HDoV-tree: The Structure, The Storage, The Speed

Lidan Shou Zhiyong Huang Kian-Lee Tan

National University of Singapore
Department of Computer Science
3 Science Drive 2, Singapore 117543
{*shoulida, huangzy, tankl*}*@comp.nus.edu.sg*

Abstract

In a visualization system, one of the key issues is to optimize performance and visual fidelity. This is especially critical for large virtual environments where the models do not fit into the memory. In this paper, we present a novel structure called HDoV-tree that can be tuned to provide excellent visual fidelity and performance based on the degree of visibility of objects. HDoV-tree also exploits internal level-of-details (LoDs) that represent a collection of objects in a coarser form. We also propose three storage structures for the HDoV-tree. We implemented HDoV-tree in a prototype walkthrough system called VISUAL. We have evaluated the HDoV-tree on visibility queries, and also compared the performance of VISUAL against REVIEW, a walkthrough system based on R-tree. Our results show that the HDoV-tree is an efficient structure. Moreover, VISUAL can lead to high frame rates without compromising visual fidelity.

1. Introduction

Interactive visualization systems are gaining importance in various applications in industry, academics and entertainments. The models to be visualized are getting increasingly complex. In some applications such as a large-scale Virtual Environment, or a CAD software, the models consist of tens of thousands of objects, each of which may consist of thousands of polygons. Moreover, each object typically has multi-resolution representations called level-of-details (LoDs). Thus, the storage requirement of such systems is extremely large, ranging from thousands of megabytes to hundreds of gigabytes and even terabytes. Conventional visualization systems that assume models fit into the main memory are no longer affordable. This calls for novel techniques to handle models that are too large to fit into the memory, and has to be stored in disk.

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.

In this paper, 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

Proceedings of the 19th International Conference on Data Engineering (ICDE'03)
1063-6382/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

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 computing power of the machines, different users may see visible objects with different degree of fidelity.

We propose a threshold-based traversal algorithm that allows the structure 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. Our results show that the proposed scheme is efficient and provide excellent visual fidelity. Our VISUAL system also outperforms REVIEW, a walkthrough system based on R-tree, in terms of frame rates and visual fidelity.

The rest of this paper is organized as follows. In the next section, we review some related work on managing large virtual environments. We present the structure of the HDoV-tree in Section 3. The search method of the HDoV-tree is also discussed in this section. In Section 4, we present the storage schemes for the HDoV-tree. Section 5 reports the results of our experimental study on our prototype system, and finally, we conclude in Section 6.

2. Related Work

Most of the earlier works have assumed that virtual environments are memory resident. It is not until recently that managing large virtual environments has become an active area [8, 12]. Most of the works have adopted spatial indexes to organize, manipulate and retrieve the data in large virtual environments.

Kofler et al. proposed the LoD-R-tree [8] that combines the R-tree index with a hierarchy of multi-representations of the three-dimensional data. This data structure considers only the spatial proximity of objects and does not incorporate any visibility data. To minimize the amount of data to be fetched from disk, the search method converts the viewing-frustum into a few rectangular query boxes (instead of one single large query box that bounds the view frustum), and retrieves only objects within these boxes. Thus, the structure leads to high frame rates as long as the user stays within the viewing-frustum. However, its performance degenerates significantly as the user view changes.

In [12], Shou et al. proposed the REVIEW walkthrough system. REVIEW also exploits spatial proximity for retrieving visible objects. It employs R-tree as the underlying spatial data structure, but extended the R-tree search scheme such that data that have been retrieved in earlier operations do not need to be accessed again. It also supports a semantic-based cache replacement strategy based on spatial distance between the viewer and the nodes. Prefetching and in-memory optimization are some other optimization strategies that have been deployed to improve the system performance.

