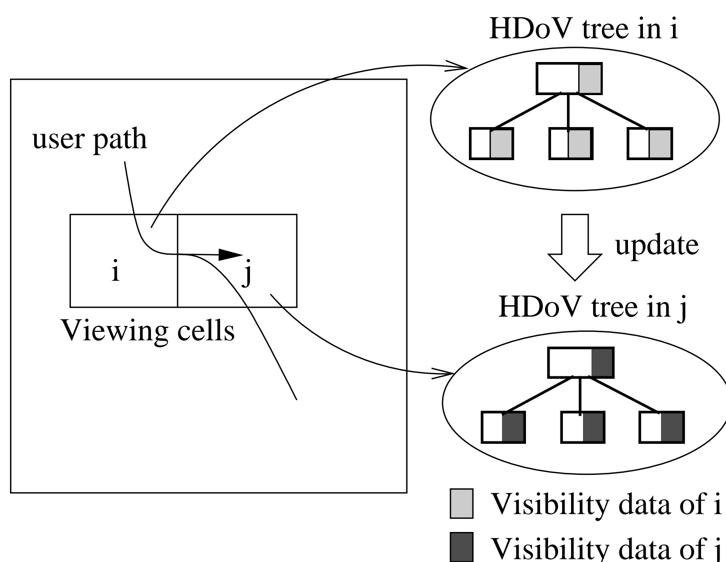


Figure 5.2: Dynamic update of the HDoV-tree.

and Ptr indicates the address of the object LODs. Each leaf node also contains internal LODs. These internal LODs are coarse representations of the aggregation of objects indexed by the node. Entries in internal nodes are also of the form (VD, MBR, Ptr). However, VD now contains the aggregated DoV value of the objects that MBR bounds, and Ptr points to the child node that leads to these objects. Each internal node also contains a pointer to levels of internal LODs that are even coarser representations of all objects bound by the node. A node is said to be *visible*, if any of its entries contains a DoV value greater than zero. The DoV field in the HDoV-tree has the following attributes:

1. The DoV in any entry is always greater than or equal to zero.

2. The DoV value of an entry E in an internal node equals to the summation of all the DoV values in the node that E points to. This is because the spherical projection of the visible part of a group of objects is always equal to the sum

of all the visible parts of each object in the group. This feature is useful for computing DoV values for entries of internal nodes.

3. If node N is visible, N must have at least one child node (or object) that is also visible.

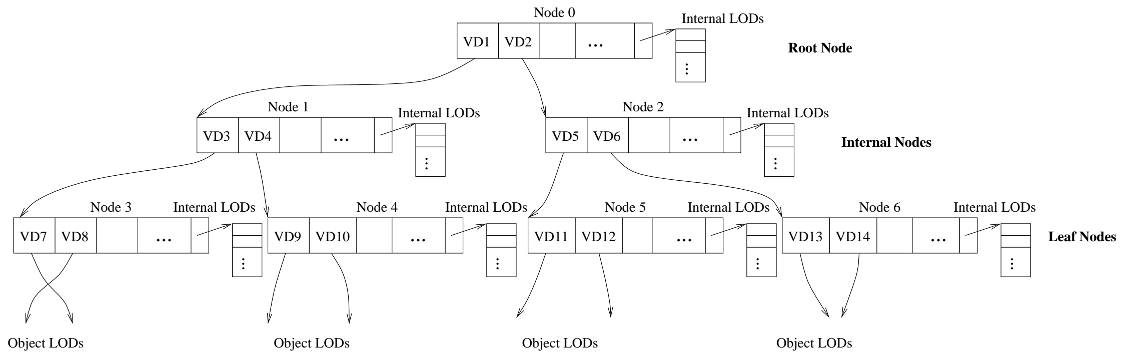


Figure 5.3: A Hierarchical Degree-of-Visibility Tree.

Since the DoV values stored in the tree depend on the viewing region that the viewer is in, the VD fields stored in each entry in the nodes are *view-variant*. In other words, for different viewpoints, the VD values are different. In contrast, the Ptr fields which determine the topology of the HDoV-tree, the internal LODs, and the object LODs are not dependent on the viewer, they are therefore *view-invariant*. Similarly, the MBR fields are view-invariant.

The main challenge in implementing the HDoV-tree is that the view-variant data change from one viewing cell to another. We shall defer the discussion of an efficient implementation of the HDoV-tree to Section 5.4.

Before leaving this section, let us look at the strengths of the HDoV-tree scheme, in particular, its advantages over the simple (cell, list-of-objects)-based method. First, a threshold DoV value, say η, can be used to balance the visual fidelity and

performance. The η value can be used to control the LODs to be fetched — objects with DoV values larger than η can be loaded in great detail, while those that are smaller can be represented by coarser LODs. For example, consider a large object that is very near the viewpoint but is barely visible because of obstruction from other large occluders. This object has very low DoV value. Under traditional method, the object is treated as visible (equivalent to DoV value of 1) and the detailed object model will be accessed. However, if its DoV is smaller than the threshold, a coarse representation may be loaded instead. Second, which follows the same logic as the first, we can potentially terminate the search at internal LODs if the aggregated DoV value of a node is small. Both of the above points translate to minimizing the amount of data to be loaded, and hence improving the performance of the system. By picking an optimal threshold value, the visual fidelity will not be compromised significantly.

Third, the spatial structure being used facilitates the design of a traversal algorithm that prioritizes the nodes to be searched. In other words, regions that are closer to the viewing direction can be traversed first, while regions that are outside the viewing direction can be delayed. This can further improve the response time significantly.

5.3 Search Algorithm

In a virtual environment, the main query type is the visibility query that asks for all objects (at their corresponding representations) that are visible from a query point q. More complex queries such as those involving the movement of a point in

walkthrough applications can be seen as a sequence of point queries, with optimizations to exploit temporal coherence. As such, in this section, we shall focus just on point visibility query, and discuss how walkthrough can be supported briefly.

Since the DoV defines the solid angle of the visible part of an object (or a group of objects), it is closely related to the screen projected area of the object. In real-time visualization, using different LODs to render objects is often very effective in improving rendering performance. In database query, using coarser LODs leads to fewer disk I/Os and higher retrieval performance. However, if the LODs are too coarse, the visual quality of real-time rendering will be unacceptable. Therefore, we must find a balance to trade-off the performance and the visual quality.

To realize this, we use a threshold DoV value η to determine what should be loaded in great details and what should not be. Essentially, objects (or object groups) with spherical projections (i.e., DoV) smaller than η can be retrieved with relatively low detail (internal LOD); otherwise, a finer LOD should be considered. Therefore, η controls the visual quality and performance while traversing the tree. For larger η values, the restriction on the visual quality will be looser, and lower details are allowed. As such, fewer disk I/Os are required to retrieve the results, and this leads to higher frame rate. On the contrary, for smaller η values, more detailed LODs will be loaded giving rise to better visual fidelity at the expense of lower frame rate. If the threshold equals to zero, only leaf-level LODs are allowed to be retrieved, and the system degrades to conventional list-of-objects based method.

Following the above discussion, it is clear that η determines the levels in which the traversal can terminate. When a traversal operation accesses a node entry,

if the DoV value is smaller than η, the traversal can terminate on this branch, otherwise, the traversal needs to proceed to the child nodes. By pruning branches with zero or small DoV values, disk I/Os can be saved.

Figure 5.4 shows the algorithmic description of the HDoV-tree search algorithm. Given a query point and an instantiated HDoV-tree, the traversal starts from the root node (line 1). In line 3, it looks for objects/nodes that are completely hidden. These are entries with DoV = 0. If so, it is obvious that the whole branch pointed to by the entry is completely invisible, therefore the recursion will terminate at this branch without adding anything to the query result, and the search continues with the next entry. If the DoV is greater than 0, then it is either a visible leaf node or a visible internal node. For the former, we include the object LODs into the answer set (lines 4-5). For an internal node, the traversal algorithm will decide whether to proceed to the child node based on the DoV value (line 7). If the DoV value of the entry is smaller than the threshold η, the branch under this entry is hardly visible, so we *may* want to retrieve a low-level internal LOD and terminate the recursion (line 8). We will discuss the second condition shortly. For entries with DoV values greater than η, we proceed to search their child nodes (line 10).

One issue which may arise with the above method is that the LOD of a node, which has small DoV, may contain more polygons than the sum of its visible descendants. To solve this problem, we can store the number of visible objects (NVO) in each VD entry. So VD has two view-variant fields

$$VD = (DoV, NVO)$$

Algorithm Search (Node)

1. For each entry E in Node
2. **Begin**
3. if(E.DoV equals to 0) return;
4. if(E is leaf)
5. E.ptr→LOD_{leaf} is added into result; /*equation 5.6*/
6. else
7. if(E.DoV $\leq \eta$ AND $h(1 + log_M s) < log_M(E.NVO)$)
8. E.ptr→$LOD_{internal}$ is added into result; /*equation 5.5*/
9. else
10. Invoke Search on E.ptr;
11. **End**;

Figure 5.4: The HDoV-tree traversal algorithm

Now we can apply a heuristic to determine whether to terminate the search at a node or to traverse down to the next level. This corresponds to the second condition in line 8 of the traversal algorithm (see figure 5.4). Suppose a node N has m leaf descendants, if the fan-out of the internal nodes is M, the subtree on N has an estimated height of $h = log_M m$. If there are n leaf nodes visible in the subtree, and these leaf nodes have equal DoV values, then the DoV of these leaf nodes is $\frac{DoV(N)}{n}$. Suppose that each visible object in the leaf nodes has f faces, and the ratio of the number of faces in the parent nodes over the sum of those in child nodes is s, or

$$s = \frac{nface(node)}{\sum_i nface(child_i)},$$

then the estimated number of faces in node N is $m \cdot f \cdot s^h$. On the other hand, the number of faces in the visible leaf nodes sum up to $f \cdot n$. So the condition to terminate the traversal is

$$m \cdot f \cdot s^h < f \cdot n \tag{5.3}$$

Substituting m into (5.3) gives

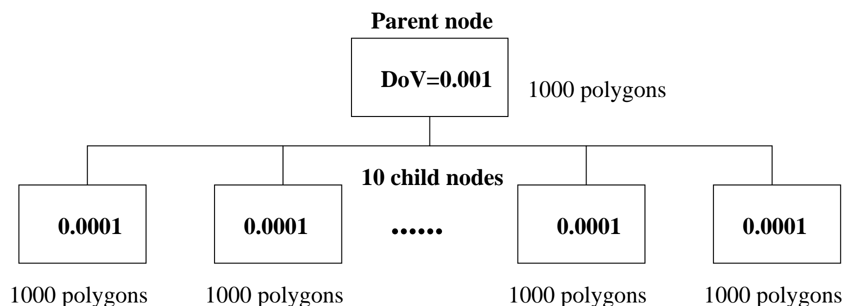
$$h(1 + log_M s) < log_M n \tag{5.4}$$

In (5.4), we observe that the larger the number of visible leaf nodes (n), the more likely the internal LOD will be used.

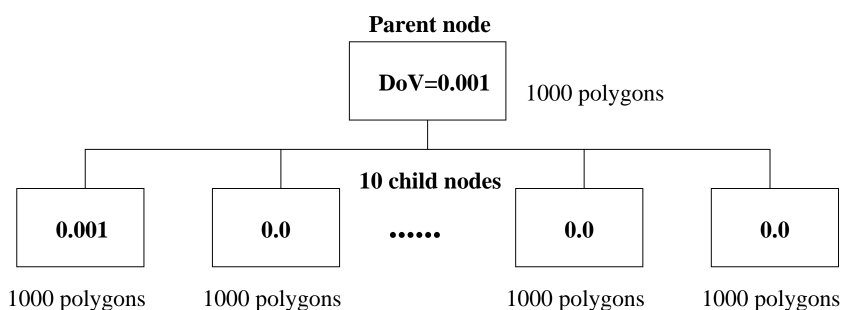
We give an example to demonstrate the issue. Figure 5.5 shows two simple scene trees. In figure 5.5a, the parent node has ten visible children, each of which having the DoV value of 0.0001. If we render the child nodes, it may require a total number of 10000 polygons to be drawn. However, if we use the internal LOD, only 1000 polygons are rendered. In this case, using internal LOD is more beneficial. However, in figure 5.5b, where only one of the children is visible, both the internal LOD and the leaf LOD have 1000 polygons. Since the leaf LOD is a finer representation of the visible child, we should therefore use the leaf LOD.

LOD of an active internal node can be selected as

$$LOD_{internal} = \frac{DoV}{\eta} LOD_{highest} + \left(1 - \frac{DoV}{\eta}\right) LOD_{lowest} \tag{5.5}$$



(a) A case where internal LOD is to be used



(b) A case where internal LOD is NOT to be used

Figure 5.5: Examples of when to use internal LOD in the HDoV-tree

where $0 < \frac{DoV}{\eta} \leq 1$. LOD of an active leaf node, meanwhile, can be selected as

$$LOD_{leaf} = k \cdot LOD_{highest} + (1 - k)\, LOD_{lowest} \tag{5.6}$$

where $k = min(\frac{DoV}{MAXDOV}, 1)$. Since the spherical projection of an object will not exceed 0.5 if the viewpoint is outside the bounding box of the object, we set $MAXDOV = 0.5$.

During a walkthrough session if the walkthrough path intersects the boundary of the current viewing cell, indicating that the user is "moving" out of it, the HDoV-tree is updated by the visibility data corresponding to the new cell. The

tree is subsequently queried using a pre-determined η value. Object LODs and internal LODs returned by the query are loaded into memory. To improve system performance, we can buffer some LODs in the main memory, and check if an LOD has already been loaded in previous queries. The η value can be used to control the walkthrough performance. Larger η value leads to shorter search time, and therefore smoother virtual walkthrough. While smaller η value leads to better visual quality.

5.4 Storage Schemes for HDoV-Tree

Recall that the HDoV-tree is essentially a view-variant structure: depending on the user's viewpoint, the visibility data of the tree may be different. There are essentially three ways in which the HDoV-tree can be implemented. First, we can associate each cell with a HDoV-tree, since all points within the cell have the same HDoV-tree. This method, while efficient, would incur substantial amount of storage. Moreover, such a scheme fails to exploit the view-invariant data of the HDoV-tree to minimize the storage overhead.

The second strategy is to construct a HDoV-tree on the view-invariant data. At runtime, the view-variant portion of the data is dynamically inserted into the tree. In other words, depending on the cell that a viewpoint is in, the tree is updated accordingly. This, however, requires the entire tree structure to be updated, and hence the performance penalty is very significant.

In this work, we advocate a third approach, which is to "store" the view-variant content into the HDoV-tree. In other words, the HDoV-tree captures the informa-

tion for all cells, so that if the viewpoint is in cell i, then the content of cell i is accessed. In this section, we present several storage schemes for this purpose. We shall use the notations in Table 5.1.

N_{obj}	Number of all leaf-level objects in the tree
N_{node}	Number of nodes in the tree
N_{vobj}	Number of visible objects in a viewing cell
N_{vnode}	Number of visible nodes in a viewing cell
$size_{integer}$	Size of an integer
$size_{pointer}$	Size of a pointer
$size_{vpage}$	Size of a Vpage
c	Number of cells

Table 5.1: Some notations.

5.4.1 The horizontal storage scheme

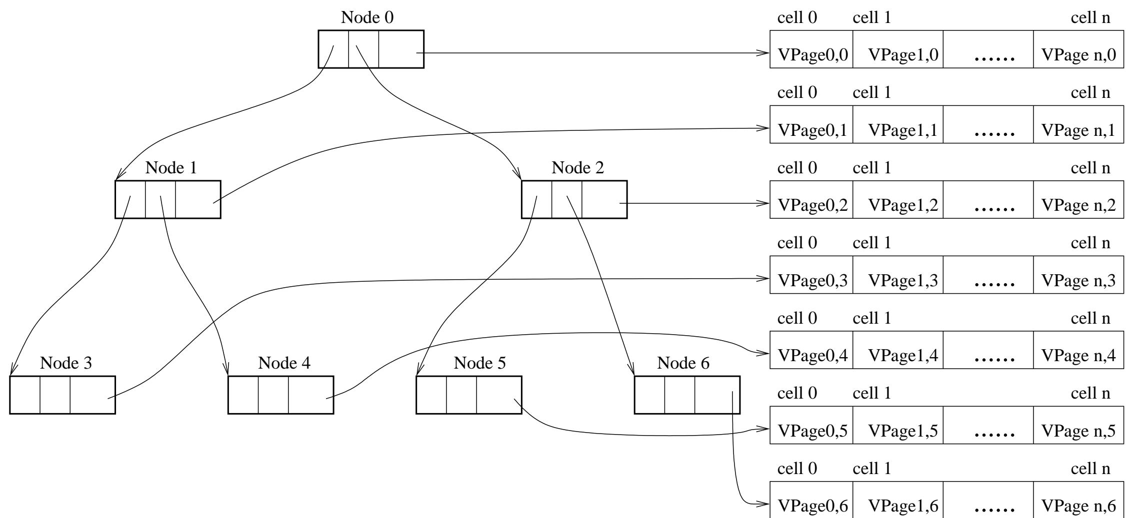


Figure 5.6: Horizontal Storage Scheme. *VPage i,j* represents the V-page of node j in cell i.

