The Hierarchical Degree-of-Visibility Tree

Lidan Shou, Zhiyong Huang, *Member*, *IEEE*, and Kian-Lee Tan, *Member*, *IEEE Computer Society*

Abstract—In this paper, we present a novel structure called the Hierarchical Degree-of-Visibility Tree (HDoV-tree) for visibility query processing in visualization systems. The HDoV-tree builds on and extends the R-tree such that 1) the search space is pruned based on the *degree of visibility* of objects and 2) internal nodes store level-of-details (LoDs) that represent a collection of objects in a coarser form. We propose two tree traversal algorithms that balance performance and visual fidelity, explore three storage structures for the HDoV-tree, and develop novel caching techniques for disk-based HDoV-tree. We implemented the HDoV-tree in a prototype walkthrough system called VISUAL. Our experimental study shows that VISUAL can lead to high frame rates without compromising visual fidelity.

Index Terms—Degree of visibility, HDoV-tree, level-of-details, performance and visual fidelity, virtual environment.

1 Introduction

Interactive visualization systems are widely used in various applications in industry, academics, and entertainments. The models required are getting increasingly complex and may involve tens of thousands of objects each consisting of thousands of polygons. Moreover, each object typically has multiresolution representations called level-of-details (LoDs). Thus, conventional systems that assume models fit into the main memory are no longer affordable. This calls for novel techniques to handle models that are stored on disk.

In visualization systems, one of the most frequently used operations is the *viewpoint* query that 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. 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 amount of data representing objects to be loaded may be unnecessarily high. 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.

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*. In other words, at different viewpoint positions, the tree "captures" 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 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 two algorithms that traverse the HDoV-tree to balance visual fidelity and performance. We also propose three storage organizations of the HDoV-tree on secondary storage, and develop novel cache replacement policies to expedite the accesses to the nodes of the HDoV-tree. 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 provides excellent visual fidelity.

A preliminary version of this paper appears in [10]. There, we only present the disk-based version HDoV-tree. We have extended the paper to consider cache replacement policies, and proposed a novel performance guaranteed traversal scheme. We also report a more comprehensive performance study on the VISUAL system and look at how the HDoV tree performs in a memory-based system.

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 HDoV-tree structure and the traversal algorithms in Section 3, and several storage schemes in Section 4. In Section 5, we discuss two cache replacement policies that are based on the degree-of-visibility values. Section 6 reports the results of our experimental study and, finally, we conclude in Section 7.

2 Related Work

Most of the earlier works have assumed that virtual environments are memory resident. More recent works on managing large virtual environments [6], [9] used spatial

- *L. Shou is with the Handsome International Software Co. Ltd, 9F, Tower 1, Chang Di Torch Mansion, #259 Wen San Rd., Hangzhou, P.R. China 310012. E-mail: should@handsome.com.cn.*
- *Z. Huang and K.-L. Tan are with the Department of Computer Science, National University of Singapore, 3 Science Drive 2, Singapore 117543. E-mail: {huangzy, tankl}@comp.nus.edu.sg.*

Manuscript received 7 Apr. 2003; revised 3 Nov. 2003; accepted 10 Feb. 2004. For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0029-0403.

indexes to organize, manipulate, and retrieve the data. Kofler et al. proposed the LoD-R-tree [6] that combines the R-tree index with a hierarchy of multirepresentations 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). Thus, the structure leads to high frame rates as long as the user's viewing-frustum stays within the query region. However, its performance degenerates significantly as the user view changes.

In [9], 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 methods offer a neat solution to real-time visualization, they suffer from two problems. First, they 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 visible objects that are out of the query box will not be retrieved. Thus, the visual fidelity is poor. Second, they 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 [1], [2], [3], [4], [7], [8], [12]. However, computing visibility at runtime is expensive. Moreover, how the data sets and visibility data are managed at runtime has not been studied and reported.