Although spatial access method offers a neat solution to real-time visualization, it suffers from two problems. First, it may miss some visible objects that are far from the viewpoint. This is because a typical spatial query only retrieves objects that are within (or overlap) the query box, and thus visible objects that are out of the query box will not be retrieved. As we shall see in our experimental study, this may affect the visual fidelity of the scene. Second, it may waste I/O and memory resources by retrieving objects that are "hidden". These are objects that are located within the spatial query box, but are not visible because they are blocked by other larger objects.

We believe a better solution to the aforementioned two problems is to compute the visibility of objects with respect to a viewpoint or cell. Many visibility algorithms that compute objects that are visible from a given viewpoint or a viewing cell have been proposed by the computer graphics community, for example, the work of Teller and Sequin [13], Funkhouser et al. [5], Aliaga et al. [2], Bittner et al. [4], Saona-Vázquez et al. [10], Agarwal et al. [1], Koltun et al. [9]. However, computing visibility at runtime is expensive. Moreover, how the datasets and visibility data are managed at run-time has not been reported.

3. The HDoV-Tree

As described in the introduction, one straightforward method to managing visibility in large virtual environments is to partition the viewpoint space into disjoint cells, and associate each cell with a list of visible objects. At runtime, the objects corresponding to the viewpoint are loaded into memory. We shall refer to this method as the (cell, list-of-objects)-based method. As noted in the introduction, there are several limitations with such a method. In this paper, we adopt a similar strategy of partitioning the viewpoint space into disjoint cells. However, we propose a novel tree structure, HDoV-tree, to overcome the limitations of the simple method. In this section, we shall discuss the tree structure and the traversal algorithm, and defer the discussion on the storage schemes to the next section. Before that, we shall introduce the novel concept of degree-of-visibility.

3.1. Degree of Visibility

In computer graphics, existing visibility algorithms usually mark an object as *visible* or *invisible*. We observe that such boolean representation tends to be too 'conservative'

Proceedings of the 19th International Conference on Data Engineering (ICDE'03)
1063-6382/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

as it will mark an object as visible even if only a very small part of it can be seen.

We introduce *degree-of-visibility* (DoV) to represent visibility more precisely. Firstly, 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 points s, whose interconnecting line with p, $\overline{sp}$, intersects O, while s is not in O [10]. 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 as:

$$X_{visible} = X - \bigcup_i S(p, O_i) . \quad (1)$$

We define the Degree of Visibility (DoV) 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

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

The DoV provides an indication on how visually important an object is, considering all possible viewing directions. The geometric meaning of DoV is the solid angle of the "visible" part of the point set. Thus, DoV of an object takes on values between 0 and 1. An object with DoV value of 0 is unimportant since it is hidden from the viewpoint and therefore should not be accessed. On the other hand, an object that is completely visible (with DoV > 0) should be retrieved. Intuitively, the larger the DoV value of an object, the more likelihood that it be noticed and so it is more critical for it to be shown in greater details. On the contrary, an object that has very low DoV value with respect to a viewpoint may not be noticeable, and hence can be represented by a coarse LoD.

The concept of DoV can also be extended to a group of objects, where the DoV of a group is defined as if the aggregation of the group of objects is an individual point set. For a viewing region(cell), the DoV of an object viewed from region $\mathbb{R}$ can be defined conservatively as

$$DoV(\mathbb{R}, X) = max(DoV(p, X)), \qquad \forall p \in \mathbb{R} \quad (2)$$

In this paper, we use region-based DoV.

3.2. The Logical Structure of The HDoV-Tree

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 1 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 are visible.

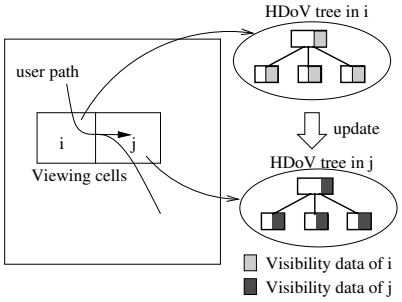


Figure 1. Dynamic update of the HDoV-tree.

In this paper, for simplicity as well as because we are dealing with 3D objects only, we employ the R-tree [7] as the spatial structure in our implementation.