The most straightforward scheme is to store a pointer in each node pointing to a list of visibility data, which is indexed by the cell ID number. Figure 5.6 shows the data structure of the scheme, which we call a *horizontal scheme*. In this scheme,

the visibility data of a node N in cell C are stored in a fixed-size page, called the *V-page*. The V-page contains V-entries, one for each entry in a tree node, i.e., each MBR has a corresponding V-entry. The nth V-entry contains the visibility data of the nth entry in the corresponding tree node. In the horizontal scheme, internal nodes point to V-pages containing visibility data of the nodes, while leaf nodes point to V-pages containing object DoVs. A visibility query to a node costs one V-page access only. Unfortunately, the storage cost of the horizontal scheme is very expensive. As many of the nodes and objects in the tree are hidden when viewed from a cell, the horizontal scheme reserves the storage space of a V-page even if the node is not visible in the cell at all. More precisely, the storage cost of the horizontal scheme (excluding the tree structure as all the storage schemes have similar storage) can be estimated as

$$size_{vpage} \cdot c \cdot N_{node} \tag{5.7}$$

5.4.2 The vertical storage scheme

Another scheme, which requires lesser storage space, is the *vertical scheme*. As figure 5.7 shows, this scheme deploys an intermediate index structure, named *V-page-index*, between the nodes and the V-pages. The V-page-index is segmented by the cells so that each segment contains as many as N_{node} pointers. Each of the pointers, which are called *V-page pointers*, points to a V-page or to *nil*. Each node in the tree stores an offset starting from the beginning of the segment of the V-page-index. These offset values do not change by cells, therefore, they do not

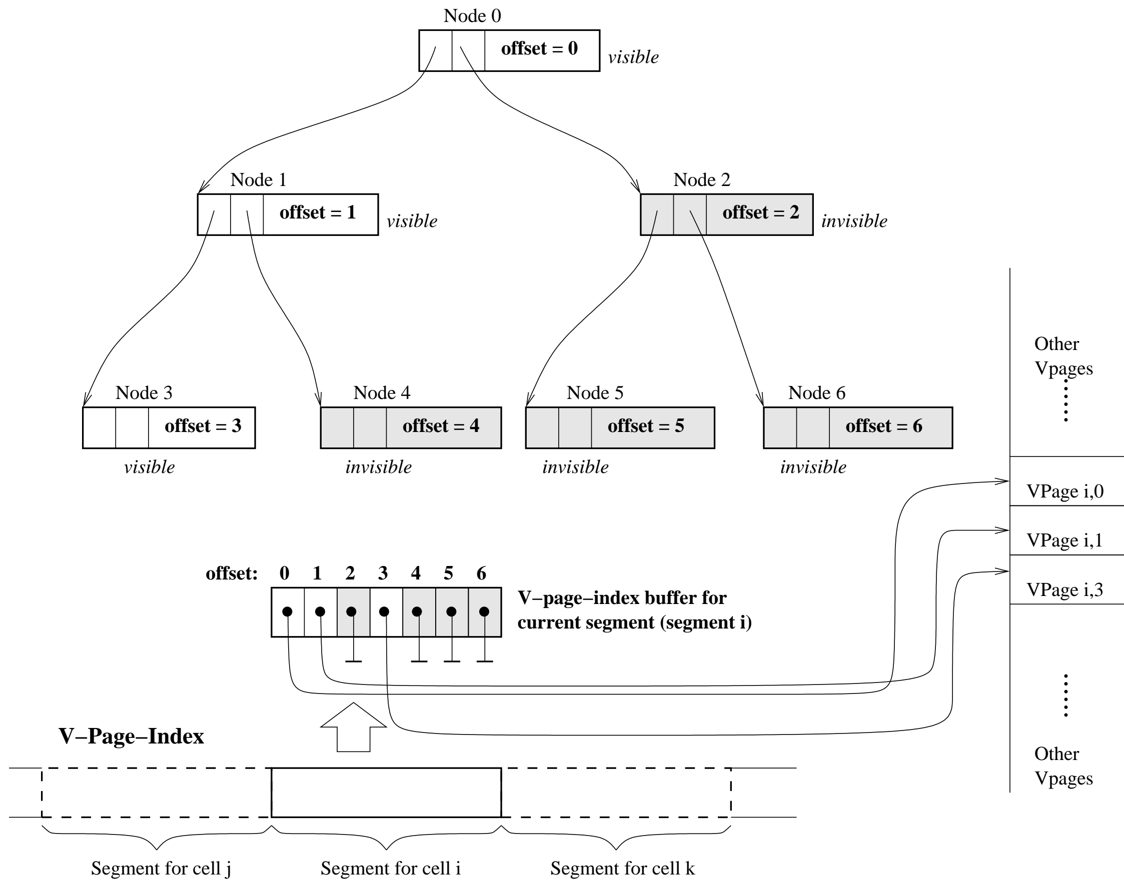


Figure 5.7: Vertical Storage Scheme.

require any update. When the visibility query traverses to node N, the V-page pointer in the V-page-index is found by using the offset value stored in N. If the V-page pointer is *nil*, it indicates that the branch is not visible in the current cell, so the branch below node N can be pruned; otherwise, the V-page of node N is retrieved from the V-page table. When the cell of the visibility query changes, the old segment of V-page-index is simply "flipped" to a new one by retrieving a new segment, which contains N_{node} pointers.

To expedite the V-page access, we also store the V-pages of the same cell together. The V-pages of a cell are sorted in the order of the tree nodes accessed in

the depth-first traversal, so that all V-pages accessed during a visibility query can be retrieved in a sequential scan.

If the average number of visible nodes in a cell is N_{vnode}, the total storage cost of the vertical scheme (excluding the tree structure) can be estimated as

$$size_{pointer} \cdot N_{node} \cdot c + size_{vpage} \cdot N_{vnode} \cdot c \tag{5.8}$$

While the vertical scheme uses offset in the tree structure and the horizontal scheme uses pointers, we assume that the size of an offset is the same as that of a pointer in the above expression. Since N_{vnode} is usually much smaller than N_{node} and the size of a pointer is also very small compared to a V-page, the storage taken by the vertical scheme is much smaller than the horizontal one. More importantly, N_{vnode} is not directly related to N_{node}, so the scalability of the vertical scheme is better than that of the horizontal scheme.

As the V-page-index keeps the current segment(cell) in memory, getting the pointer to the the V-page requires only memory access. If the visibility query changes the cell, the I/O cost of retrieving the new segment is $size_{pointer} \cdot N_{node}/size_{page}$ number of page accesses. The extra cost can be amortized over the visibility queries in the cell. However, the I/O cost to retrieve a new segment is $O(N_{node})$. If the number of nodes increases, "flipping" the V-page-index will be more expensive.

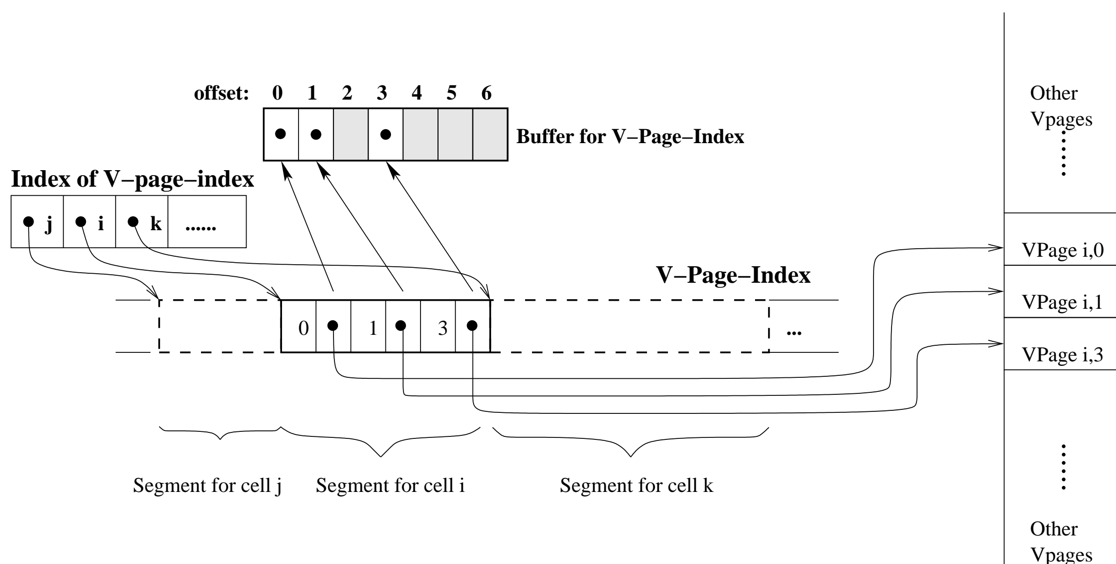


Figure 5.8: Indexed Vertical Storage Scheme.

5.4.3 The indexed-vertical storage scheme

As mentioned, the cost of "flipping" the V-page-index can be costly. To reduce the I/O cost during the segment flipping of V-page-index and the space of V-page-index, we can deploy another simple one-to-one index for the V-page-index file, as figure 5.8 shows (The tree nodes are omitted, as they are the same as those in figure 5.7). Only the offset numbers and the V-page pointers of the visible nodes are saved in the V-page-index file. As a result, only a visible node has a pointer stored in the V-page-index file, i.e. only non-*nil* pointers are stored in V-page-index. Therefore the size of the segments can be reduced dramatically. Flipping the V-page-index in this scheme requires only $O(N_{vnode})$ I/Os. Note that the segments stored in the V-page-index file can be of variable length. This scheme is named *indexed vertical scheme*. The I/O cost of each cell-flipping is only $N_{vnode} \cdot size_{pointer}$. The total

storage cost of this scheme is

$$(size_{pointer} + size_{integer}) \cdot N_{vnode} \cdot c + size_{vpage} \cdot N_{vnode} \cdot c \tag{5.9}$$

Before leaving this section, let us have a feel of N_{vnode}. Note that given an instance of the HDoV-tree (associated with a particular cell), some nodes are visible while others are not. If a node N is visible, N must have at least one visible child. So the number of visible nodes (objects) at a certain level in the tree is always larger than that at a higher level. Therefore, if there are N_{vobj} objects visible, there are at most N_{vobj} leaf nodes visible. Also there are at most N_{vobj} parent nodes of the leaf nodes visible, and so on. Hence, the total number of visible nodes in the tree is given by

$$N_{vnode} \leq N_{vobj} \cdot levels \leq N_{vobj} \cdot log_m N_{obj} \tag{5.10}$$

where m is the minimum number of entries for non-root nodes defined for the R-tree. Therefore, from expression 5.9, we can determine that the storage cost of the scheme is no more than

$$C \cdot N_{vobj} \cdot log N_{obj}$$

where C is a constant.

5.5 Computing DoV

In this section, we will discuss how to compute the degree-of-visibility for nodes in HDoV. As computing DoV at run-time could be computational expensive, our

system precomputes from-region visibility. The method can be divided into the following three steps for each viewing cell: (1) testing conventional visibility; (2) computing DoV for individual objects; (3) computing DoV for internal nodes. Step 1 is necessary because the subsequent steps are not conservative, and it is more efficient to prune completely hidden objects first. Based on the definition of the DoV, we propose an image-space approach to compute the DoV for step 2.

5.5.1 Testing Conventional Visibility

Many from-region visibility algorithms assume that the occluders are spatially near to the viewpoint, and are geometrically convex. Starting from the same assumptions, in our system, a simple yet effective visibility algorithm is applied. This algorithm, adapted from the method described in [71], assumes that there are large convex occluders near to the viewing region.

First, to expedite the spatial queries during the precomputation, the model set is organized in a spatial hierarchy, the R-tree structure. Given a 3D rectangular viewing region $\mathbb{R}$, the system performs a spatial query to find nearby convex objects with large size. A pre-defined 'belt' around the viewing region is used as a query region for selecting occluders. If the occluder set is too large or too small, the algorithm is able to adaptively change the 'thickness' of the 'belt' in order to obtain a reasonable number of occluders. The 2D version of the query region is plotted as the shadowed area in figure 5.9.

The objects indicated in the figure are selected as occluders. For each of the occluders, all the eight vertices of the viewing region are selected as the viewpoint

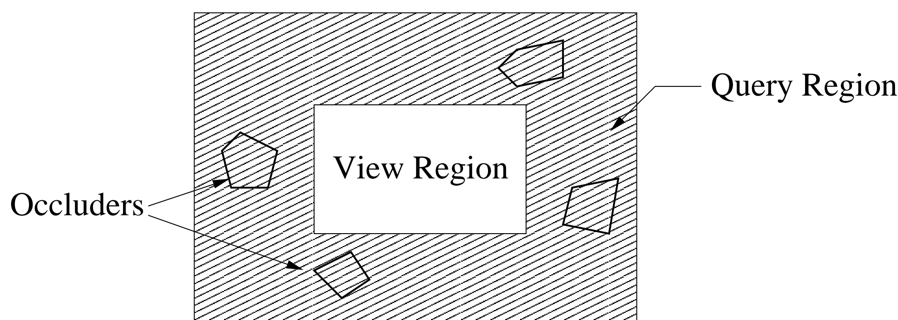
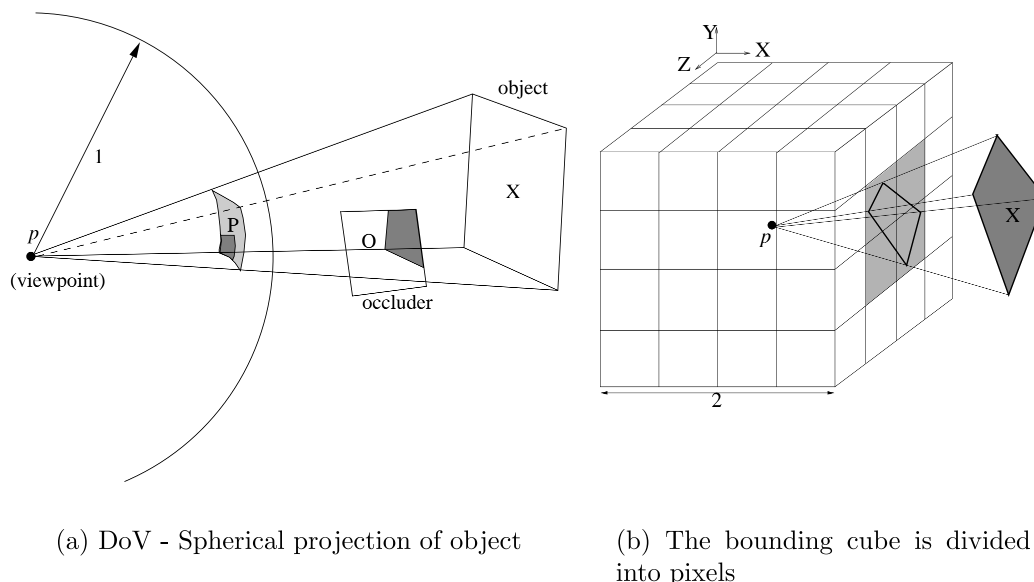


Figure 5.9: Querying occluders around viewing region

one by one and the occluder is rendered into the z-buffer after-wards. The hierarchy of bounding boxes in the tree is then tested against the z-buffer to find which nodes are hidden by the occluder. The hidden sets for eight vertices are intersected, generating the hidden set of the occluder. And finally, all the hidden sets of all occluders are union-ed to make up the global hidden set occluded by the occluder set. Testing against the z-buffer can be done with the assistance of standard OpenGL functions.

5.5.2 DoV of Individual Objects – An Image-Space Approach

According to the definition of the DoV, it can be approximately computed in the image space. As for the from-region DoV defined in equation 5.2, we compute the point DoV for all vertices of the viewing region, a rectangular cell, and choose the maximum value as its approximated from-region DoV. The point DoV can be computed in the image space with a *cubic projection* method. This method was inspired by the *hemicube method*, which was proposed to compute form factor in radiosity [15].



(a) DoV - Spherical projection of object

(b) The bounding cube is divided into pixels

Figure 5.10: Spherical projection vs. cubic projection

As figure 5.10(a) shows, given a unit sphere centered at viewpoint p and an object X, we need to compute the spherical area that X projects onto the sphere. The occluded area, or the dark shaded area in the figure must be excluded. In order to compute the area in image space, we set up a bounding cube around the sphere, as figure 5.10(b) shows. The faces of the cube are divided into equally small pixels.

The spherical projection of the object can be computed by testing the pixels of the Z-buffer. As figure 5.11 shows, for a very small area on the faces of the cube, ΔA, its DoV, which we call *delta DoV*, can be computed as

$$\Delta DoV = \frac{\Delta A}{4\pi(x_i^2 + z_i^2 + 1)^{3/2}} \tag{5.11}$$

Spherical projection of ΔA on the sphere can be computed as follows. For clarity we draw figure 5.12 in 2D. Firstly we compute the spherical projection of ΔA on

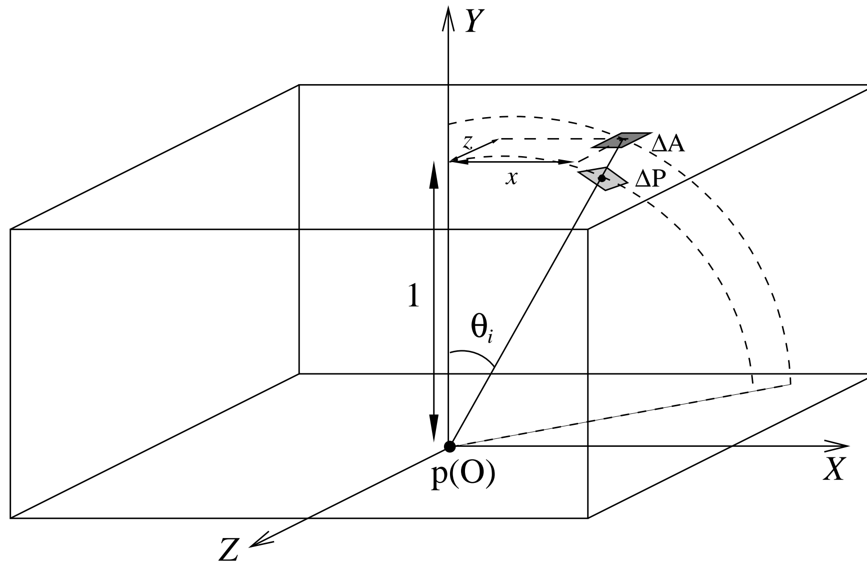


Figure 5.11: Computing spherical projection for ΔA

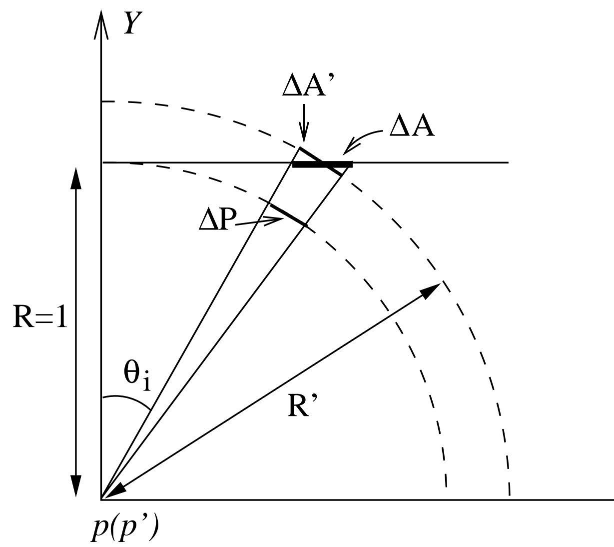


Figure 5.12: Computing the spherical projection of ΔA (2D).

another sphere, p', which is also centered at p and has a radius of $\sqrt{x_i^2 + z_i^2 + 1}$.

$$\Delta A' = \Delta A \cdot \cos\theta_i \ ,$$

where $\Delta A'$ is the effective spherical projection on sphere p', which has a radius of $R' = \sqrt{x_i^2 + z_i^2 + 1}$. Since

$$\frac{\Delta P}{\Delta A'} = \left(\frac{R}{R'}\right)^2,$$

we have

$$\begin{aligned} \Delta P &= \frac{\Delta A \cdot \cos\theta_i}{R'^2} \\ &= \Delta A \cdot \frac{1}{(x_i^2 + z_i^2 + 1)^{3/2}} \end{aligned} \tag{5.12}$$

Hence, we obtain

$$\Delta DoV = \frac{\Delta P}{4\pi R^2} = \frac{\Delta A}{4\pi(x_i^2 + z_i^2 + 1)^{3/2}}$$

We note the difference between ΔP and the delta form factor used to compute radiosity [15], as the latter is calculated as

$$\Delta F = \frac{\cos\phi_i \cos\phi_j}{\pi r^2} \Delta A.$$

This is because the lighting model considers the direction of source patch$_i$, while our projection does not.