3 The HDoV-Tree

In this paper, we adopt the (cell, list-of-objects)-based method to manage visibility in large virtual environments, i.e., the viewpoint space is partitioned into disjoint cells, and each cell has its associated list of visible objects. We present the proposed HDoV tree structure and the two proposed traversal algorithms in this 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 recognize an object as *visible* or *invisible*. We observe that such a Boolean representation is too "conservative" as it marks an object as visible even if only a very small part of it can be seen.

We introduce *degree-of-visibility* (DoV) to measure visibility more precisely. First, 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 [8]. 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:

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

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

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

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, the 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 visible (with $\mathrm{DoV} > 0$) should be retrieved. Intuitively, the larger the DoV value of an object, the more likely it will be noticed and so it is more critical for it to be shown in greater detail. On the contrary, an object that has very small 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. 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 aggregated object can be computed by adding up all the DoVs of the objects to be aggregated. As an example, given a number of objects, A, B_1, B_2, ..., and B_k, if B_i $(i = 1, 2, \ldots, k)$ are aggregated into object A, then we have

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

For a viewing region (cell), the DoV of an object viewed from region R can be defined conservatively as

$$DoV(\mathrm{R}, X) = max(DoV(p, X)), \qquad \forall p \in \mathrm{R}.$$

In this paper, we use region-based DoV because it is valid for a longer period than point DoV when a viewpoint moves. It is only invalid if the viewpoint enters another region.

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

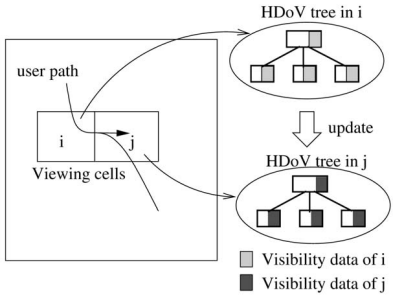


Fig. 1. Dynamic instantiation of HDoV-tree. (As the user viewpoint moves from cell *i* to cell *j*, the HDoV-tree is reinstantiated.)

spatial data structure essentially captures the spatial distribution of the objects in the virtual environment. However, there are several features that distinguish the HDoV-tree 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. Fig. 1 illustrates this.

For simplicity as well as because we are dealing with 3D objects only, we employ the R-tree [5] as the spatial structure in our implementation. Fig. 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 greater than zero. We note that 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 (see (1)). Moreover, if node N is visible, at least one child node (or object) of N is also visible.

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*, i.e., for different viewing region, 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.

We note that the HDoV-tree has several 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 greater detail, while those that are smaller can be represented by coarser LoDs. Second, we can potentially terminate the search at internal nodes if the aggregated DoV value of a node is small, which follows the same logic as the first. 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 further away to the view frustum can be delayed. This can further improve the search performance significantly.

3.3 The Search Algorithms

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.

To provide a balance between performance and visual fidelity, we use a threshold DoV value η to control the granularity (or LoDs) of objects to be loaded. Essentially, objects (or object groups) with DoV values smaller than η can be retrieved with relatively low detail (internal/coarse LoD); otherwise, a finer LoD should be considered. We note that for larger η values, the restriction on the visual quality will be looser, and lower details are allowed. As such, fewer