Figure 2 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 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 values 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

Proceedings of the 19th International Conference on Data Engineering (ICDE'03)
1063-6382/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

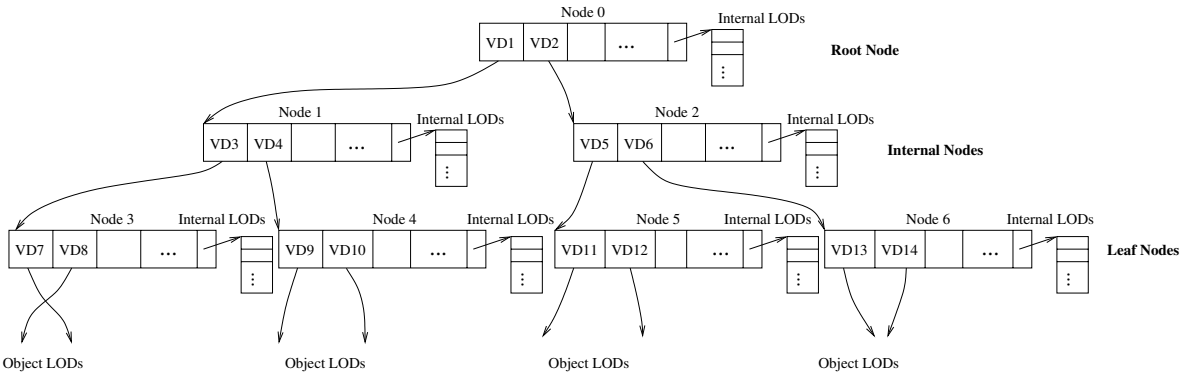


Figure 2. A Hierarchical Degree-of-Visibility Tree.

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.

Since the DoV values stored in the tree depends 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 viewpoint, 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 field is 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 in Section 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. η 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 current view frustum can be traversed first, while regions that are outside the view frustum can be delayed. This can further improve the response time significantly.

3.3. 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 coherency). As such, in this section, we shall focus on just point visibility query.

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 tradeoff the performance and the visual quality.

Proceedings of the 19th International Conference on Data Engineering (ICDE'03)
1063-6382/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

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.

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 3 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 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

$$VD = (DoV, NVO)$$

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 3). Suppose 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

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. Add E.ptr→LOD_{leaf} into result; //equation 6
6. else
7. if(E.DoV $\leq \eta$ and $h(1 + log_M s) < log_M$(E.NVO))
8. Add E.ptr→$LOD_{internal}$ into result; //equation 5
9. else
10. Search(E.ptr);
11. **end**;

Figure 3. The HDoV-tree traversal algorithm.

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 each visible object in the leaf nodes has f polygons, and the ratio of the number of polygons in parent nodes over the sum of those in child nodes is s, or $s = \frac{npoly(node)}{\sum_i npoly(child_i)}$, then the estimated number of polygons in node N is $m \cdot f \cdot s^h$. On the other hand, the number of polygons in the visible leaf nodes sum up to $f \cdot n$. So the condition to terminate the traversal is

$$m \cdot f \cdot s^h < f \cdot n \,. \tag{3}$$

Substituting m into equation 3 gives

$$h(1 + log_M s) < log_M n \,. \tag{4}$$

LoD of an active internal node can be selected as

$$LoD_{internal} = \frac{DoV}{\eta} LoD_{highest} + \left(1 - \frac{DoV}{\eta}\right) LoD_{lowest} \tag{5}$$

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

$$LoD_{leaf} = k \cdot LoD_{highest} + (1 - k)\, LoD_{lowest} \tag{6}$$

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 $MAXDOV = 0.5$.

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

Proceedings of the 19th International Conference on Data Engineering (ICDE'03)
1063-6382/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

viewpoints 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 paper, 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 information 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 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 1. Some notations.