The DoV of an object is therefore

$$DoV = \sum \Delta DoV = \sum_i \frac{\Delta A}{4\pi(x_i^2 + z_i^2 + 1)^{3/2}}$$

The occlusion issue can be easily dealt with by using an 'identity buffer'. Each object is pre-assigned a unique identity number, which is stored in the identity buffer if the object is visible in the conventional visibility test. After the projection of the visible objects onto the cube, we scan the pixels of the identity buffer and compute the DoV for each object. Since the projection is conducted in image space, the accuracy of the result may be affected by the granularity of the discretization. To guarantee both performance and accuracy, an appropriate pixel size needs to be used.

Performing the cubic projection for each of the vertices of the viewing cell could introduce some error, as we ignore the point DoV of all other viewpoints in the viewing region. For example, an object may be invisible for all the eight vertices of a rectangular cell, however, it may still be visible from some viewpoints inside the cell. For these objects, which are called *visible zero-DoV* objects, we can change their zero-DoV to a very small value, such as

$$DoV_s = \frac{\Delta A}{4\pi(x_i^2 + z_i^2 + 1)^{3/2}}$$

Therefore, we will still render these objects, but probably at low level of detail. If we select more sampling viewpoints inside the viewing cell to undertake cubic projection, the number of *visible zero-DoV* objects may be reduced, and the re-

sultant DoVs would be more accurate. But doing so would inevitably increase the processing time. In our experiments, we choose to use only the vertices of a viewing cell for cubic projection, and there is no noticeable problem in the visual quality.

5.5.3 DoV of Internal Nodes

Computing the DoV of internal nodes from the original model would be much more demanding than computing that of individual objects. Since the spherical projection of a group of objects(with occlusion) is equal to the aggregation of those of the individual objects, the degree-of-visibility information of the parent node can be computed from that of its child nodes by adding up all the DoVs of its children. As an example, given a number of nodes in the R-tree hierarchy, A, B_1, B_2, ..., and B_k, if B_i $(i = 1, 2, \ldots, k)$ are children of A, then we have

$$DoV(p, A) = \sum_{i=1}^{k} DoV(p, B_i)$$

The from-region DoV, $DoV(A)$, is the maximum value among the point DoVs at the vertices of the viewing region. If we use the image-space method in section 5.5.2 to compute the DoVs for the objects, the Z-buffer algorithm can guarantee that overlaps in the spherical projection among different objects will only be calculated for once. Adding up child DoVs is equivalent to adding up the spherical areas of separated regions on the sphere and to compute the ratio between the sum and the surface area of the sphere. And the effect of *visible zero-DoV* is trivial. So the result $DoV(A)$ will always be in interval [0, 1].

5.6 LODs in HDoV-Tree

LODs are usually generated using polygon simplification algorithms for complex objects. There are many polygon simplification algorithms in the literature, for example, the work in [73, 64, 40, 63, 28, 21], and so on.

We associate LODs with the higher-level nodes in the HDoV-tree, which we call *internal LODs*. Meanwhile, the LODs for individual objects are called *leaf LODs*. We apply Garland's vertex contraction algorithm [28], which is based on the quadric error metric. This algorithm can simplify a group of disjoint objects in very short time (a few seconds for the bunny model) while preserving high fidelity in its output.

The following are the steps we take to generate internal LODs for HDoV-tree:

1. We create the spatial hierarchy of the HDoV-tree. A three-dimensional R-tree is generated from the dataset in our work.

2. For each internal node, we aggregate the data of its descendants which can represent the corresponding portion of the scene, and apply Garland's simplification algorithm to generate different levels of detail, based on the node's position in the spatial hierarchy. The number of polygons that each LOD contains is chosen such that a parent node's finest LOD has fewer polygons than the sum of the coarsest ones of its children's.

There can be various methods to aggregate the geometries from the children of a node. Determining which method to use is implementation-dependent. The LODs of a parent node can be computed from the aggregation of the geometries of

its direct children or its leaf-level descendants. In our work, we choose to generate the internal LODs from the simplified models in the leaf nodes so that the results can preserve highest fidelity since there is less local error introduced during the simplification.

5.7 Caching The HDoV-tree Nodes

When the search threshold η is small, many of the nodes in the HDoV-tree need to be accessed. Since all the nodes in a HDoV-tree are disk resident, traversal of the tree can be further expedited by employing a cache (memory buffer) for the nodes. Since the nodes in the logical structure consist of the *view-invariant* and *view-variant* components, in a single-user environment the view-variant data buffered in memory may soon become invalid if the user's viewing cell changes. So it is more beneficial to cache the view-invariant parts (the spatial data) in this case, as they do not change by the viewpoint.

We now consider the problem of selecting the appropriate buffered node to be replaced, i.e. the cache replacement policy for the HDoV-tree. As much as we have used DoV to prune the search space during traversal of the HDoV-tree, we can also use it for cache replacement algorithm. Similar to chapter 4, the traditional method which we use for comparison is the Least-Recently-Used (LRU) policy.

The first algorithm that we look at is a DoV cache replacement policy, which replaces node entries according to their DoV values. The second one is a hybrid policy which combines LRU and DoV.

5.7.1 The DoV Cache Replacement Policy

The basic idea of the DoV cache replacement policy is that the visually important nodes may still be visible in the near future, so they may need to be kept in memory for future accesses.

With the DoV cache replacement policy, the system keeps a log on the degree-of-visibility for all the nodes cached in the memory buffer. All the entries, which are in the form of pair $(nodeID, nodePtr)$, are stored in a *hash map* [83], where the key is the $nodeID$. A map list structure, which contains pairs in the form of $(DoV, nodeID)$, is maintained in memory, and is sorted by the DoV values. Each time a request for a node comes in, we search the $(nodeID, nodePtr)$ hash map to determine if it is currently being cached. If it is, the node pointed to by the $nodePtr$ field is returned immediately; otherwise, the node is retrieved from the disk file and the map list structure must be updated. The candidate to be replaced is always the first entry in the $(DoV, nodeID)$ map list, since the first item always has the smallest DoV value. And the $(nodeID, nodePtr)$ hash map is subsequently updated according to the replacement.

The advantage of the DoV cache is that if the cache size is small, the DoV-based replacement policy helps to maintain the visually more important nodes in the memory buffer. As comparison, for the LRU cache, if the number of cache entries is smaller than that of the nodes accessed in a round-up of the recursive traversal, the cached nodes are very likely to be swapped (replaced) before they could ever be reused (hit) in subsequent queries.

5.7.2 The DoV-Time Cache Replacement Policy

The DoV cache replacement policy may be efficient in exploiting the *visual coherence* that exists between consecutive visibility queries. However, some of the nodes which have large DoV values may reside in the DoV cache for a long time without being accessed. This is possible for nodes that may be visible for earlier queries but not in subsequent ones. To avoid such "dead entries", we can combine the DoV metric and the time metric to have a *DoV-Time (DT)* replacement policy.

The basic idea of the DT replacement policy is that the DoV value stored in the cache can be adjusted by a function of the time. To be more specific, we define a function f to be

$$f(DoV, T) = DoV + \kappa \cdot T$$

where DoV is the DoV value of the node, κ is a predefined constant, and T is the latest time at which the node was accessed. The function f describes how the key values in the cache entries are to be updated as the time goes on. In the experiments, we set the κ value to 0.001.

5.8 Performance Results

5.8.1 Implementation and Experimental Setup

We have implemented the HDoV-tree as a component of a prototype visualization system, VISUAL. VISUAL is a virtual walkthrough system implemented on a Pentium 4 PC running RedHat 7.2 that also facilitates visibility queries on spe-

cific viewpoint. The dataset we use is a synthetic city model containing numerous buildings and bunny models. The raw datasets excluding the visibility data vary in sizes from 400 MB to 1.6 GB.

Generating the HDoV-tree requires a few steps. Firstly, an R-tree spatial index is created to organize the object models. The insertion algorithm applies a linear node splitting algorithm [3] to minimize the overlap of the bounding boxes. To generate internal LODs, descendants of each internal node are found. For leaf nodes, the internal LODs are generated by aggregating the object models and running a polygon simplification software, namely *qslim* [28]. Internal LODs of nodes at higher levels are then generated in a bottom-up order.

A conservative visibility algorithm is also applied on pre-determined cells to find visible objects in each cell. A hardware-accelerated DoV algorithm is then applied on the visible set to evaluate the DoV values of each individual object and node. DoV values of objects and nodes are then stored in *V-Page*s according to the various storage structures. The flow chart in figure 5.13 shows how the HDoV structure is generated step by step.

Figure 5.14 shows a screen shot of the 1.6 GB dataset which is used as the default dataset in all experiments. For this dataset, we generated the internal LODs and precomputed the DoVs for more than four thousand viewing cells. The precomputation takes about 1.02 seconds for each cell.

Three sets of experiments are conducted. (1) In the first set, we test the performance of the three cache replacement policies, which are noted as LRU, DoV, and DT respectively, in visibility queries. (2) In the second set, we evaluate the

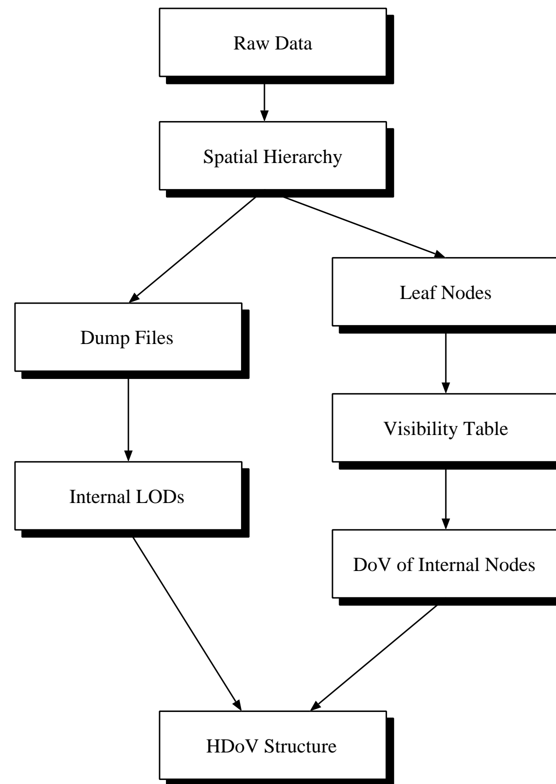


Figure 5.13: Steps to build a HDoV-tree.

performance of the HDoV-tree for visibility queries. Here, we test the scalability of the proposed search algorithm. We also compare the proposed search algorithm against a (cell, list-of-objects)-based method. We shall refer to the latter scheme as the naïve method. (3) In the third set, we study the real-time walkthrough performance of VISUAL. We use the REVIEW system (see Chapter 4) as our reference. Recall that REVIEW is a real-time walkthrough system that indexes objects using R-tree, and performs complement window queries in accessing the objects during a walkthrough session. As the original REVIEW system was implemented on SGI workstation, for fairness in comparison, we ported REVIEW to the PC platform. We shall denote the DoV threshold used in the HDoV-tree as η. As threshold values smaller than 0.008 generate very good visual fidelity, we shall use η values

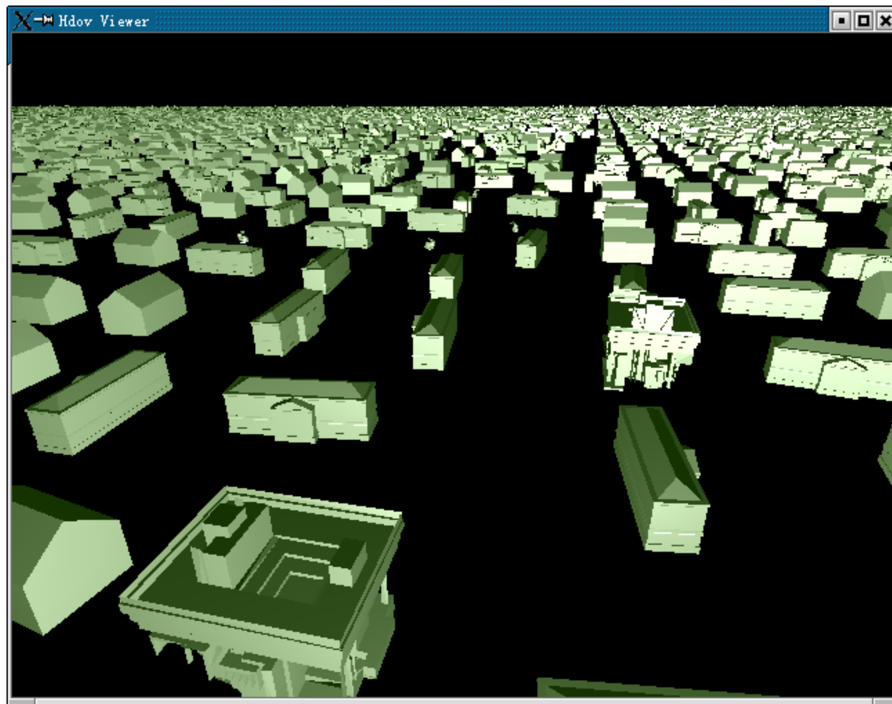


Figure 5.14: Bird's eye view of the default dataset

in [0, 0.008].

5.8.2 The storage cost of the storage schemes

We begin by considering the storage overhead of the HDoV-tree under the three storage schemes. Table 5.2 shows the result for the default dataset (The storage cost for the raw dataset is excluded since it is the same for all schemes). We note that the space taken by the horizontal scheme is very huge, as it stores V-pages for all the nodes in each cell. In fact, its storage cost is almost 20 times that of the other two schemes. The vertical scheme is more compact compared to the horizontal scheme. The indexed-vertical scheme is the most space efficient scheme as it stores both the V-page-index and the V-pages in a compact format. These data show that designing appropriate storage structure is very important for the

visibility query system.

Storage Scheme	Horizontal	Vertical	Indexed-vertical
Size	4 GB	267 MB	152.8 MB

Table 5.2: Storage space required by various schemes. Data in the table include V-page data and V-page-index structures only.

5.8.3 Experiment 1: On caching

We have implemented the three replacement policies for the cache of the view-invariant nodes in the HDoV-tree. They are named LRU, DoV, and DT respectively. The groups of experiments are conducted for the cache: (1) In the first group, we tested the cache performance for spatially continuous viewing cells; (2) and in the second group, we pick random viewpoints and test the cache performance for random visibility queries. The first case is very similar to the walkthrough environment, as the cells can be cascaded into a chain of the cells along the walkthrough path. We did not use a real walkthrough, simply because a real walkthrough session touches fewer cells.

One issue with the cache of the HDoV-tree is that the object data are heavy-weighted compared to the tree structure itself. Therefore, we note that the results of the caches show only the cost to traverse the HDoV-tree, and excludes the cost to retrieve the objects.

Results of Group 1 – Continuous Queries

Figure (5.15 – 5.17) show the cache hit rates in 4000 queries for the three cache replacement policies with various threshold values (η = 0.00001, 0.0001, 0.0005).

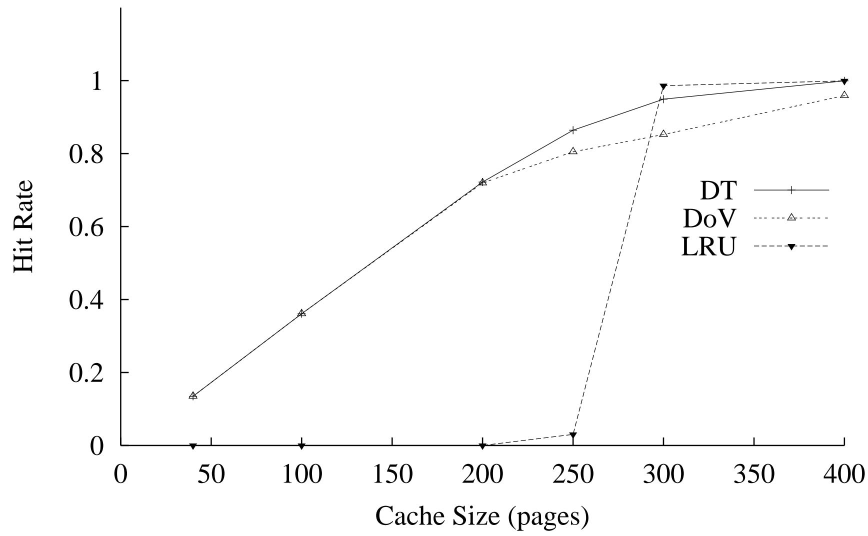


Figure 5.15: Cache Hit Rate For *Continuous Queries* Using $\eta = 0.00001$

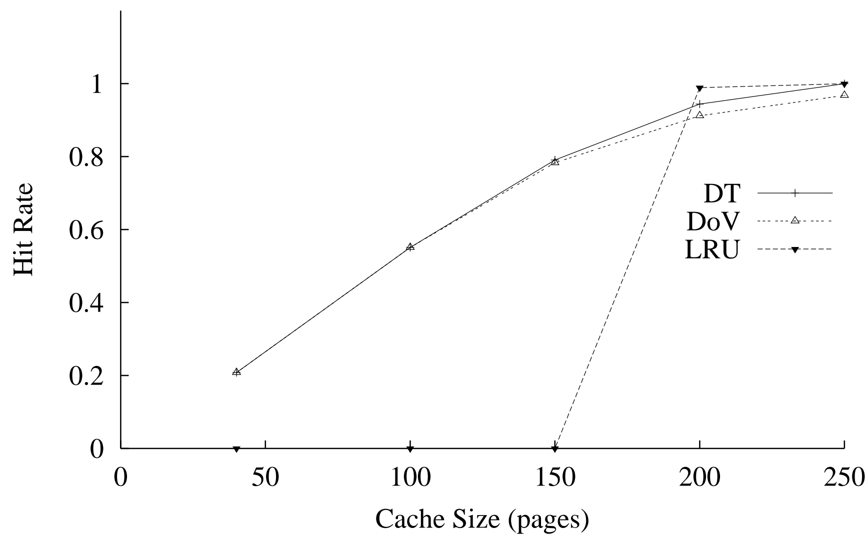


Figure 5.16: Cache Hit Rate For *Continuous Queries* Using $\eta = 0.0001$

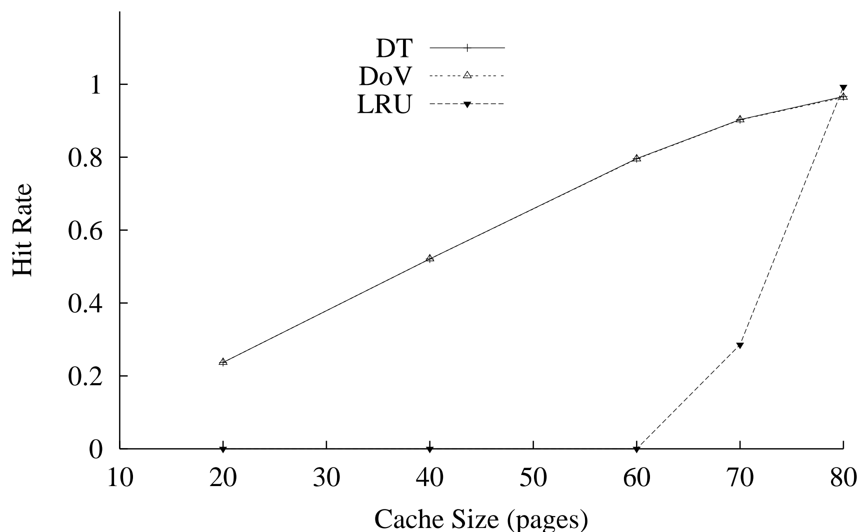


Figure 5.17: Cache Hit Rate For *Continuous Queries* Using $\eta = 0.0005$

These threshold values are small enough to make the traversal algorithm access a large number of nodes in the HDoV-tree. Since large threshold values may lead to too few node accesses, we only look at these small η values in our experiments. These three figures show that for different threshold values, as the cache size increases, the hit rates of all the three methods increase too. When the cache size is relatively small, the hit rate of the LRU algorithm is almost always equal to zero for different threshold values. This phenomenon is expected because if the cache size is smaller than the set of nodes to be accessed in a query, the nodes being buffered in memory are replaced before they can be reused in the next query (sequential flooding). In this case, the LRU cache replacement policy is very wasteful in buffering the nodes, as the cache cannot save disk accesses at all! The DoV and DT replacement policies perform better than the LRU under such circumstances. Also we note that the curves of the DoV and DT replacement policies are very close to each other when the cache size is small.