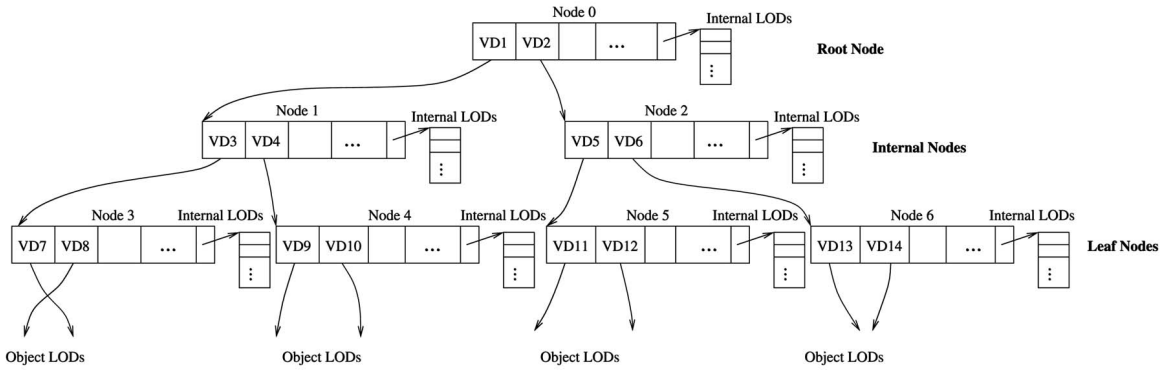


Fig. 2. A hierarchical degree-of-visibility tree.

```
Algorithm ThresholdSearch (Node)
1. For each entry E in Node
2.     if (E.DoV equals to 0) return;
3.     if (E is leaf)
4.         Add E.ptr→LoD_leaf into result;
5.     else
6.         if (E.DoV ≤ η and h(1 + log_M s) < log_M(E.NVO))
7.             Add E.ptr→LoD_internal into result;
8.         else
9.             ThresholdSearch(E.ptr);
```

Fig. 3. The threshold-based traversal algorithm.

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. Here, we shall present two traversal algorithms that are based on this.

3.3.1 Threshold-Based Traversal Algorithm

The first proposed traversal algorithm adopts the basic idea of pruning the search space using a threshold DoV value. We refer to this scheme as the threshold-based traversal algorithm. The scheme essentially controls the maximum allowable screen areas which could be replaced by lower internal LoDs, while achieving fast rendering speed.

Fig. 3 shows the algorithmic description of the algorithm. Given a query point and an instantiated HDoV-tree, the traversal starts from the root node (line 1). For objects/nodes that are completely hidden, i.e., whose entries have DoV = 0, the entire branches pointed to by these entries are 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 3-4), and the recursion will also terminate. For an internal node, the traversal algorithm will decide whether to proceed to the child node based on the DoV value (line 6). 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 7). We will discuss the second condition shortly. For entries with DoV values greater than η, we proceed to search their child nodes (line 9).

One issue with the above method is that for a node with small DoV, its LoD may contain more polygons than the sum of its visible descendants. Thus, terminating at this node may incur higher retrieval cost. To solve this problem, we can store the number of visible objects (NVO) in each VD entry. So, VD has two view-variant fields, i.e., $VD = (DoV, NVO)$. 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 7 of the traversal algorithm. 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 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 $m \cdot f \cdot s^h < f \cdot n$, which implies $h \cdot (1 + log_M s) < log_M n$. LoD of an active internal node can be selected as

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

where LoD_{lowest} and $LoD_{highest}$ are integer values denoting the lowest and highest LoD of an (aggregated) object at a level, respectively, and $0 < \frac{DoV}{\eta} \leq 1$ (because of the second condition at line 7). LoD of an active leaf node, meanwhile, can be selected as

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

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 $MAXDOV = 0.5$.

3.3.2 Polygon Budget Traversal Algorithm

While the threshold-based scheme can be used to control the visual fidelity, there is no performance guarantee for this algorithm. If the threshold specified by user is too small, the system may retrieve too many polygons, 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 the *Polygon Budget* (PB) traversal strategy that aims at performance-guaranteed walkthrough.

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.

With HDoV-tree, 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 child nodes 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 child nodes, $c_1, \cdots, c_m$, with descending DoV values, $d_1 \geq \cdots \geq d_m$. The budget allocated to child c_i with DoV value of d_i is

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

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, the refinement will proceed to other nodes with DoV values in descending order. If all the attempts fail, the recursive refinement process 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 that are visually more important are allocated more polygon budgets. It also sets relatively higher LoDs to 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 refinement algorithm used in the PB strategy is listed in Fig. 4.

```
Algorithm Refine (Node)
1.    LOD = ComputeLodBudget(DoV(Node), PolygonBudget);
2.    if (Node is leaf)
3.       if (LOD.NumPolygon < PolygonBudget)
4.          Node.SetLOD(LOD);
5.          PolygonBudget = PolygonBudget - LOD.NumPolygon;
6.       else
7.          Node.SetLOD(NULL); /* Drop the node */
8.    else
9.       Success=TRUE;
10.      For (each Child from MaxDov to MinDoV)
11.         if ( Refine(Child) < 0) Success= FALSE;
12.      if (NOT Success)
13.         if (LOD.NumPolygon < PolygonBudget)
14.            Node.SetLOD(LOD);
15.            PolygonBudget = PolygonBudget - LOD.NumPolygon;
16.            Attempts = 0;
17.            return TRUE;
18.         else if (Attempts > MAX_ATTEMPTS) /* Too many attempts */
19.            Node.SetLOD(LOD);
20.            return TRUE;
21.         else
22.            Node.SetLOD(NULL); /* Drop the node */
23.            Attempts = Attempts + 1;
24.            return FALSE;
25.      else
26.         return TRUE;
```