4.1. The horizontal storage scheme

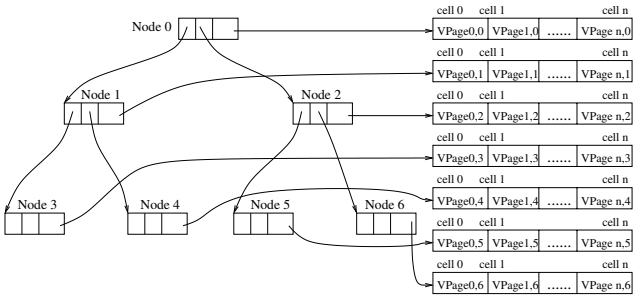


Figure 4. Horizontal Storage Scheme. *VPage i,j* **represents the V-page of node** *j* **in cell** *i*.

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 4 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, can be estimated as (We have not included the storage for the tree structure as all the storage schemes have similar storage.)

$$size_{vpage} \cdot c \cdot N_{node} .$$

4.2. The vertical storage scheme

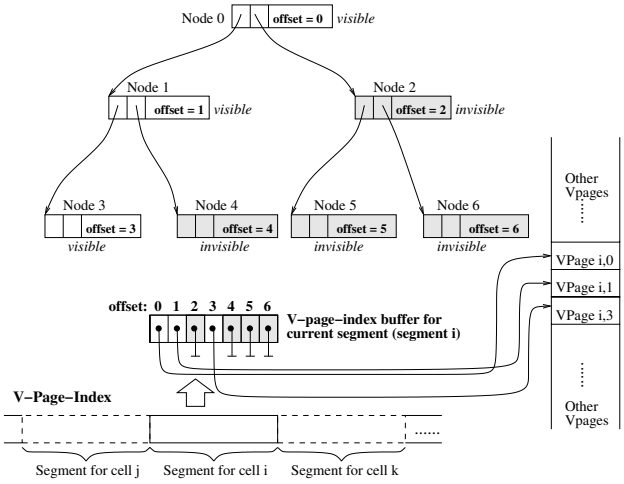


Figure 5. Vertical Storage Scheme.

Another scheme, which requires lesser storage space, is the *vertical scheme*. As figure 5 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 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

Proceedings of the 19th International Conference on Data Engineering (ICDE'03)
1063-6382/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

offset value stored in N. If the V-page pointer is nil, it means 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 (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.)

$$size_{pointer} \cdot N_{node} \cdot c + size_{vpage} \cdot N_{vnode} \cdot c\,.$$

Since N_{vnode} is 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 costly.

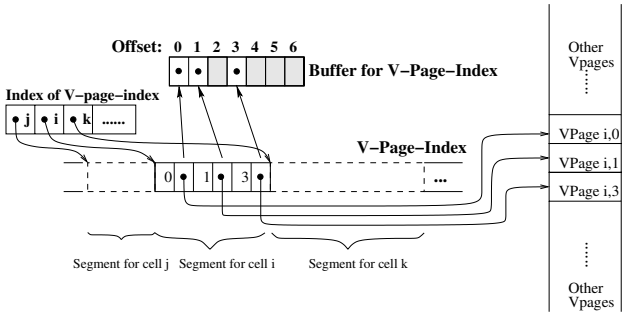


Figure 6. Indexed Vertical Storage Scheme.

4.3. The indexed-vertical storage scheme

As mentioned, "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 6 shows (The tree nodes are omitted, as they are the same as those in figure 5). 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

$$(size_{pointer} + size_{integer}) \cdot N_{vnode} \cdot c + size_{vpage} \cdot N_{vnode} \cdot c\,.$$

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

$$N_{vnode} \leq N_{vobj} \cdot levels \leq N_{vobj} \cdot log_m N_{obj} \qquad (7)$$

where m is the minimum number of entries for non-root nodes defined for the R-tree.

5. Performance Study

5.1. 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 specific 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* [6]. Internal LoDs of nodes at higher levels are then generated in a bottom-up order.