When the cache size gets larger, the hit rate of the LRU increases dramatically in all the figures, and is very close to 100%. The DT method (drawn in dashed lines in the figures) increases slightly faster than the DoV method, and has almost the same hit rate when cache size is large. Therefore, the DT method outperforms the DoV method in hit rate, and is comparable with the LRU when the cache size is relatively large. Hence, the DT cache replacement policy is a very promising scheme among the three in terms of hit rate. Note the maximum cache size used in our experiments is much smaller than the total size of the tree nodes. For example, when $\eta = 0.00001$, the maximum cache size is 400 pages, while the view-invariant part of the tree nodes occupy more than 7500 pages.

Results of Group 2 – Random Queries

Figure (5.18 – 5.20) show the experimental results of the cache hit rates for 4000 random visibility queries in the viewing regions. The threshold values are same as those that are used in group 1.

As expected, the hit rates of the random queries are slightly less than those in group 1. But they display similar patterns with regard to the cache size. The DT cache replacement policy also performs slightly better than the DoV policy. And like the results in group 1, the LRU policy also performs at either of its two extremes – being 0 or very close to 100%. When the cache size is relatively large, the curves of the DT policy is very close to those of the LRU.

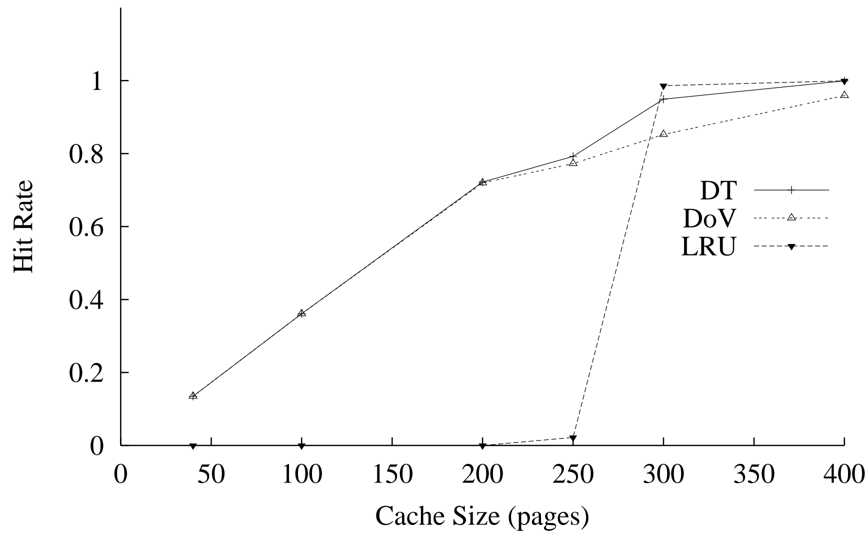


Figure 5.18: Cache Hit Rate For *Random Queries* Using $\eta = 0.00001$

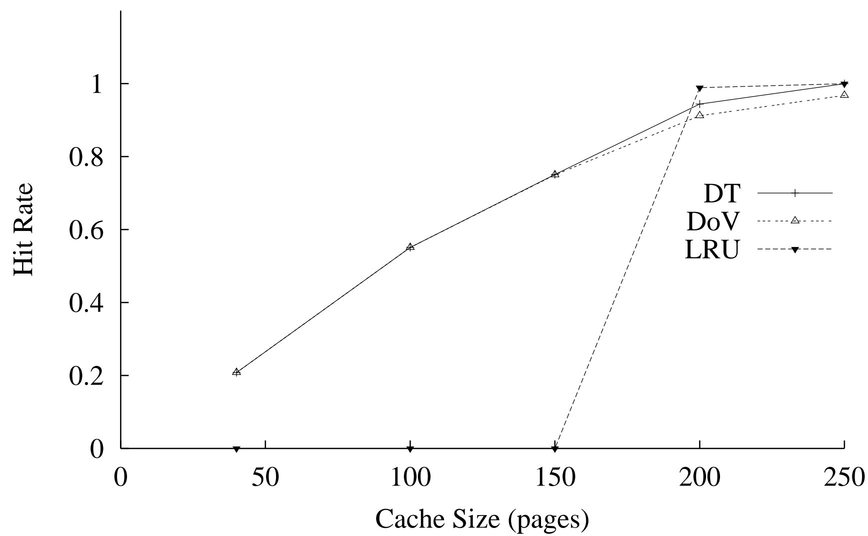


Figure 5.19: Cache Hit Rate For *Random Queries* Using $\eta = 0.0001$

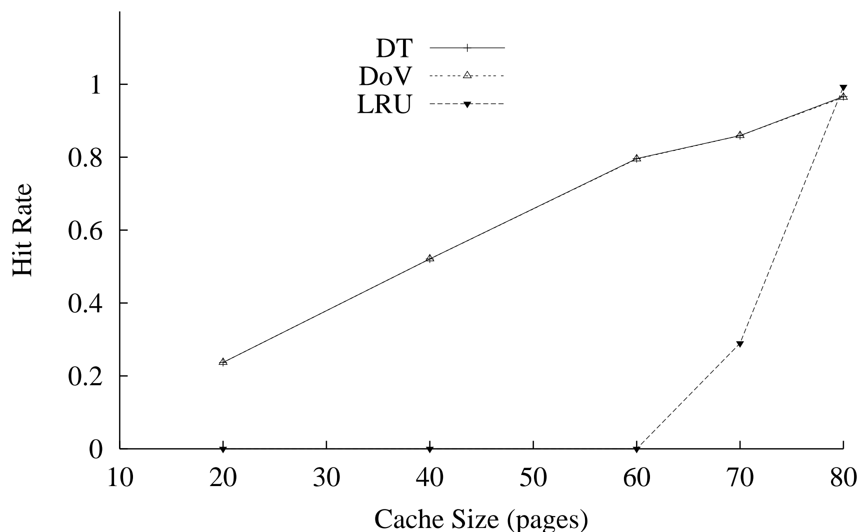


Figure 5.20: Cache Hit Rate For *Random Queries* Using $\eta = 0.0005$

5.8.4 Experiment 2: On visibility queries

In this experiment, we study the search performance of the HDoV-tree. We shall look at all the three storage schemes. We use the naïve (cell, list-of-objects)-based algorithm for comparison. In our implementation, this scheme accesses the V-pages of visible leaf nodes only. Moreover, all the models retrieved by the algorithm are from the object LODs. We note that the naïve method outperforms a spatial-query based method, as it accesses visible objects only. We shall defer the comparative study against a spatial-query based method to section 5.8.5.

We tested 10000 visibility queries at random viewpoint positions obtained from the precomputed cells.

Figure 5.21 shows the results of the search time as η (DoV threshold) varies from 0 to 0.008. From the figure, we observe that when η increases, the search time for all HDoV-tree-based schemes decrease significantly. This is expected as a large η value implies that the traversal will terminate more often at internal nodes. As a

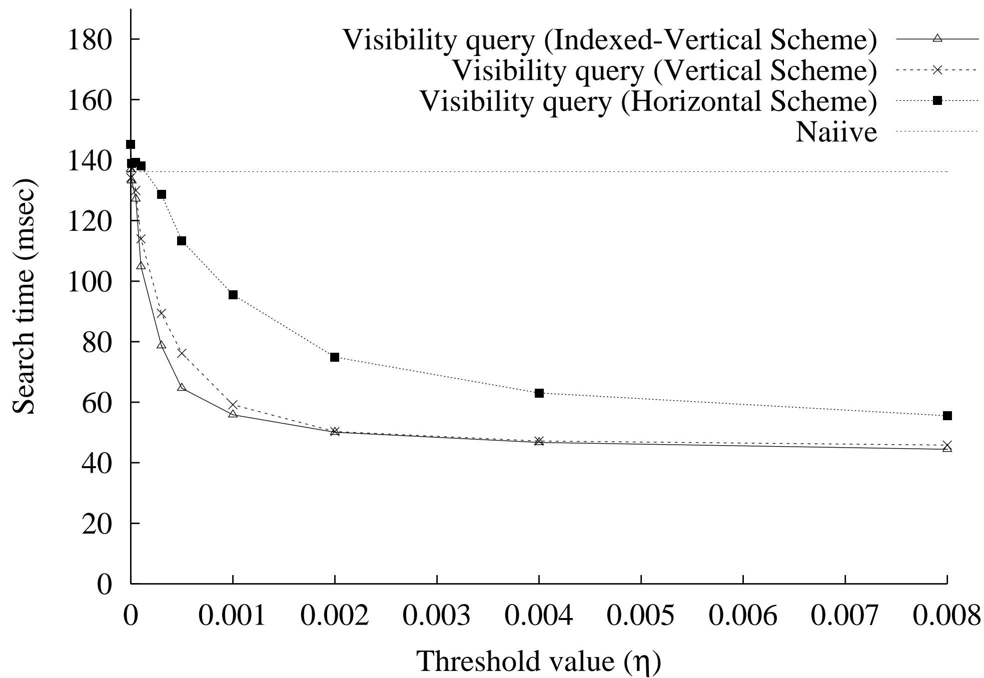


Figure 5.21: Search time with different η values

result, more internal LODs are allowed in the result set. Since the internal LODs have fewer details, the loading time of these objects is shorter. We also observe that the search performance for $\eta = 0$ is almost the same as that of the naïve method. This confirms our expectation that the HDoV-tree degenerates to a (cell, list-of-visibility)-based algorithm when $\eta = 0$.

Comparing the HDoV-tree-based schemes, we note that the performance of the vertical scheme and the indexed-vertical scheme is comparable. The indexed-vertical scheme is only marginally better than that of the vertical as it loads fewer data during the cell-flipping. The horizontal scheme performs the worst. This is expected as more disk seek is required in accessing the V-pages — all V-pages of a particular cell are not consecutively stored.

In view of the above results on the performance and storage cost of the HDoV-

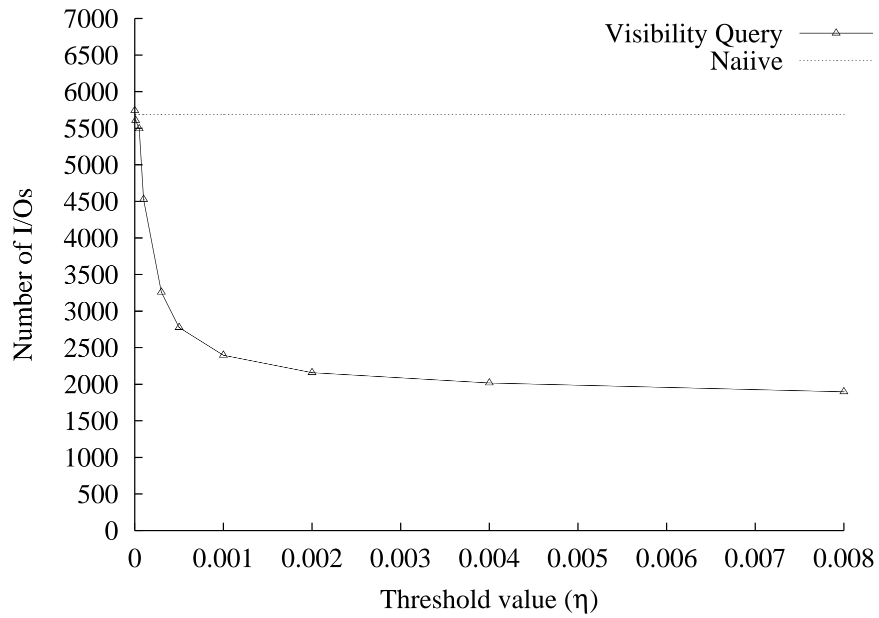
tree-based schemes, for the remaining experiments, we shall present the results for the indexed-vertical scheme only.

Figure 5.22(a) shows the number of disk I/Os as η varies from 0 to 0.008. Note that disk I/Os of tree nodes and V-pages, as well as the retrieval of the heavy-weighted model data are accounted in this figure. The results of the I/O cost of accessing the nodes and V-pages excluding the model data, or the *light-weight* I/O cost of the searches, are shown in figure 5.22(b).

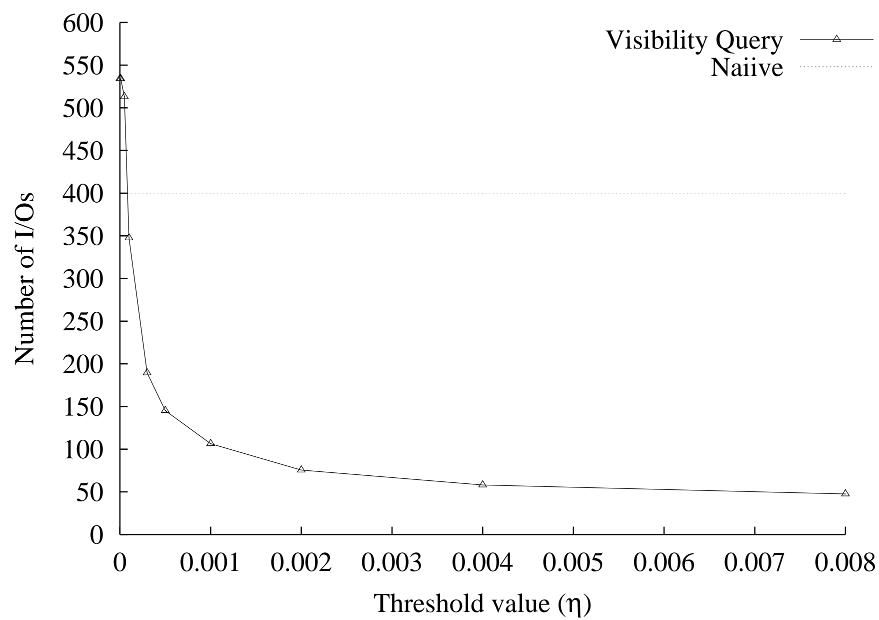
From figure 5.22(b) we found that for very small η values, the light-weighted I/O cost of HDoV-tree is higher than that of the naïve search. This is expected, as extra I/Os are needed to access the internal nodes and internal V-pages of the HDoV-tree. However, as η increases, the costs accessing the internal nodes are compensated by the much larger benefits obtained from being able to retrieve the internal LODs for nodes which have small DoVs. Therefore, the I/O costs of the HDoV-tree are always smaller than that of the naïve method.

To test the scalability of the search performance of the proposed HDoV-tree, we built a series of data sets ranging from 400 MB to 1.6 GB. In the precomputed cells, we chose 1000 random viewpoints as the experimental query set, and performed the same 1000 visibility queries on the datasets. The average search time and the average number of I/Os are shown in figure 5.23(a) and 5.23(b). We note that the results show only the cost to traverse the HDoV-tree, and excludes the cost to retrieve the objects (since all visible objects must be retrieved).

As the figure shows, the average response time and I/O cost increases only marginally with increasing dataset sizes. The I/O cost only increases in very small



(a) I/O cost of the visibility queries



(b) I/O cost of accessing the tree nodes and V-pages

Figure 5.22: Performance results on disk I/Os

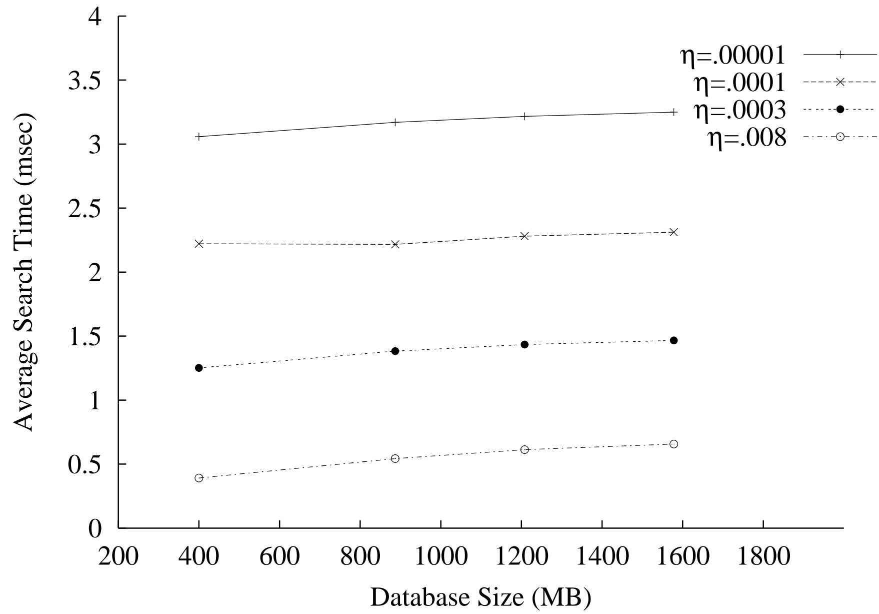
amount as the database size increases. The increase in search time is almost negligible. These results support our motivation for designing an efficient and scalable search algorithm and disk access method for querying visible objects.