Fig. 4. The refinement algorithm in polygon budget traversal algorithm.

4 Storage Schemes for HDoV-Tree

Recall that the HDoV-tree is essentially a view-variant structure: Depending on the user's viewing region of the current viewpoint, the visibility data of the tree may be different. In this section, we examine three storage schemes for the HDoV-tree that capture the information for all cells, so that if the viewpoint is in cell i, then the content of cell i is accessed.

4.1 The Horizontal Storage Scheme

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. Fig. 5 shows the data structure of the scheme, which we call a *horizontal scheme*. In this scheme, the DoV values of a node N respective to cell C are stored in a fixed-size page, called the *V-page* (denoted as $VPage_{C,N}$ in Fig. 5). 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—it reserves the storage space of a V-page even if the node and objects are not visible in the cell at all.

4.2 The Vertical Storage Scheme

Another scheme, which requires less storage space, is the *vertical scheme*. As Fig. 6 shows, this scheme deploys an intermediate index structure, named *V-page-index*, between the nodes and the V-pages. Let N_{node} denote the number of nodes in the HDoV-tree. The V-page-index is segmented by the cells so that each segment contains as many as N_{node}

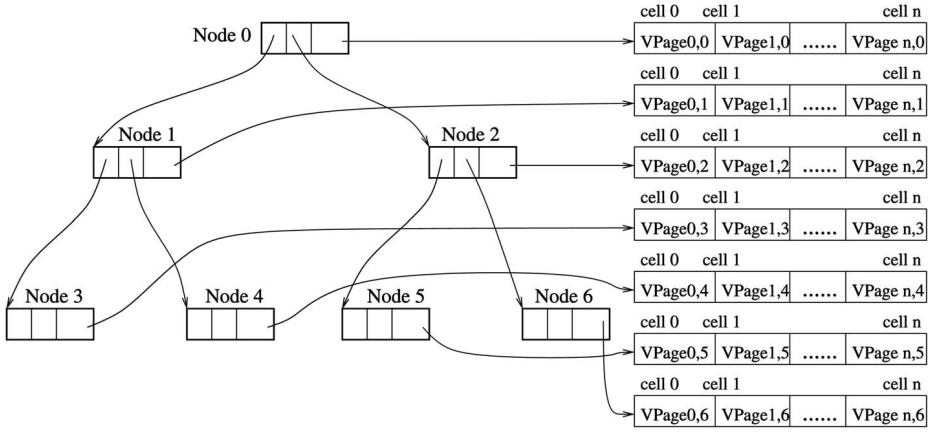


Fig. 5. Horizontal Storage Scheme. *VPage i,j* represents the V-page of node *j* in cell *i*.