Proceedings of the 19th International Conference on Data Engineering (ICDE'03)
1063-6382/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

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. As computing DoV is not the focus point of this paper, the reader is referred to [11] for more details on the DoV computation algorithm. DoV values of objects and nodes are then stored in *V-Page*s according to the various storage structures. For the largest 1.6 GB dataset, we generated the internal LoDs and precomputed the DoVs for more than 4000 viewing cells. The precomputation takes about 1.02 seconds for each cell.

Two sets of experiments are conducted. (1) In the first set, we evaluate the 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 this latter scheme as the naïve method. (2) In the second set, we study the real-time walkthrough performance of VISUAL. We use the REVIEW system [12] as our reference. Recall (see Section 2) that REVIEW is a real-time walkthrough system that indexes objects using R-tree, and performs window queries in accessing the objects during a walkthrough session. 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 in [0, 0.008].

5.2. The storage cost of the storage schemes

We begin by considering the storage overhead of the HDoV-tree under the three storage schemes. Table 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 2. Storage space required by the schemes.

5.3. Experiment 1: 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 5.4.

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

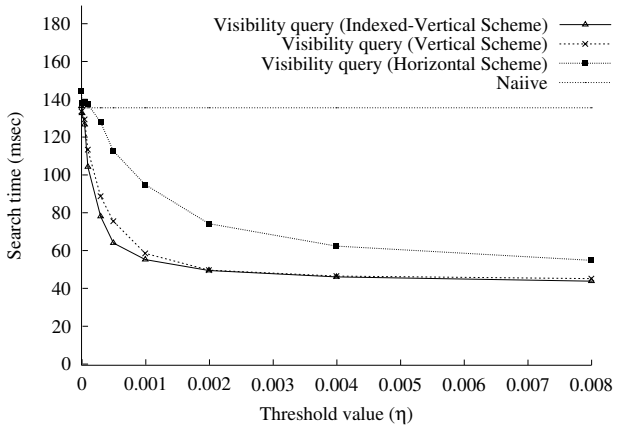


Figure 7. Search time with different η values.

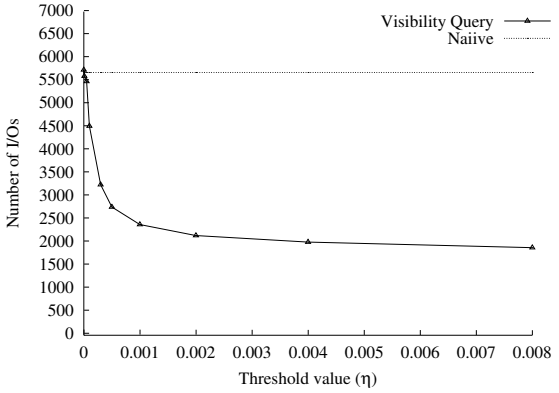
Figure 7 shows the results of the search time as η (DoV threshold) varies from 0 to 0.008. From the figure, we observe that when η increases, the search time for all HDoV-tree-based schemes decrease significantly. This is expected as a large η value implies that the traversal will terminate more often at internal nodes. As a 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 $\eta = 0$.

For the HDoV-tree-based schemes, we note that the performance of the vertical scheme and the indexed-vertical scheme is comparable. The performance of the indexed-vertical scheme is only marginally better than that of the vertical scheme as the former 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.

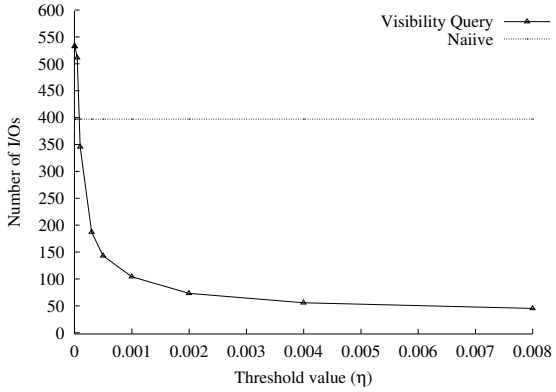
Proceedings of the 19th International Conference on Data Engineering (ICDE'03)
1063-6382/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

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.