5.8.5 Experiment 3: on interactive walkthrough

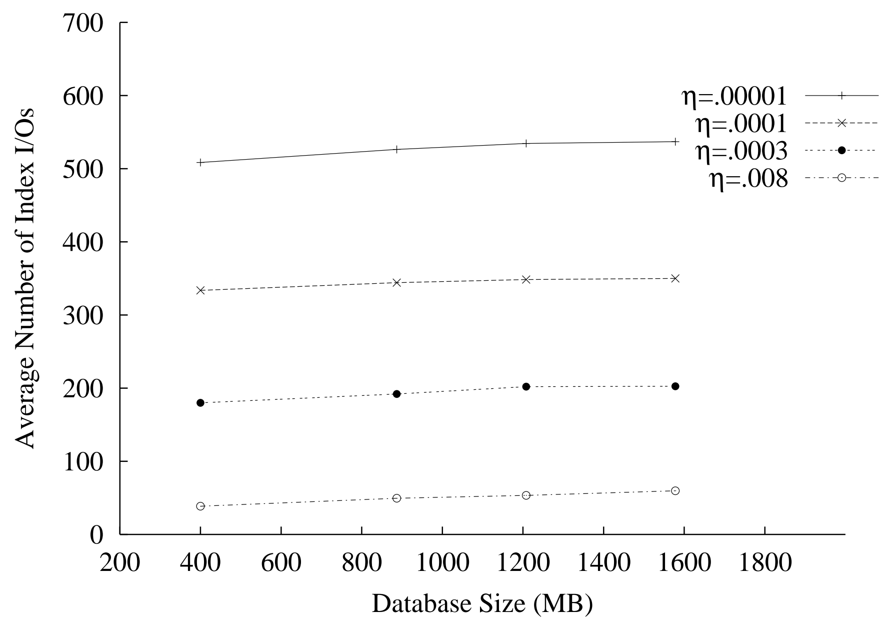
In this experiment, we evaluate VISUAL against REVIEW. For continuously moving viewpoint, there is often some spatio-temporal coherence to be exploited. For example, two neighboring cells often share a number of visible objects. For VISUAL, the search algorithm can be improved to a "delta" search algorithm which does not retrieve objects that have been retrieved in the previous queries. Objects that are retrieved in previous queries are buffered in memory and maintained in a hash set. Before the models are retrieved from disk, they are searched for in the hash map. A model is loaded from the database only if it is not buffered in the memory. As the models stored in the database are heavy-weighted, delta search algorithm can reduce the I/O cost significantly. For REVIEW, it also has its own "delta" search algorithm, known as the *complement search* algorithm (see Section 4.2.2 of Chapter 4).

The main metrics that we use for comparing the performance of interactive walkthrough are *average frame time* and *variance of frame time*. We recorded a few walkthrough sessions and played them back on the interactive walkthrough application. Each session is played back on both the VISUAL system and the REVIEW system.

In the REVIEW system, the size of the spatial-query boxes can be modified



(a) Average search time of datasets with different sizes



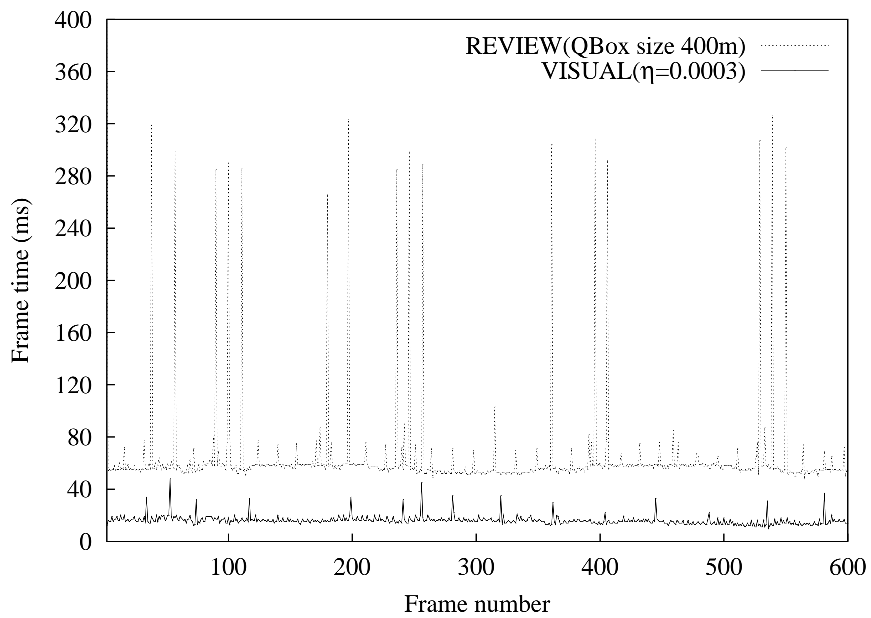
(b) I/O cost of datasets with different sizes

Figure 5.23: Scalability of the visibility query performance

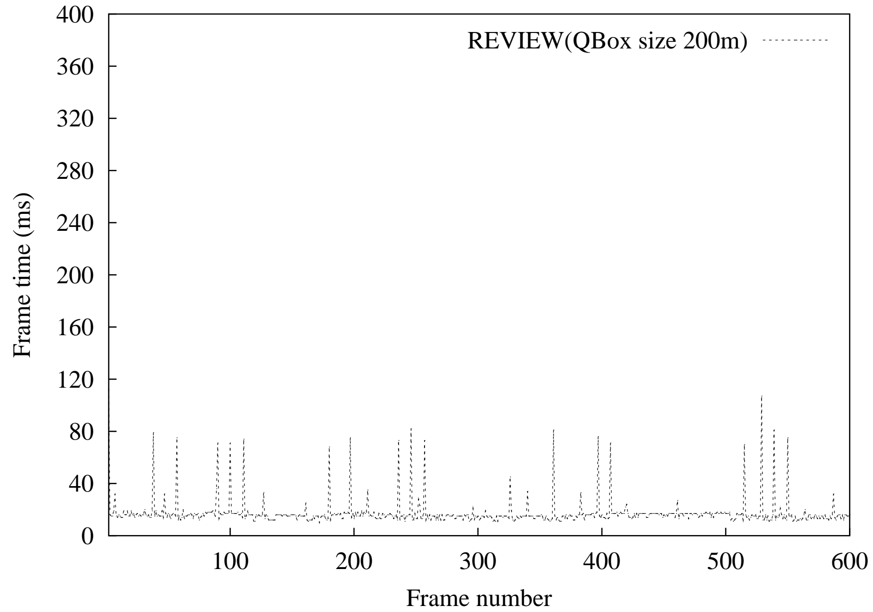
to update the system performance and visual fidelity. If the query boxes become larger, the system performs worse, but generates better visual fidelity, since more data are being retrieved. We shall demonstrate that when the spatial-query based system generates good visual fidelity, its performance is worse than the visibility querying system, and when the spatial-based system performs well, its visual quality suffers from "short-sightedness".

Figure 5.24(a) shows the comparison of time spent on each rendering frame between the spatial-query based REVIEW system and the VISUAL system. The size of the query box in the REVIEW system is set to 400m. The visual quality of the REVIEW system in this case is slightly worse, though comparable to the VISUAL system. However, the rendering frame time is very different as shown in figure 5.24(a). The REVIEW system is not only slower than VISUAL, but also "choppier", as the delay (marked by the spikes in the curves) caused by database queries are much longer. Therefore, the user of the VISUAL system can experience smoother walkthrough. The results shown in figure 5.24(a) confirms our discussion about one of the problems of the spatial methods, i.e., the spatial methods may retrieve invisible models, wasting the I/O resources.

If we reduce the size of the boxes of spatial-queries, the REVIEW system can produce faster frames, and the walkthrough can be smoother. For clarity, we plot the curve in figure 5.24(b). The average frame time is comparable to that of the VISUAL system in 5.24(a), although the spikes caused by spatial queries are still obviously taller. The visual fidelity of the REVIEW system in this case, however, is bad compared to VISUAL. This confirms that the spatial methods may loose some



(a) Comparison between VISUAL(η = 0.001) and REVIEW (large query boxes)



(b) Frame time of REVIEW (smaller query boxes)

Figure 5.24: Comparison of frame time

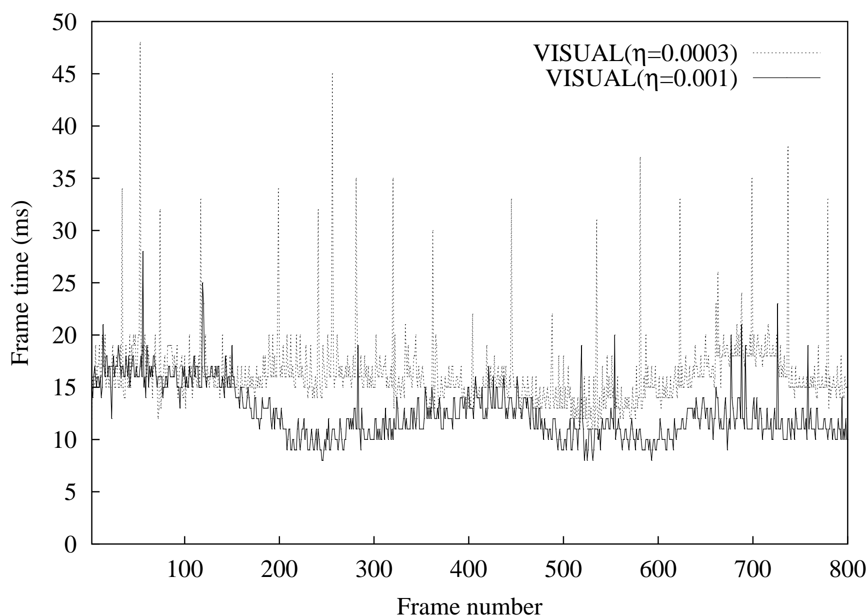


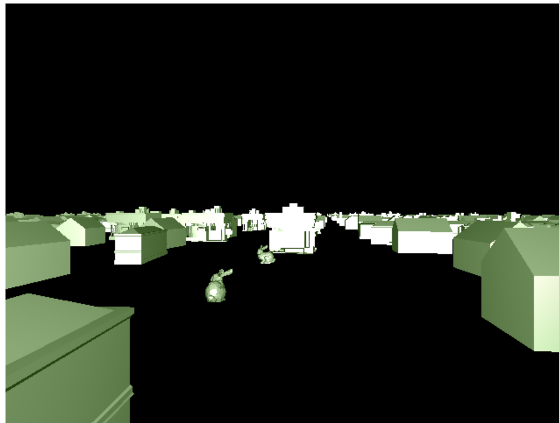
Figure 5.25: Comparison of frame time between VISUAL(η = 0.001) and VISUAL($\eta = 0.0003$)

visible models in the output. As a result, many objects which should be visible are not in the picture.

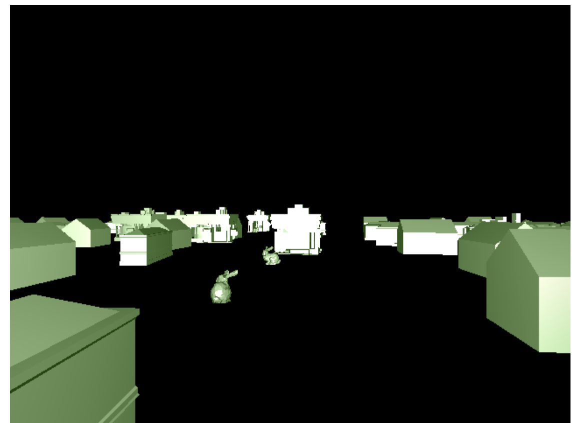
Figure 5.25 compares the results for VISUAL using two different threshold values: $\eta = 0.001$ and $\eta = 0.0003$. As shown, with $\eta = 0.001$, the frame rate can be up to 20% faster than that with $\eta = 0.0003$. This is expected since a larger η implies coarser representations are retrieved. However, as we shall see next, the visual fidelity is not much compromised.

Figure 5.26 illustrates an example on the visual fidelity of the two systems. Comparing figure 5.26(a) and (b), it is clear that REVIEW misses out on some objects. These are objects that are more than 200m away from the query box. Looking at figure 5.26(c) and (d), it is clear that VISUAL not only provides better visual fidelity than REVIEW, but the loss in visual fidelity is not obvious. Com-

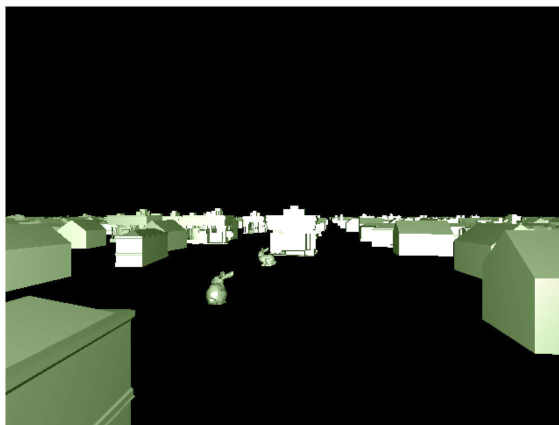
paring figure 5.26(c) and (d), we note that a threshold size of 0.001 can provide good visual fidelity comparable to that of 0.0003 much more efficiently.



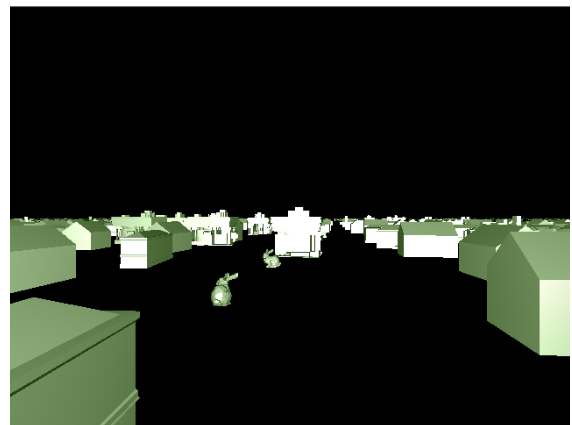
(a) Original models



(b) REVIEW (size of query boxes 200m)



(c) VISUAL ($\eta = 0.0003$)



(d) VISUAL ($\eta = 0.001$)

Figure 5.26: Comparison of Visual Fidelity. Far objects are lost in (b) due to the spatial method. The visual fidelity of VISUAL system is very good even if the threshold η is as large as 0.001

We recorded a few walkthrough sessions with different motion patterns. Session 1 is a normal walkthrough; session 2 turns left and right; and session 3 moves back and forward frequently. These sessions are played back for both the VISUAL system and the REVIEW system. Figure 5.27(a) shows the average search time

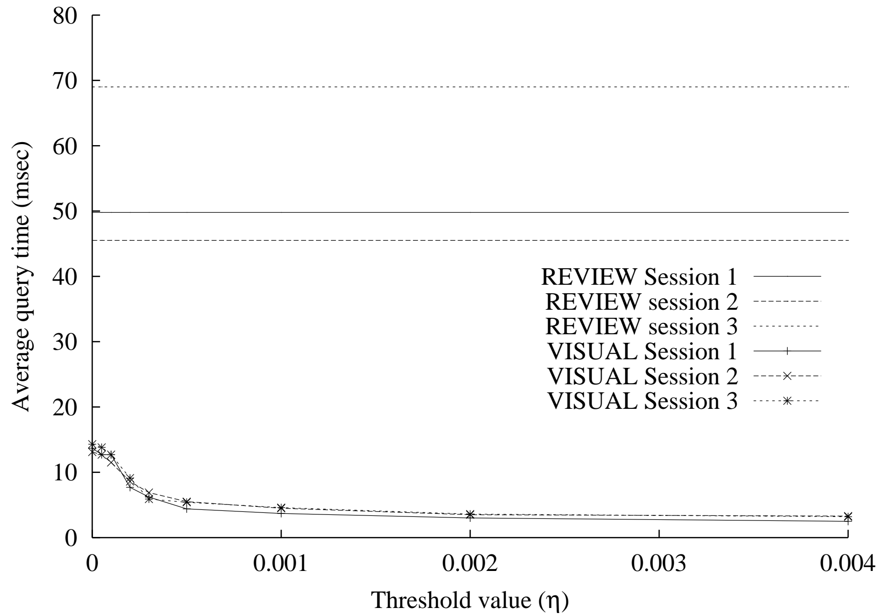
in each query for different walkthrough sessions. Figure 5.27(b) shows the average number of I/O operations in each walkthrough session. From these figures, it is clear that the queries in the VISUAL walkthrough are much faster than the spatial queries in the REVIEW system.

Table 5.3 shows the average frame time and the variance of frame time at different threshold values of session 1. Basically, as the threshold value increases, the average frame time decreases, due to shorter search time and coarser LODs being rendered. The variance of the frame time also decreases, therefore, the smoothness of the walkthrough also improves as the threshold increases. The average frame time of the REVIEW system with comparable visual fidelity (size of query boxes is 400m) is much longer than that of VISUAL. So is the variance of frame time. From this table, it is clear that the VISUAL based walkthrough performs smoother than the spatial access method.

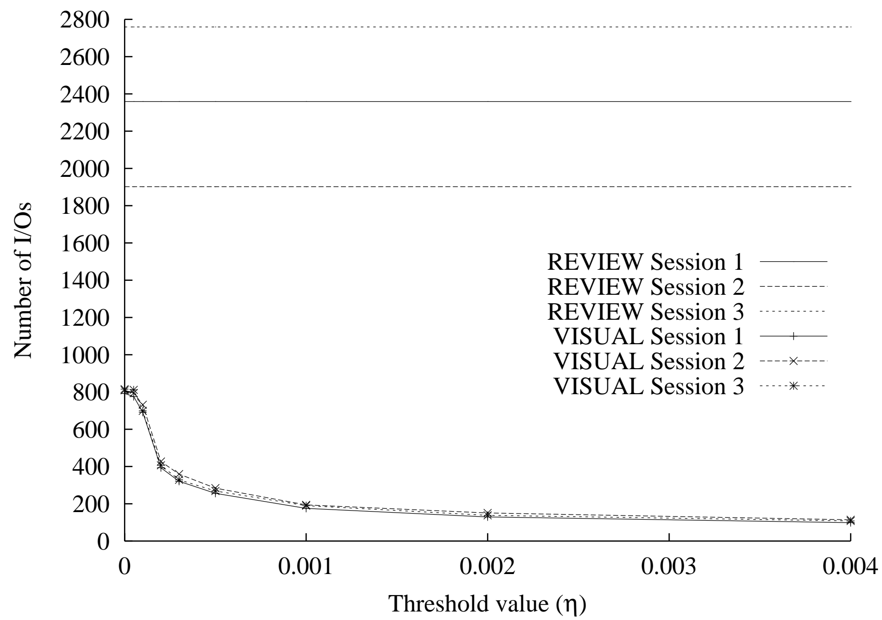
η	Avg Frame Time(ms)	Variance of Frame time
0	15.92	6.34
.00005	15.91	6.35
.0001	16.06	6.13
.0002	15.58	5.56
.0003	15.47	5.10
.0005	13.94	4.93
.001	12.78	4.35
.002	12.79	4.14
.004	12.65	4.15
REVIEW	57.84	16.46

Table 5.3: Results of frame time

In the experimental walkthrough sessions, the maximum memory used by the VISUAL system is 28MB, while the REVIEW system with a query box size of 400 meters requires 62MB. The memory requirement of the REVIEW system is



(a) Average query time of different sessions



(b) Average number of I/Os of different sessions

Figure 5.27: Search Performance in Different Walkthrough Sessions

closely related to the size of the query boxes. Since many invisible objects are retrieved by the spatial query, the spatial method requires more memory. The memory consumed by VISUAL is smaller, and is related to the search threshold. If the threshold becomes larger, more internal LODs are allowed in the query results, so less memory is consumed.