(a) I/O cost of the visibility queries



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

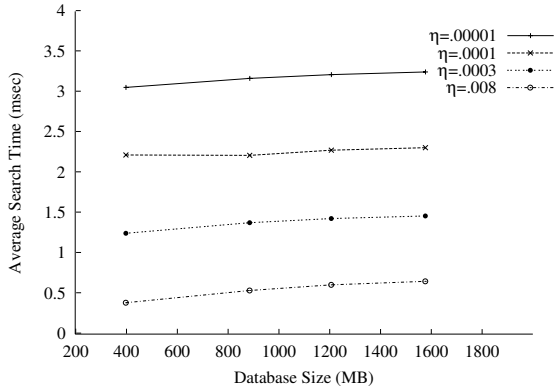
Figure 8. Performance results on disk I/Os.

Figure 8(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 8(b).

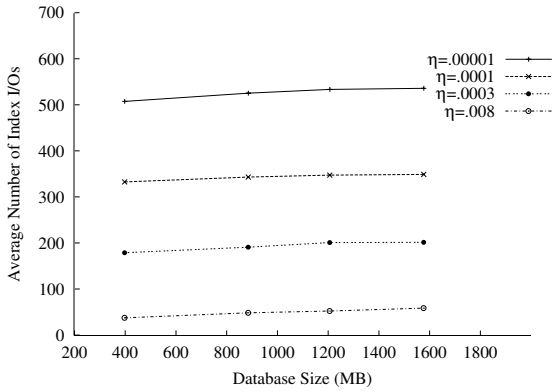
From figure 8(b) we found that for very small η values, the light-weighted I/O cost of HDoV-tree is higher than that of the naïve search. This is expected, as extra I/Os are needed to access the internal nodes and internal V-pages of the HDoV-tree. However, as η increases, the costs accessing the internal nodes are compensated by the much larger benefits we obtain 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 datasets 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 9(a) and 9(b) respectively. 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 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.



(a) Average search time of datasets with different sizes



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

Figure 9. Scalability of the visibility query.

Proceedings of the 19th International Conference on Data Engineering (ICDE'03)
1063-6382/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

5.4. Experiment 2: 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. 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 called the *complement search* algorithm.

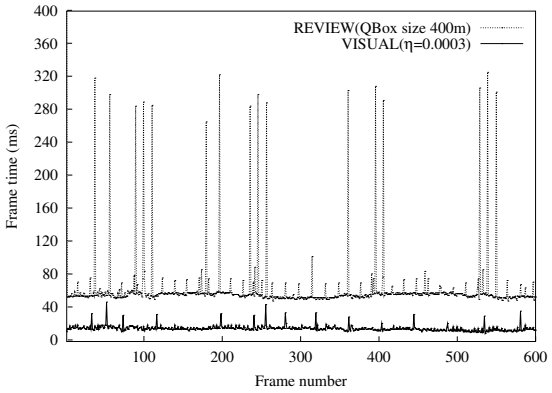
The main metrics that we use for comparing the performance of interactive walkthrough are *average frame time* and *variance of frame time*. We recorded a few walkthrough sessions and played them back on the interactive walkthrough application. Each session is played back on both the VISUAL system and the REVIEW system. None of the two systems caches the tree nodes in the queries.

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

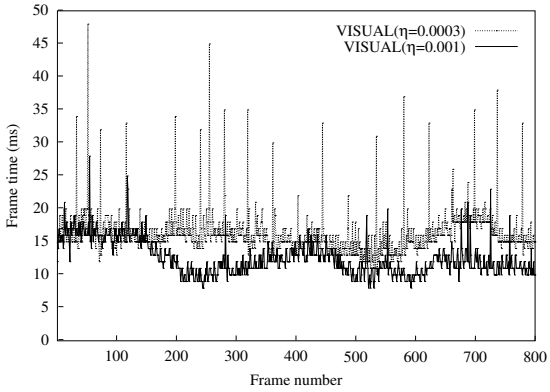
Figure 10(a) shows the results of time spent on each rendering frame between the spatial-query based REVIEW system and the VISUAL system. The size of the query box in the REVIEW system is set to 400m. The visual quality of the REVIEW system in this case is slightly worse, though comparable to the VISUAL system. However, the rendering frame time is very different as shown in figure 10(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 10(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.

We also evaluated REVIEW with different query box sizes, and the results showed that REVIEW is not as effective as VISUAL. While its average frame time is comparable to VISUAL for small boxes, the spikes caused by spatial queries are tall. Moreover, as we will soon discuss, the visual quality is unacceptable compared to VISUAL.

Figure 10(b) compares the results for VISUAL using two different threshold values: $\eta = 0.001$ and $\eta = 0.0003$. As



(a) Comparison between VISUAL($\eta = 0.001$) and REVIEW (large query boxes)



(b) Comparison between VISUAL($\eta = 0.001$) and VISUAL($\eta = 0.0003$)

Figure 10. Comparison of frame time.

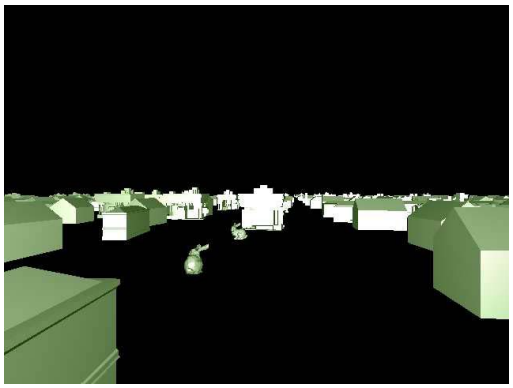
shown, with $\eta = 0.001$, the frame rate can be up to 20% faster than that with $\eta = 0.0003$. This is expected since a larger η implies coarser representations are retrieved. However, as we shall see next, the visual fidelity is not much compromised.

Figure 11 illustrates an example on the visual fidelity of the two systems. Comparing figure 11(a) and (b), it is clear that REVIEW misses some objects. These are objects that are more than 100m away from the query box. This confirms the problem of the spatial methods, i.e., the spatial methods may lose some visible models in the output. Looking at figure 11(c), it is clear that VISUAL not only provides better visual fidelity than REVIEW, but the loss in visual fidelity is not obvious. Comparing figure 11(c) and (a), we note that a threshold size of 0.001 can provide good visual fidelity.

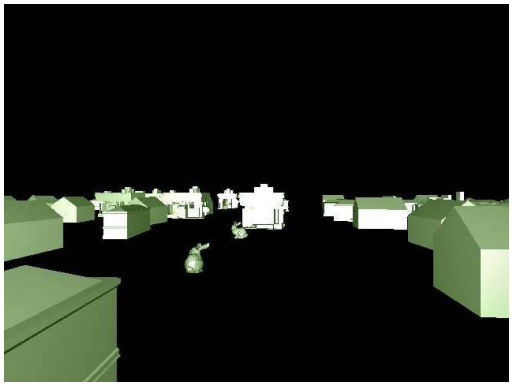
We recorded a few walkthrough sessions with different

Proceedings of the 19th International Conference on Data Engineering (ICDE'03)
1063-6382/03 $ 17.00 © 2003 IEEE