5.9 Summary

In this chapter, we have addressed the problem of optimizing performance and visual fidelity in visualization systems, especially in virtual walkthrough systems. We have proposed a novel structure called HDoV-tree that can be tuned to provide a trade-off between excellent visual fidelity and performance. The HDoV-tree is essentially an R-tree that contains visibility data and LODs. We also proposed three storage structures and two cache replacement policies for the HDoV-tree. We have implemented HDoV-tree in a prototype walkthrough system called VISUAL, and conducted extensive experiments to evaluate the performance of HDoV-tree. Our results show that HDoV-tree can provide excellent visual fidelity efficiently.

Chapter 6

Memory Based HDoV Scene Tree

In chapter 5, we proposed a novel data structure called the Hierarchical Degree-of-Visibility (HDoV) tree structure. The HDoV-tree captures the spatial, degree-of-visibility, and LOD information in a tree structure, and facilitates visibility query based on the hierarchical search algorithm.

Since a lot of existing models fit into memory, for performance reason we can consider the case when the whole data set fits into main memory. It is therefore natural to look at the search performance for HDoV-tree in memory.

Interestingly, the memory-resident HDoV-tree can be a scene tree by itself. As the whole tree is loaded into memory upon system initialization, the scene tree is also constructed. So there is no conversion cost during the real-time walkthrough and the rendering process is equivalent to the recursive traversal now.

In this chapter, we propose two traversal algorithms for the memory based HDoV scene tree. The first algorithm, called *Visibility Simplification Culling* (VSC) algorithm, is a memory version of the threshold-based traversal algorithm for the

HDoV-tree. The second one, named *Polygon Budget* (PB) traversal algorithm, aims at performance-guaranteed walkthrough for memory HDoV scene tree.

6.1 HDoV Scene Tree

Figure 6.1 shows the HDoV scene tree structure. Since the basic idea of the scene tree is very similar to that of the disk-resident HDoV-tree, we shall only highlight the structural differences. A node in memory-resident HDoV scene tree stores just one VD field, which contains the aggregated DoV value of all its children; while a node of the disk-resident HDoV-tree contains a number of VD entries. The reason we use multiple entries in disk-resident HDoV is that we need to put a disk-resident HDoV node into a disk page. Another difference is that the levels-of-details in memory based HDoV-tree are memory resident scene graph nodes that can be directly rendered. But in disk based HDoV-tree, the LODs are stored in files and need to be converted into renderer-friendly format. The other parts of the figure are self-explanatory, given the descriptions in chapter 5.

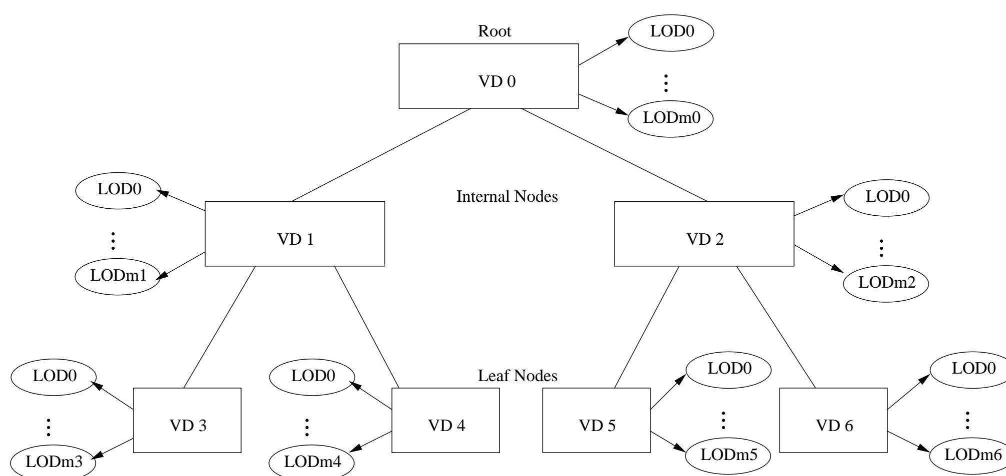


Figure 6.1: A Memory HDoV Scene Tree

6.1.1 The Basic Traversal Algorithm

We proposed a DoV threshold-based traversal algorithm for the disk-resident HDoV-tree in chapter 5. The basic traversal algorithm of a memory HDoV scene tree is similar to it. In order to differentiate it from the disk search algorithm, we use the term *Visibility Simplification Culling* (VSC) algorithm to refer to the threshold-based traversal algorithm for memory HDoV-tree. The VSC algorithm uses a designated DoV threshold value to control the recursive traversal. The condition to terminate the recursive traversal is also similar to that for disk-resident HDoV-tree. Selection of LODs of the active nodes follows the same rules. To avoid repetition, we do not present these points.

6.1.2 Polygon Budget Traversal Algorithm

The main purpose of the VSC algorithm is to control the maximum allowable screen areas which could be replaced by lower internal LODs, while achieving faster rendering speed. However, there is no performance guarantee for this algorithm. If the threshold specified by user is too small, the system may retrieve too many geometries, making the walkthrough perform poorly. In contrast, if the threshold is too large, the system may retrieve too many coarse LODs and therefore generate unacceptable visual quality.

To address this problem, we propose a new traversal algorithm which aims at performance-guaranteed walkthrough. This new algorithm is called *Polygon Budget* (PB) traversal algorithm. The PB mode restricts the maximum number of polygons, namely *polygon budget*, sent to the graphics engine based on the DoV

values and the position of the viewpoint. By setting different polygon budgets, the system can optimize the visual quality with guaranteed frame rates. If the budget becomes infinity, the system will degrade to conventional rendering strategy.

We will show how the DoV fields in the scene tree nodes can be used to allocate the polygons to appropriate nodes in the HDoV scene tree. By default, the traversal algorithm will apply a *view frustum culling* procedure to cull nodes whose bounding boxes do not intersect the current view frustum.

With HDoV structure, we can easily allocate the polygon budget, with an initial value of I, to nodes based on their DoV values. Starting from the root node, we refine the children with the DoV values in descending order. The polygon budget allocated to each child node is proportional to its DoV value and is propagated recursively top-down. Suppose node N, with DoV of D and budget of B, has m children, $c_1, \cdots, c_m$, with descending DoV values, $d_1 \geq \cdots \geq d_m$. The budget on each child is allocated as

$$B(c_i) = B \cdot \frac{d_i}{D} \cdot k,$$

where

$$k = \frac{\text{remain budget}}{I}.$$

The LOD of an active node is determined by its DoV and the current budget B_i as

$$LOD = min(\lambda \cdot LOD_{highest} + (1 - \lambda)LOD_{lowest}, B_i)$$

where

$$\lambda = min(\frac{DoV}{MAXDOV}, 1).$$

The remaining budget is calculated by deducting the polygon cost of the selected LOD. Before each refinement, we check if the remaining polygon budget will be less than zero if the operation was carried out. If so, the refinement is dropped, and attempt of selecting LOD is made on the parent node. Otherwise, refinement will proceed on other nodes with DoV values in descending order. If all the attempts fail, the recursive refining will terminate, and all the current active LODs are rendered.

The polygon budget strategy gives nodes with larger DoV values more privilege when allocating polygon budget. Objects which are visually more important are allocated more polygon budgets. It also sets relatively higher LODs for those with larger DoV values, therefore, it is more “visually optimized” as compared to conventional view-dependent rendering algorithms which sets LODs only based on the distance metric. The algorithm of refinement is listed in figure 6.2.

We note that the PB traversal algorithm tends to access more nodes than the threshold-based traversal algorithm, so its performance on disk-resident HDoV-tree may be no better than the DoV-threshold based traversal algorithm. Therefore, we propose it for the memory HDoV scene tree only.

6.2 Implementation

Our system has two distinct phases, known as the *precomputation phase* and the *run-time phase*. We used a complex city model, which contains many buildings and bunnies, as the dataset. There are a total of 50 bunnies, more than 100 complex building models and many simple buildings in the dataset, which includes more

```
Algorithm Refine (Node)
BEGIN
    LOD = ComputeLodBudget(DoV(Node), PolygonBudget);
    if(Node is leaf){
        if(LOD.NumPolygon < PolygonBudget){
            Node.SetLOD(LOD);
            PolygonBudget = PolygonBudget - LOD.NumPolygon;
        } else {
            Node.SetLOD(NULL); /* Drop the node */
        }
    }else{
        Success=TRUE;
        For (each Child from MaxDov to MinDoV) {
            if ( Refine(Child) < 0) {
                Success= FALSE;
            }
        }
        if (NOT Success) {
            if(LOD.NumPolygon < PolygonBudget){
                Node.SetLOD(LOD);
                PolygonBudget = PolygonBudget - LOD.NumPolygon;
                Attempts = 0;
                return TRUE;
            }else if(Attempts > MAX_ATTEMPTS){ /* Too many attempts */
                Node.SetLOD(LOD);
                goto END;
            }else{
                Node.SetLOD(NULL); /* Drop the node */
                Attempts = Attempts + 1;
                return FALSE;
            }
        } else
            return TRUE;
    }
END
```

Figure 6.2: The HDoV refine algorithm in Polygon Budget Mode

than 4 million triangles.

6.2.1 Precomputation

The precomputation phase was mainly implemented on a Pentium 4 PC with 1 GB memory and a GeForce 2/MX graphics card. A Sun workstation was also deployed to run batch scripts and the polygon simplification algorithm.

Firstly, we load the complete scene into memory on the PC to build the spatial hierarchy, which, in our case, is a k-D tree [6]. To generate internal LODs, descendants of each internal node are found, and for simplicity, dumped into individual files on the disk. Therefore, each internal node corresponds to a dumped file which includes all the geometries of the leaf objects. Meanwhile, the leaf nodes are saved in a separate file and can be used for DoV computation. The next steps can be performed in parallel on the Sun workstation and the PC. The DoVs are computed on the PC with the algorithm presented in section 5.5. The position of the viewpoint is restricted to a pre-defined view-cell hierarchy. For each viewing cell, which represents a potential viewing region, the DoV pre-computation is performed. After the conventional visibility information is obtained, the DoV values of the leaf nodes can be computed as section 5.5.2 describes. In parallel, the simplification program *qslim*, which applies Garland's quadric-based simplification algorithm [28] to polygon models, is run iteratively on the workstation to compute various levels of details for each internal node in the scene hierarchy, based on the node's position in the hierarchy. After the internal LODs are generated, the results are transferred to the PC and combined with the locally computed DoV values.

6.2.2 Run-time Visualization

The run-time visualization is conducted for viewpoints within the precomputed area only. At the beginning of a frame, the position of the viewpoint is obtained, and the viewing cell which contains the viewpoint is found. If the viewing region is the same as that of the previous frame, the HDoV scene tree will not be changed. Otherwise, if the viewing cell has changed from the previous frame, the visibility data, including the VD fields of the leaf nodes, of the new viewing region need to be retrieved. The DoV values of the internal nodes can be updated by summing up those of the children recursively from bottom up. Meanwhile, the number of visible leaf descendants in the nodes can also be updated. Besides the tree structure, we also store the addresses of the tree nodes in a hash table, so that the update of the VD fields can be performed quickly.

Rendering of the scene starts from the root node of the HDoV-tree, and can work in *visibility simplification culling mode* or *polygon budget mode*. The rendering methods of the scene tree have already been discussed in section 6.1.2 and in chapter 5.

6.3 Performance Results

The most time-consuming part in the precomputation is computing the DoV values of the leaf nodes for viewing cells (see figure 5.13). In our experiments, for each viewing cell, it takes about 1 second to compute the DoV with cubic projection, excluding the conventional visibility computation. Therefore, for a viewing grid

with a grid size of 64×64, the cubic projection approach (256 × 256 pixels) takes slightly more than 1 hour. Generating the internal LODs and leaf LODs for the HDoV nodes on the Sun workstation usually takes shorter time, and can run in parallel with the DoV pre-computation.

We compare the performance results of our proposed algorithms to a conventional rendering algorithm that deploys a hierarchical view frustum culling algorithm. The metrics we use are average frame time and number of polygons being rendered. For fairness in comparison, the conventional system, which is referred to as "*CONV*", uses the same spatial subdivision as the HDoV structure does. We also use the following notations for simplicity:

$VSC(\eta)$ The results of visibility simplification culling mode running with a DoV threshold of η.

$PB(n)$ The results of polygon budget rendering mode running with a polygon budget of n.

Figure 6.3 shows a bird's-eye view of the nodes being rendered in VSC mode in the HDoV-tree. We compare the results of VSC rendering mode with different η values to *CONV*. The DoV value and the number of visible leaf descendants determine whether the traversal will proceed to go down or just terminate at the current internal node, which, in the latter case, is known as an *active internal node*. We run the same walkthrough session iteratively under a few η values. The average frame time and number of polygons rendered, as well as the average number of nodes accessed in each traversal are listed in table 6.1. As the table shows, the average frame time of VSC is much smaller than that of CONV. The

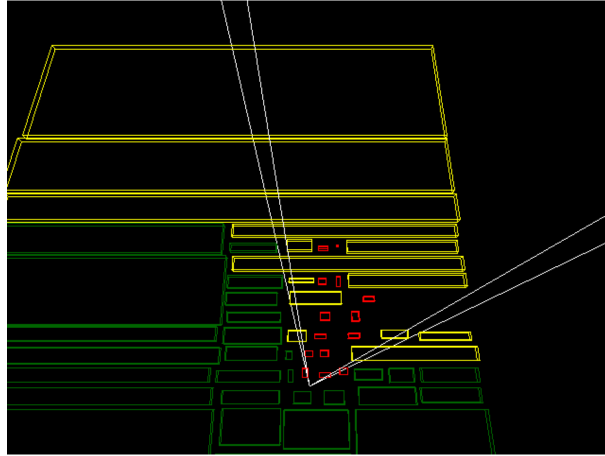


Figure 6.3: Top view of nodes being rendered in VSC mode. The white lines show the position of the view frustum. The yellow boxes bound internal nodes where the traversal terminates; the red boxes bound leaf nodes (individual objects); the green boxes bound nodes culled by VSC culling or view frustum culling.

polygon numbers rendered and the nodes accessed are also much smaller compared to CONV. When the η value increases, the frame time becomes shorter, and vice versa. This is consistent with our goal in designing the VSC algorithm.

Figure 6.4 shows the number of polygons rendered in each frame. The VSC mode greatly reduces the number of polygons being rendered. This is due to the active internal LODs and reduced number of nodes accessed.

Figure 6.5 shows the rendering time of each frame for various modes. The rendering time of the VSC mode is much smaller than that of the CONV. And the curves in rendering time display patterns similar to figure 6.4.

Figure 6.6 shows the bird's eye view of a snapshot (in both polygon and wire-frame modes) of the walkthrough. Note the active internal nodes rendered at low details in 6.6(a), and also rendered in yellow wire-boxes in 6.6(b). We also observe that when the threshold becomes larger, the visual quality will degrade, as more high-level nodes will become active. A threshold below 0.01 can achieve fairly good

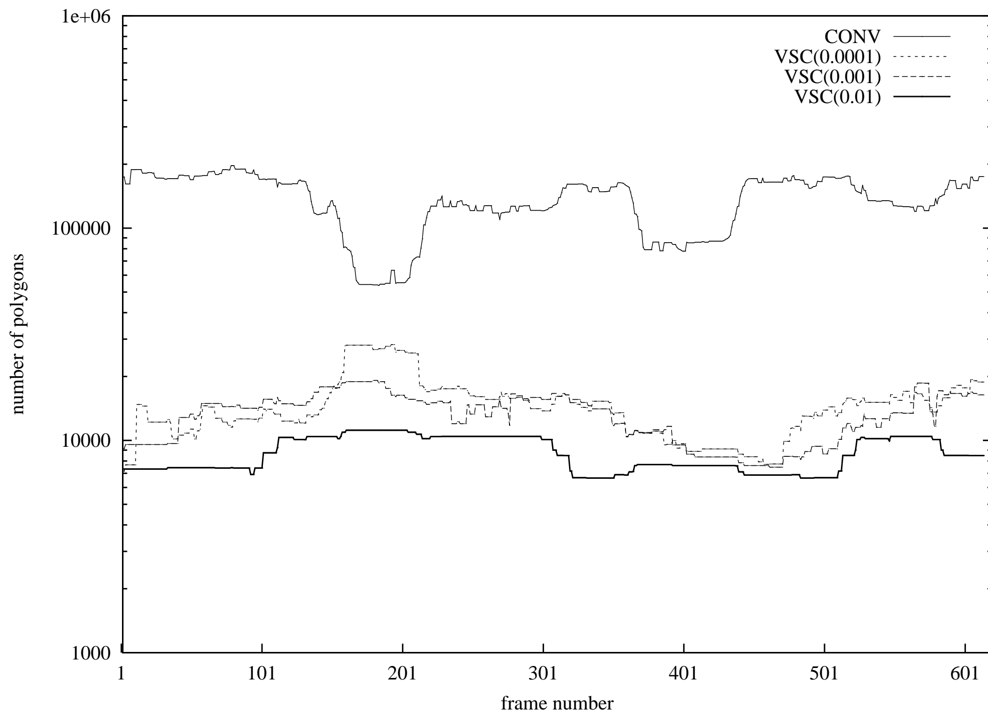


Figure 6.4: Number of polygons rendered in each frame (VSC mode vs. CONV mode). Note the *logarithm y-coordinate*.

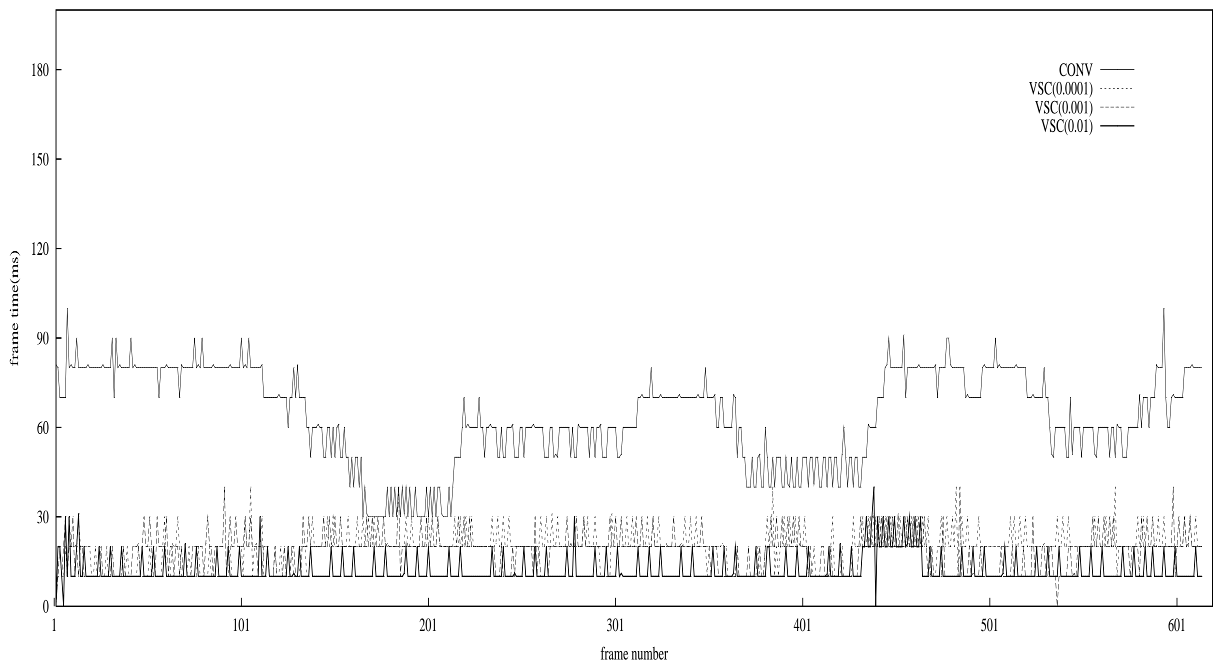


Figure 6.5: Frame time of the same walkthrough (VSC mode vs. CONV mode).

visual quality while keeping high frame rates. For the VSC(0.001), the system can gain a speedup of 3.5 times as compared to the CONV.

We captured a few video clips from the interactive virtual walkthrough using the memory HDoV scene tree to study the performance of the proposed algorithms. In the first video file (available in *http://www.comp.nus.edu.sg/˜shoulida/hdov.html*), the same walkthrough is conducted for the VSC mode with DoV threshold $\eta = 0.001$, and the CONV mode. With the same number of frames, the VSC walkthrough is much faster than the CONV. This is consistent with the experimental results shown in figure 6.5. Our experiment also showed that when $\eta = 0$, the system performance is very close to CONV. This confirms our prediction in section 5.3 that if $\eta = 0$ the DoV search will degrade to the conventional method.

System	Avg Frm Time(ms)	Polygons/frm	Nodes
CONV	63.66	136004	1102
VSC(.0001)	19.23	14604	116.5
VSC(.001)	18.23	13210	83.8
VSC(.01)	12.13	8723	46.8

Table 6.1: Run-time performance of the same walkthrough path by different DoV thresholds.

For the PB rendering mode, the polygon budget is a loose upper-bound for the total number of polygons to be sent to the graphics engine. Different upper bounds can change the LODs being used for each node. Our polygon budget rendering algorithm sets the LODs for active nodes based on the DoV values.

Figure 6.7 shows the number of polygons rendered under various polygon budgets. As shown in the figure, the number of polygons being rendered is well under the control of the respective budget number for most of the frames. This feature

is very useful for rendering complex scenes at high frame rate. As a comparison, the CONV mode can render unlimited number of polygons in a frame. Therefore, by changing the polygon budget, the PB mode provides a method to control the rendering performance while producing good visual quality. In figure 6.7, the number of polygons being rendered exceeds the budget somewhere around frame number 160. This is because the coarsest LOD of a complex model has occupied all the remaining budget. As the upper-bound is loose, the PB algorithm makes a few attempts to restrict the total number of polygons during the traversal. If all the attempts fail, it will terminate the traversal and render the current node. Our experiment also showed that when the budget is very large, the number of polygons rendered in each frame is very close to the CONV system. This confirms our prediction that if the budget becomes infinity, the system will degrade to conventional rendering scheme.

Figure 6.8 shows the screen shots at different polygon budgets. Note the difference in visually unimportant objects among the pictures (like the bunny and buildings around the screen center). For the PB 20000 walkthrough, the system obtains a speedup of 2.7 times as compared to the CONV walkthrough.

The second video file (available in *http://www.comp.nus.edu.sg/~shoulida/hdov.html*) shows two clips of the same walkthrough under different polygon budgets. The walkthrough performance of PB 20000 is much faster than that of PB 80000.

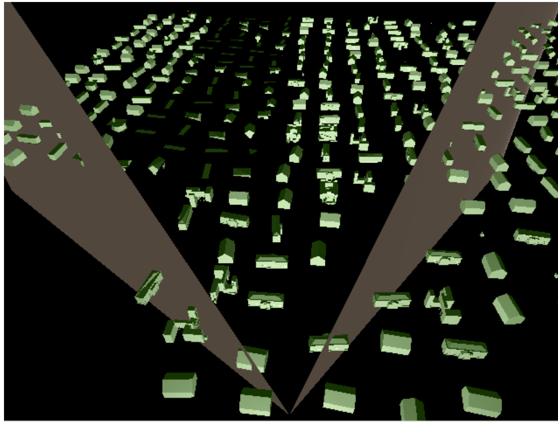
The VSC algorithm and the PB algorithm use different parameters to control the recursive traversal paths. The former uses a screen-projection related DoV

threshold to control the maximum allowable screen-projected area that a coarse LOD can be use in. While the latter uses an integer value, the polygon budget, to restrict the number of polygons that can be rendered. These two schemes can be chosen depending on the specific requirements to the application and the user's preferences. Results of the VSC mode show that it can control the visual quality of the rendering and improve the rendering performance. Results of the PB mode show that the polygon budget mode provides control to the system performance while optimizing the visual quality of the output.

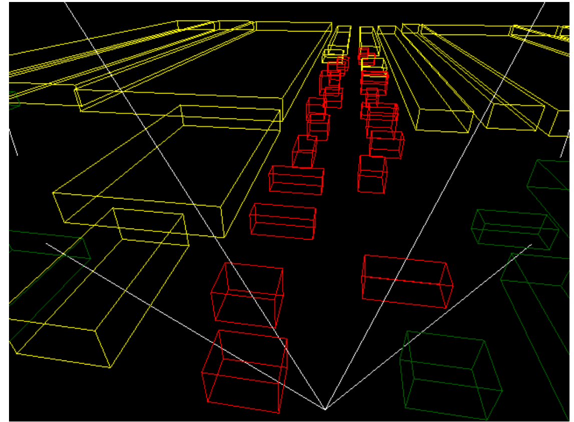
6.4 Summary

In this chapter we have presented a hierarchical degree-of-visibility scene tree structure, which combines the degree-of-visibility of objects or object groups with their level-of-details. Traversal (rendering) of the scene tree was based on the degree-of-visibility, which, as a comprehensive visual metric, could be precomputed. The method to precompute the DoV has been discussed. The real-time rendering could run in visibility simplification culling mode or polygon budget mode. Both rendering modes have shown promising performance in the memory HDoV scene tree.

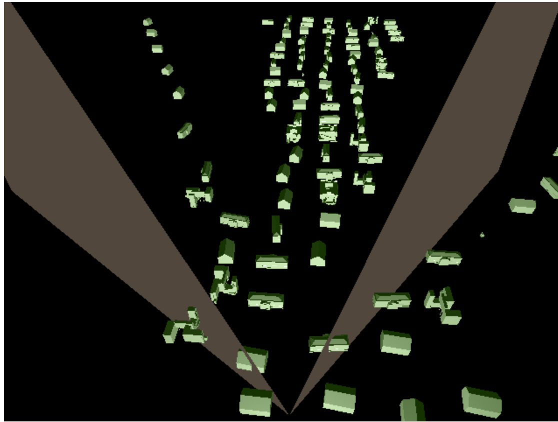
Implementation of an HDoV scene tree is straightforward. Since the structure of the HDoV-tree is generic, some of the existing scene hierarchies with LOD support for internal nodes can be easily updated to an HDoV scene tree.



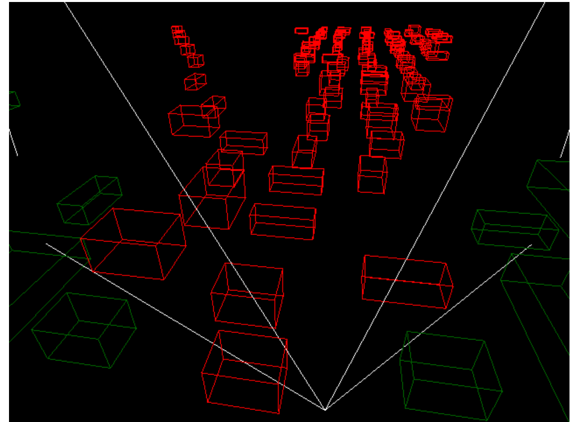
(a) bird's-eye view (VSC 0.001)



(b) wireframe (VSC 0.001)



(c) bird's-eye view (CONV)



(d) wireframe (CONV)

Figure 6.6: Snapshots of VSC and CONV modes. Note the very-low details of the active internal nodes on top-left corner of (a), which are represented by the yellow boxes in (b). (b) and (d) show the difference in the number of nodes being rendered. Although (a) looks as if containing more objects, it actually contains much fewer geometries than (c).

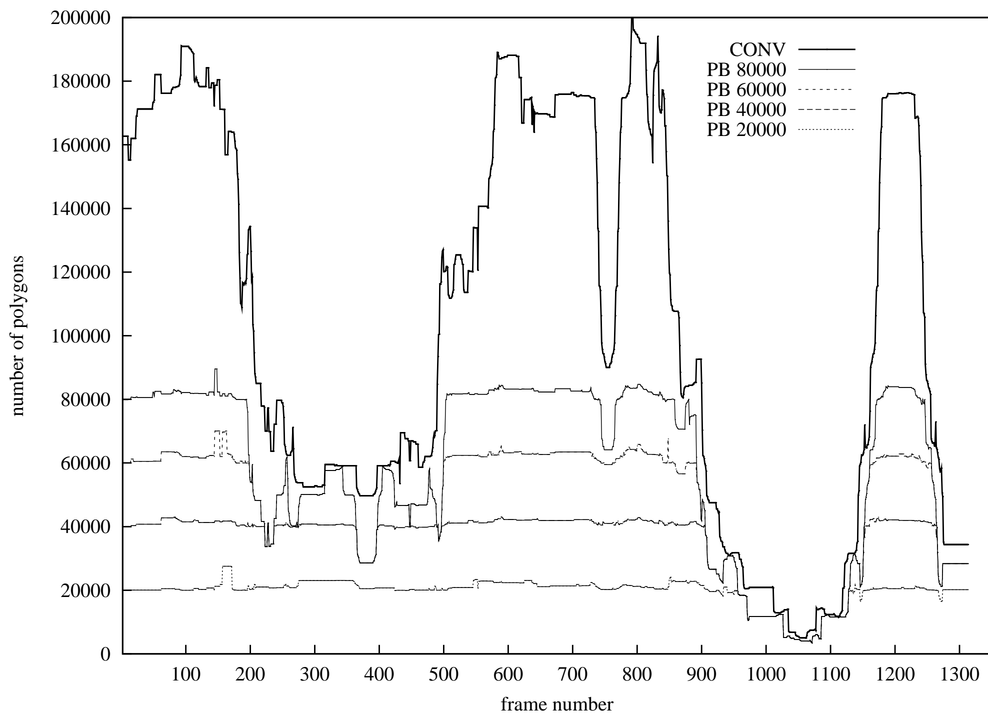
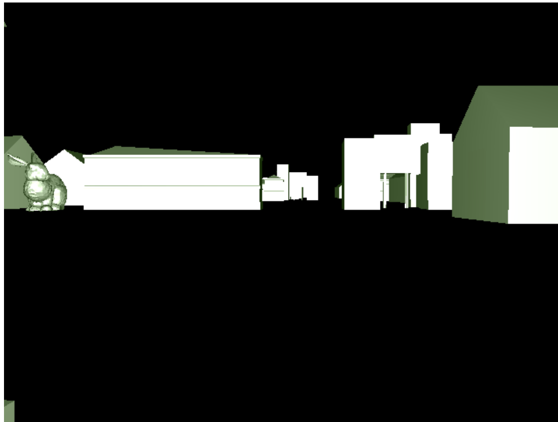
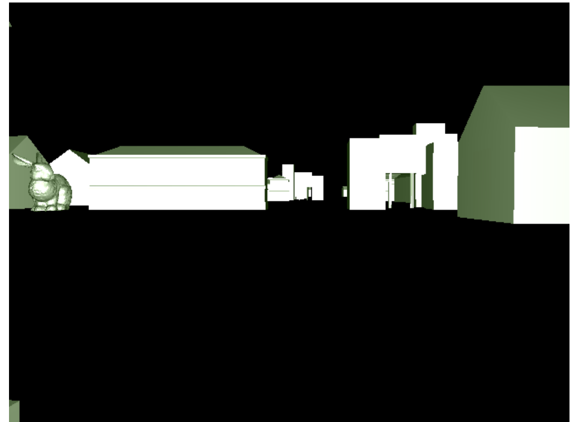


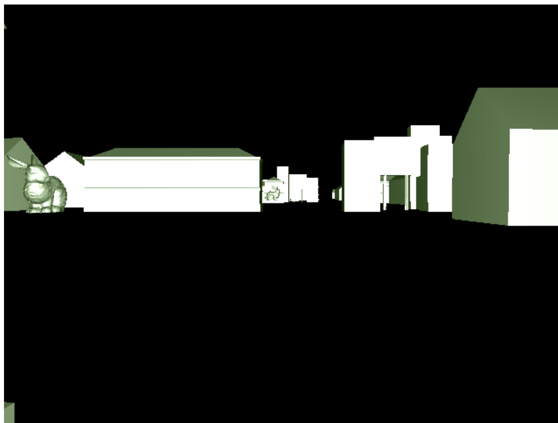
Figure 6.7: Number of polygons rendered in Polygon Budget Mode.



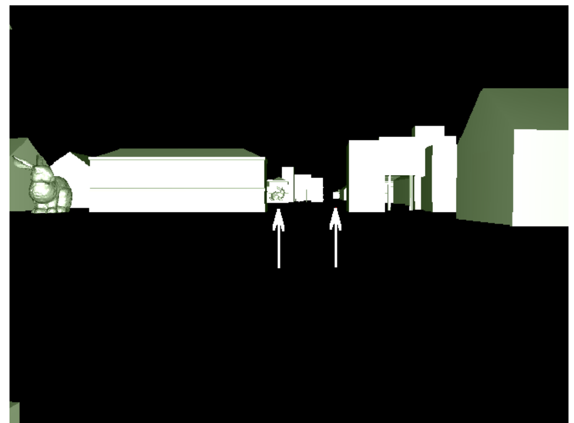
(a) screen shot, PB 20000



(b) screen shot, PB 40000



(c) screen shot, PB 60000



(d) screen shot, PB 80000

Figure 6.8: Snapshots of PB modes. Note the low detail of the building on the left in (a), the missing bunny in (a) and (b), and the missing building in the center of the screen in (a,b,c).

Chapter 7

Conclusion

In this thesis, we addressed the problem of real-time walkthrough in large Virtual Environment. In this chapter, we summarize our contributions in the thesis, and indicate possible directions for future research.

7.1 Summary Of Thesis Contributions

This thesis addressed a few problems in the interactive walkthrough systems which deploy databases of large-scale Virtual Environments. The work in the thesis is organized in three parts:

1. A generic architecture for virtual walkthrough of a large database of the VE (chapter 3),

2. Spatial techniques which optimize the data organization, manipulation, retrieval, and rendering of the large dataset (chapter 4), and

3. Visibility techniques which optimize the same things for a large database which integrates visibility, LOD, and spatial data (chapter 5 and 6).

We consider the following to be the major contributions of this thesis:

- A survey of existing work was presented in the literature on virtual walkthrough, related spatial techniques, visibility techniques and multi-resolution representations.

- A system framework for large database of Virtual Environment was designed to support high performance walkthrough.

  We proposed the generic system structure and presented various components required by interactive walkthrough systems. We identified the problems that hinder the run-time performance of such systems. We introduced various techniques and optimizations based on the scene graph structure, and discussed the mechanisms of the components in the system and how they work together.

- A novel search algorithm was proposed to query the database of the VE efficiently, despite the walkthrough path.

  We studied the spatial optimization techniques in the interactive walkthrough of disk-resident scenes. We observed that the data in the large walkthrough system had two distinct representations, the data stored in the secondary storage and the data in the scene graph. Two cells were defined for data retrieval and rendering in the REVIEW system. One was the disk-cell that controlled

the database query, the other was the frustum-cell that controlled the rendering. Each time the frustum-cell falls out of the disk-cell, a database query will be activated. The database was organized by a performance-optimized R-tree index. Since the regions in a walkthrough have always had large overlaps in consecutive queries, we proposed the complement search algorithm for the R-tree to remove the overlaps. As a result, the retrieval was more efficient. The performance experiments showed that the complement search algorithm performed better than the conventional R-tree in terms of disk I/O and search time.

- Optimization techniques such as prefetching, caching, and frustum culling were also designed to support real-time performance.

  The prefetching technique was designed to make predictions to the user's motion-path, while the distance-priority-LRU caching technique was designed to improve the traversal performance in the R-tree. The view frustum culling technique aimed at pruning irrelevant data in memory. The performance results showed that (1) the DPLRU cache replacement policy performed better than the conventional LRU cache, (2) the prefetching technique also improved the query performance and is nearly optimal (if not optimal) for a specific k value, (3) the frustum culling algorithm improved the rendering performance.

- We implemented the REVIEW walkthrough system, which employed the above mentioned spatial techniques, and conducted extensive experiments on it.

- A novel data structure was developed to capture the degree-of-visibility, the hierarchical LODs, and the spatial information.

  We presented a novel data structure, called *HDoV-tree*. The HDoV-tree extends the conventional spatial data structure by imbuing the *degree-of-visibility* attributes into the tree nodes. We gave the mathematical definitions of various concepts in the HDoV-tree. We also associated the hierarchy of LODs with the HDoV-tree. We developed the DoV-threshold based traversal algorithm to facilitate *visibility queries* on the HDoV-tree. The traversal path can therefore be determined by the visibility attributes in the levels of the nodes. We designed the image-space approach to compute DoV values for the objects. We also proposed three disk storage schemes for an efficient implementation of the HDoV-tree, as well as two cache replacement policies that are based on the DoV properties. The visibility query engine was also applied in a walkthrough context where much spatial coherence could be exploited. In the performance study, we compared the visibility query system with a naïve list-of-objects based system. We also compared the visual and performance results between the HDoV-tree based system and the REVIEW system. Our performance results showed that (1) the visibility query had better performance in I/O and search time as compared to the naïve system; (2) the indexed-vertical storage scheme was space-efficient and also outperformed the other two schemes, and therefore, designing appropriate storage structure is a crucial process for the system efficiency; (3) the scalability of the HDoV-tree was good in terms of search time and index I/Os; (4) the VI-

SUAL system which was based on HDoV tree had better overall performance and higher visual quality; (5) the proposed cache replacement policies performed better than the conventional LRU method when the cache size was small.

- We studied the memory version of the HDoV-tree.

  We implemented the threshold-based algorithm (VSC algorithm) for the memory HDoV scene tree. The threshold-based traversal also worked well for the memory HDoV-tree. We also designed a novel traversal algorithm which could provide *performance-guaranteed* rendering. The polygon budget algorithm could allocate the polygon budget according to the the degree-of-visibility of the tree nodes. The performance study showed that (1) the threshold-based traversal algorithm (VSC algorithm) had tunable speed and visual quality, and the performance of the VSC algorithm was better than the conventional traversal algorithm, (2) the polygon budget algorithm was effective when controlling the system performance while optimizing the visual output.