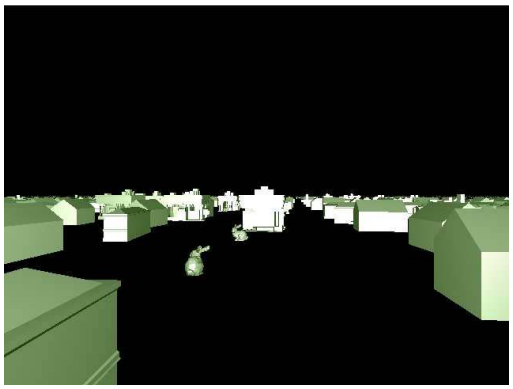
IEEE COMPUTER SOCIETY



(a) Original models

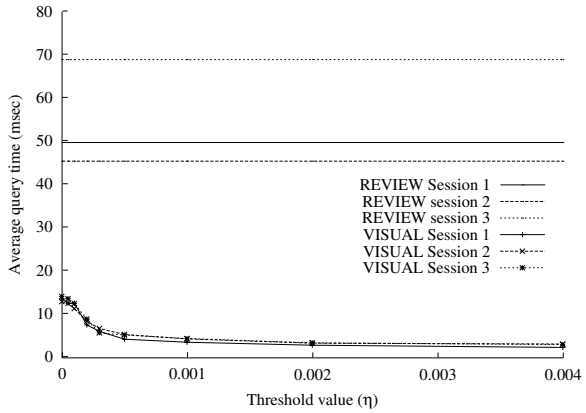


(b) REVIEW (size of query boxes 200m)

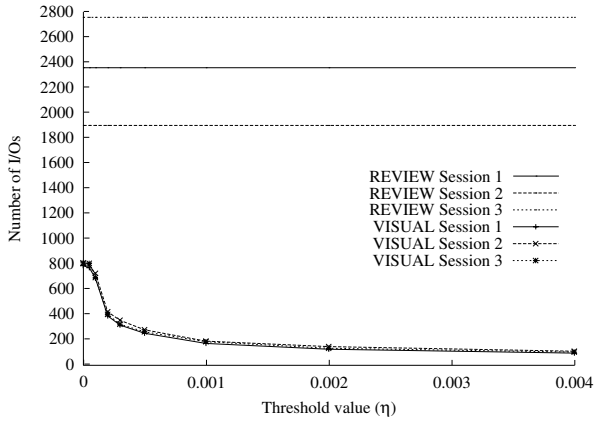


(c) VISUAL ($\eta = 0.001$)

Figure 11. 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.



(a) Average query time of different sessions



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

Figure 12. Search Performance in Different Walkthrough Sessions.

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 12(a) shows the average search time in each query for different walkthrough sessions. Figure 12(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 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

Proceedings of the 19th International Conference on Data Engineering (ICDE'03)
1063-6382/03 $ 17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

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 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 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.

6. Conclusion

In this paper, we have addressed the problem of optimizing performance and visual fidelity in visualization systems. We have proposed a novel structure called HDoV-tree that can be tuned to provide 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 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. In our current work, we have not exploited the MBR information in the HDoV-tree. As mentioned, this information can be exploited to prioritize objects to be loaded. We are currently exploring this as part of our agenda for future work.

References

[1] P. Agarwal, S. Har-Peled, and Y. Wang. Occlusion culling for fast walkthrough in urban areas. *Eurographics 2001* (Short Presentation), September 2001.

[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] J. Bittner, V. Havran, and P. Slavík. Hierarchical visibility culling with occlusion trees. In *Proc. Computer Graphics International (CGI'98)*, pages 207–219, Hannover, Germany, June 1998.

[5] 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.

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

[7] 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.

[8] 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.

[9] V. Koltun, Y. Chrysanthou, and D. Cohen-Or. Hardware assisted culling using a dual ray space. In *Proc. Eurographics Rendering Workshop*, pages 204–213, 2001.

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

[11] L. Shou. *Querying large virtual models for interactive walkthrough*. PhD thesis, National University of Singapore, 2002.

[12] L. Shou, C. Chionh, Y. Ruan, Z. Huang, and K. L. Tan. Walking through a very large virtual environment in real-time. In *Proc. 27th International Conference on VLDB (Very Large Data Bases)*, pages 401–410, Roma, Italy, 2001.

[13] S. J. Teller and C. H. Sequin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics (Proceedings of SIGGRAPH 91)*, 25(4):61–69, 1991.

IEEE COMPUTER SOCIETY