- We implemented VISUAL, a walkthrough system based on the HDoV-tree, and the memory HDoV scene tree based walkthrough system. And we conducted extensive performance studies and experiments on these walkthrough systems.

7.2 Future Work

There remain many open problems and issues to be addressed in the walkthrough system of large database of VEs, despite the research done so far. In this section, we shall get a perspective into the possible future work which are promising as extensions to the research work in this thesis.

The spatial data structures discussed in the thesis are all based on static datasets. It is assumed that all the object models and the spatial relations among them do not change by time. We note that a natural extension to the databases is to assume dynamic features to the entities in the datasets such as moving objects. We believe that the techniques described in chapter 4 can be extended into dynamic environments. In a dynamic data set, the critical issue is how we could organize and query objects to ensure high system performance. Dynamic spatial database has recently attracted much attention from the database community [68, 67, 54]. Can these techniques be adopted to support interactive virtual walkthrough? Another problem is how to develop appropriate data structures and efficient algorithms to enable DoV-based retrieving for a very large dynamic data set.

The virtual walkthrough systems proposed in the thesis run on a stand-alone machine. It is promising to extend the techniques into a network environment, where a server that owns a database allows a client to download the appropriate objects regarding the viewpoint position and motion pattern. Existing work makes the selection based on spatial relations of the viewer and the objects in the scene [84]. Can we select object representations for network transmission based on the HDoV-tree? With the HDoV-tree, a possible selection policy of the objects and

their LODs can be based on the Degree-of-Visibility, instead of pure spatial relations. We can also look at the problem of extending the work to next-generation Web environment, where the tree structure can be organized in device-independent format and can therefore be exchanged on heterogeneous platforms, and walked through on various platforms(devices).

The proposed techniques are designed for a single user environment. We can therefore consider the case where more than one user exist in the system. How to support multi-user walkthrough still remains to be a problem. In a multi-user system, can we explore some new techniques to share data among sessions of different users? For example, can the users be attracted by some common interesting features in the Virtual Environment? Can we exploit the query patterns of these multiple users to speed up the searches? The cache replacement policies discussed in previous chapters exploit the temporal coherence of a single user only. In multi-user environment, new caching techniques need to be developed.

A study on the relationship between performance, cell size, and page size is also a possible future work. Another possible future direction is to integrate methods in ORDBMS with the techniques presented in this thesis, as ORDBMS can better manage object data.

Bibliography

[1] M J Ackerman, VM Spitzer, AL Scherzinger, and DG Whitlock. The visible human data set: an image resource for anatomical visualization. *Medinfo*, 8(2):1195–1198, 1995.

[2] D. Aliaga, J. Cohen, A. Wilson, E. Baker, H. Zhang, C. Erikson, K. Hoff, and T. Hudson. MMR: An interactive massive model rendering system using geometric and image-based acceleration. In *ACM Symposium on Interactive 3D Graphics*, pages 199–206, Atlanta, USA, 1999.

[3] C. H. Ang and T. C. Tan. New linear node splitting algorithm for R-trees. In *Advances in Spatial Databases, SSD'97*, pages 339–349, Berlin, Germany, 1997.

[4] Ulf Assarsson and Tomas Möller. Optimized view frustum algorithms for bounding boxes. *journal of graphics tools*, 5(1):9–22, 2000.

[5] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, NJ, 5 1990.

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[7] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree : An index structure for high-dimensional data. In *Proceedings of the 22th International*

Conference on Very Large Data Bases (VLDB'96), pages 28–39, Mumbai, India, 9 1996.

[8] E. Bertino, B.C. Ooi, R. Sacks-Davis, K.L. Tan, J. Zobel, B. Shilovsky, and B. Catania. *Indexing Techniques for Advanced Database Systems*. Kluwer Academic Publishers, 8 1997.

[9] Lars Bishop, Dave Eberly, Turner Whitted, Mark Finch, and Michael Shantz. Designing a PC game engine. *Computer Graphics in Entertainment*, pages 46–53, 1 1998.

[10] B. Cabral and M. Hopcroft. An introduction to systems issues for walkthrough application, 1997.

[11] J. H. P. Chim, M. Green, R.W.H. Lau, H. V. Leong, and A. Si. On caching and prefetching of virtual objects in distributed virtual environments. In *ACM Multimedia*, pages 171–180, Bristol, UK, 1998.

[12] N. Chin and S. Feiner. Near real-time shadow generation using bsp trees. *Computer Graphics (SIGGRAPH'89 Proceedings)*, 23(3):99–106, 1989.

[13] Y. Chrysanthou and M. Slater. Shadow volume BSP trees for computation of shadows in dynamic scenes. In *Proceedings of the ACM Symposium on Interactive 3D Graphics*, pages 45–49, 4 1995.

[14] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[15] Michael F. Cohen and Donald P. Greenberg. The Hemi-Cube: A radiosity solution for complex environments. In *Computer Graphics (SIGGRAPH'85 Proceedings)*, volume 19, pages 31–40, 1985.

[16] D. Cohen-Or, Y. Chrysanthou, and C. Silva. A survey of visibility for walkthrough applications. In *Proc. of EUROGRAPHICS'00, course notes*, 2000.

[17] Daniel Cohen-Or, Gadi Fibich, Dan Halperin, and Eyal Zadicario. Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. *Computer Graphics Forum*, 17(3):243–254, 1998.

[18] S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. In *Proc. of the ACM Symposium on Interactive 3D Graphics*, pages 83–90, 4 1997.

[19] Frédo Durand. 3D visibility: Analytical study and applications. In *PhD thesis, Universite Joseph Fourier*, 1999.

[20] Frédo Durand, George Drettakis, Joëlle Thollot, and Claude Puech. Conservative visibility preprocessing using extended projections. In *Proc. of SIGGRAPH 2000, Computer Graphics Proceedings*, pages 239–248, 2000.

[21] C. Erikson and D. Manocha. GAPS: General and automatic polygonal simplification. In *Proc. Symposium on Interactive 3D Graphics (I3D'99)*, pages 79–88, Atlanta, GA, 1999.

[22] C. Erikson, D. Manocha, and W. Baxter. HLODs for fast display of large static and dynamic environments. In *Proc. ACM Symposium on Interactive 3D Graphics*, pages 111–120, 2001.

[23] Évelyne Klinger, Isabelle Chemin, Patrick Légeron, Stéphane Roy, Françoise Lauer, and Pierre Nugues. Issues in the design of virtual environments for the treatment of social phobia. In *VRMHR 2002, Proceedings of the 1st International Workshop on Virtual Reality Rehabilitation (Mental Health, Neurological, Physical, Vocational)*, pages 261–273, Lausanne, Switzerland, November 7-8 2002.

[24] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proc. the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 247–252, 1989.

[25] T. A. Funkhouser and C. H. Sequin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *ACM Computer Graphics Proceedings, Annual Conference Series*, pages 247–254, 1993.

[26] T. A. Funkhouser, C. H. Sequin, and S. J. Teller. Management of large amounts of data in interactive building walkthroughs. In *Proc ACM SIGGRAPH Symposium on Interactive 3D Graphics*, pages 11–20, Boston, March 1992.

[27] Volker Gaede and Oliver Günther. Multidimensional access methods. *Computing Surveys*, 30(2):170–231, 1998.

[28] M. Garland and P.S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of ACM SIGGRAPH 97*, pages 209–216, 1997.

[29] Michael Garland. Quadric-based polygonal surface simplification, 1999.

[30] Michael Garland and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In *IEEE Visualization'98*, pages 263–270, 1998.

[31] Daniel Green and Don Hatch. Fast polygon-cube intersection testing. *Graphics Gems V*, pages 375–379, 1995.

[32] Ned Greene. Detecting intersection of a rectangular solid and a convex polyhedron. *Graphics Gems IV*, pages 74–82, 1994.

[33] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical Z-buffer visibility. *Computer Graphics (SIGGRAPH 93 Proceedings)*, 27(Annual Conference Series):231–238, 1993.

[34] Oliver Günther. The design of the Cell tree: An object-oriented index structure for geometric databases. In *Proc. of the 5th International Conference on Data Engineering (ICDE'89)*, pages 598–605, Los Angeles, California, 2 1989.

[35] Oliver Günther and Jeff Bilmes. Tree-based access methods for spatial databases: Implementation and performance evaluation. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 3(3):342–356, 1991.

[36] Oliver Günther and Hartmut Noltemeier. Spatial database indices for large extended objects. In *Proc. of the 7th IEEE International Conference on Data Engineering (ICDE'91)*, pages 520–526, Kobe, Japan, 4 1991.

[37] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[38] J. Heo, S. Jung, and K. Wohn. Exploiting temporally coherent visibility for accelerated walkthroughs. *Computers & Graphics*, 21(4):507–517, 1997.

[39] K. Hoff. Fast AABB/view-frustum overlap test, 1997. http://www.cs. unc. edu/˜hoff/ research/index.html.

[40] H. Hoppe. Progressive meshes. In *Proc. of SIGGRAPH 96 Conference, Annual Conference Series, pp. 99-108*, August 1996.

[41] HP. HP OpenGL 1.1 reference, 2000. http://www.hp.com/workstations/support/ documentation/ manuals/user_guides/graphics/opengl/Reference.html.

[42] Henk Huitema and Robert van Liere. Interactive visualization of protein dynamics. In *IEEE Visualization*, pages 465–468, 2000.

[43] Andreas Hutflesz, Hans-Werner Six, and Peter Widmayer. The R-file: An efficient access structure for proximity queries. In *Proc. of the 6th International Conference on Data Engineering (ICDE'90)*, pages 372–379, Los Angeles, California, 2 1990.

[44] H. V. Jagadish. Spatial search with polyhedra. In *Proc. of the 6th International Conference on Data Engineering (ICDE'90)*, pages 311–319, Los Angeles, CA, 2 1990.

[45] Low K.L. and Tan T.S. Model simplification using vertex-clustering. In *Proc Interactive 3D Graphics*, pages 75–81, Rhode Island, April 1997.

[46] James T. Klosowski and Cláudio T. Silva. Rendering on a budget: A framework for time-critical rendering. In *IEEE Visualization'99*, pages 115–122, 1999.

[47] M. Kofler, M. Gervautz, and M. Gruber. R-trees for organizing and visualizing 3d gis databases. *Journal of Visualization and Computer Animation*, 11:129–143, 2000.

[48] V. Koltun, Y. Chrysanthou, and D. Cohen-Or. Virtual occluders: An efficient intermediate pvs representation. In *EUROGRAPHICS Workshop on Rendering*, pages 59–70, 2000.

[49] Hans-Peter Kriegel and Bernhard Seeger. PLOP-hashing: A grid file without directory. In *Proc. of the 4th IEEE International Conference on Data Engineering (ICDE'88)*, pages 369–376, Los Angeles, California, 2 1988.

[50] M Krus, P. Bourdot, F. Guisnel, and G. Thibault. Levels of detail & polygonal simplification, 2001. http://www.acm.org/crossroads/xrds3-4/levdet.html.

[51] Subodh Kumar, Dinesh Manocha, William Garrett, and Ming Lin. Hierarchical backface computation. *Computer and Graphics (Special Issue on Visibility)*, 9(5):681–692, 1999.

[52] R. W. H. Lau, M. Green, D. To, and J. Wong. Real-time continuous multi-resolution method for models of arbitary topology. *Presence: Teleoperators and Virtual Environments*, 7(1):22–35, 1998.

[53] Jonathan K. Lawder and Peter J. H. King. Querying multi-dimensional data indexed using the hilbert space-filling curve. *SIGMOD Record*, 30(1):19–24, 2001.

[54] Iosif Lazaridis, Kriengkrai Porkaew, and Sharad Mehrotra. Dynamic queries over mobile objects. In *Proc. of 8th Intl. Conf. on Extending Database Technology (EDBT'2002)*, pages 269–286, Prague, Czech, 2002.

[55] D.B. Lomet and B. Salzberg. The hB-tree: A robust multiattribute search structure. In *Proc. 5th IEEE International Conference on Data Engineering (ICDE'89)*, pages 296–304, Los Angeles, CA, 2 1989.

[56] Stella Mills and Jan Noyes. Virtual reality: an overview of user-related design issues. *Interacting with Computers*, 11(4):375–386, 1999.

[57] Yutaka Ohsawa and Masao Sakauchi. A new tree type data structure with homogeneous nodes suitable for a very large spatial database. In *Proc. of the 6th IEEE International Conference on Data Engineering (ICDE'90)*, pages 296–303, Los Angeles, California, 2 1990.

[58] Beng Chin Ooi, Ron Sacks-Davis, and Ken J. McDonell. Spatial indexing by binary decomposition and spatial bounding. *Information Systems Journal*, 16(2):211–237, 1991.

[59] Jack A. Orenstein. Redundancy in spatial databases. In James Clifford, Bruce G. Lindsay, and David Maier, editors, *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD'89)*, pages 294–305, Portland, Oregon, 5 1989. ACM Press.

[60] R. Pajarola, T. Ohler, P. Stucki, K. Szabo, and P. Widmayer. The alps at your fingertips: Virtual reality and geoinformation systems. In *Proceedings of the ICDE'98 Conference*, pages 550–557, 1998.

[61] S. Pettifer. An operating environment for large scale virtual reality, 1999.

[62] J. Rohlf and J. Helman. Iris performer:a high performance multiprocessing toolkit for real-time 3d graphics. In *Proc. 1994 Computer Graphics Proceedings, Annual Conference Series*, pages 381–394, 1994.

[63] Rémi Ronfard and Jarek Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum*, 15(3):67–76, 1996.

[64] J. Rossignac and P. Borrel. *Multi-resolution Approximations for Rendering Complex Scenes*. Springer-Verlag, New York, 1993.

[65] Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, pages 17–31, Austin, Texas, 5 1985.

[66] Yixin Ruan, Jason Chionh, Zhiyong Huang, Kian-Lee Tan, and Lidan Shou. Balancing fidelity and performance in virtual walkthrough. In *The 6th IFIP Working Conference on Visual Database Systems(VDB'6)*, pages 219–233, Brisbane, Australia, 2002.

[67] Simonas Saltenis and Christian S. Jensen. Indexing of moving objects for location-based services. In *Proc. of 18th Intl. Conf. on Data Engineering (ICDE'2002)*, San Jose, CA, 2002.

[68] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In *ACM SIGMOD Conference*, pages 331–342, 2000.

[69] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, MA, 1990. ISBN 0-201-50300-0.

[70] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, MA, 1990. ISBN 0-201-50255-0.

[71] C. Saona-Vázquez, I. Navazo, and P. Brunet. The visibility octree: A data structure for 3d navigation. *Computers & Graphics*, 23:635–643, 1999.

[72] G. Schaufler, J. Dorsey, X. Decoret, and F. X. Sillion. Conservative volumetric visibility with occluder fusion. In *Proc. of SIGGRAPH 2000, Computer Graphics Proceedings*, pages 229–238, 2000.

[73] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH 92)*, 26(2):65–70, October 1992.

[74] Bernhard Seeger and Hans-Peter Kriegel. Techniques for design and implementation of efficient spatial access methods. In *14th International Conference on Very Large Data Bases (VLDB'88)*, pages 360–371, Los Angeles, California, 9 1988. Morgan Kaufmann.

[75] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R^+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of 13th International Conference on Very Large Data Bases (VLDB'87)*, pages 507–518, Brighton, England, 9 1987. Morgan Kaufmann.

[76] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder. Hierarchical image caching for accelerated walk-throughs of complex environments. In *Proc. of Computer Graphics (SIGGRAPH'96)*, pages 75–82, 1996.

[77] Lidan Shou, Jason Chionh, Zhiyong Huang, Yixin Ruan, and Kian-Lee Tan. Managing gigabyte virtual environment for walkthrough applications. In *Eurographics 2001 (Short Presentation)*, Manchester, UK, 2001.

[78] Lidan Shou, Jason Chionh, Yixin Ruan, Zhiyong Huang, and Kian-Lee Tan. REVIEW: A real time virtual walkthrough system (DEMO). In *Proc. of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD'01)*, page 601, Santa Barbara, CA, May 2001.

[79] Lidan Shou, Jason Chionh, Yixin Ruan, Zhiyong Huang, and Kian-Lee Tan. Walking through a very large virtual environment in real-time. In *Proc. of the 27th International Conference on Very Large Data Bases (VLDB'2001)*, pages 401–410, Roma, Italy, 2001.

[80] Lidan Shou, Zhiyong Huang, and Kian-Lee Tan. Supporting real-time visualization with the HDoV tree. In *Proc of the 2003 ACM Symposium on Applied Computing*, pages 966–971, Florida, USA, 2002.

[81] Lidan Shou, Zhiyong Huang, and Kian-Lee Tan. HDoV tree: The structure, the storage, the speed. In *Proc of The 19th IEEE International Conference on Data Engineering (ICDE'2003)*, pages 557–568, Bangalore, India, 2003.

[82] Hans-Werner Six and Peter Widmayer. Spatial searching in geometric databases. In *Proc. of the 4th International Conference on Data Engineering (ICDE'88)*, pages 496–503, Los Angeles, California, 2 1988.

[83] Bjarne Stroustrup. *The C++ Programming Language (Third Edition)*. Addison Wesley Longman, 1997. ISBN 0-201-88954-4.

[84] Eyal Teler and Dani Lischinski. Streaming of complex 3D scenes for remote walk-throughs. *Computer Graphics Forum(Eurographics 2001)*, 20(3):17–25, 2001.

[85] D. Thalmann. The role of virtual humans in virtual environment technology and interfaces, 1999.

[86] W. Thibault and B. Naylor. Set operations on polyhedra using binary space partitioning trees. *Computer Graphics*, 21(4):153–162, 7 1987.

[87] R. van Gaal, U. Schuerkamp, D. Pospisil, and P. Harrington. Racer: a free car simulation project, 2001-2002. http://www.racer.nl/.

[88] Y. Wang, P.K. Agarwal, and S. Har-Peled. An on-line occlusion culling algorithm for fast walkthrough in urban areas. In *Eurographics (Short presentation)*, 2001.

[89] Alan Watt and Mark Watt. *Advanced animation and rendering techniques: Theory and Practice*. Addison-Wesley, 1992.

[90] M. Wimmer, M. Giegl, and D. Schmalstieg. Fast walkthrough with image caches and ray casting. *Computers & Graphics*, 23:831–838, 1999.

[91] Geoffrey Wong and Vincent Wong. Virtual reality in space exploration, 1996.

[92] P. Wonka, M. Wimmer, and D. Schmalstieg. Visibility preprocessing with occluder fusion. In *Eurographics Workshop on Rendering 2000*, pages 71–82, June 2000.

[93] Hansong Zhang and Kenny Hoff. Fast backface culling using normal masks. In *ACM Symposium on Interactive 3D Graphics (I3D)*, pages 103–106, 1997.

[94] Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth E. Hoff III. Visibility culling using hierarchical occlusion maps. *Computer Graphics (SIGGRAPH 97)*, 31(Annual Conference Series):77–88, 1997.