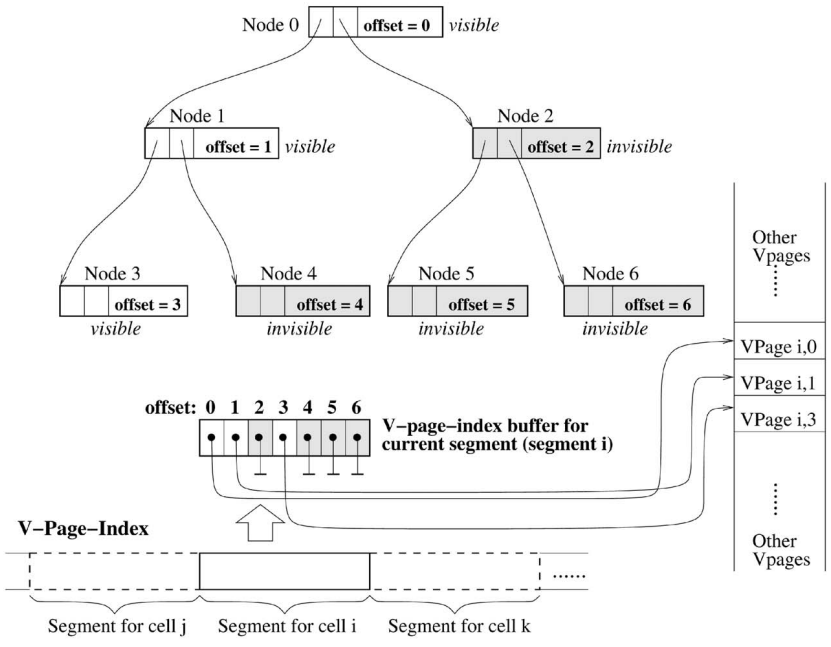


Fig. 6. Vertical storage scheme.

pointers. Each of the pointers, which are called *V-page pointers*, points to a V-page or to nil. Each tree node 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 offset value is used to locate the V-page pointer in the V-page-index. 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.

4.3 The Indexed-Vertical Storage Scheme

In the Vertical Storage scheme, "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 Fig. 7 shows (we have omitted the tree nodes as they are the same as those in Fig. 6). 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 (from O(number of nodes) in the horizontal scheme to O(number of visible nodes) [11]). Note that the segments stored in the V-page-index file may have variable lengths. This scheme is named *indexed vertical scheme*.

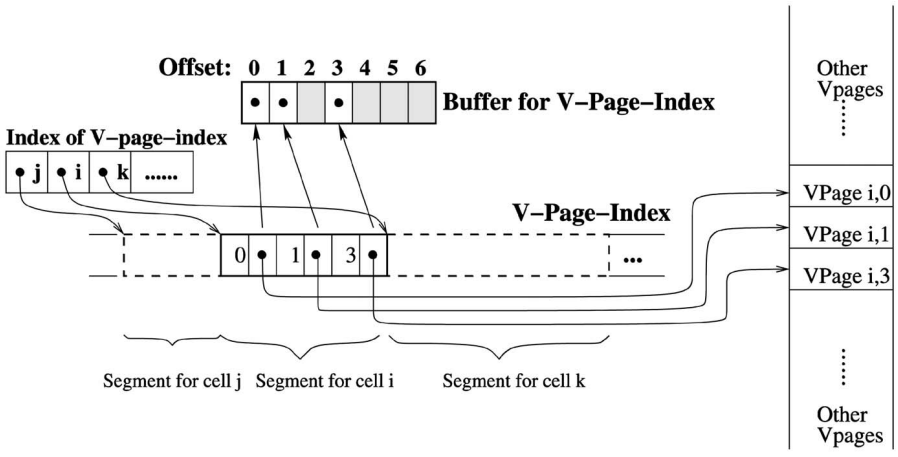


Fig. 7. Indexed vertical storage scheme.

5 Caching the HDoV-Tree Nodes

When the search threshold η is small, many of the nodes in the HDoV-tree need to be accessed. As such, 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. In this section, we propose two schemes that employ the DoV value for cache replacement.

The basic idea of the first scheme, DoV cache replacement policy, is to retain the data of visually important nodes in memory as long as possible. This is based on the assumption that the data of visually important nodes are likely to remain visible in the near future.

With the DoV cache replacement policy, the system keeps a log on the DoV values for all the nodes cached in the memory buffer. All the entries, which are $(nodeID, nodePtr)$ pairs, are stored in a *hash map*, where the key is the $nodeID$. A map list structure, which contains $(DoV, nodeID)$ pairs, is maintained in memory, and is sorted by ascending DoV values. Whenever there is a request for a node, 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 is 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 data of visually more important nodes in the memory buffer. As a comparison, for the Least-Recently-Used (LRU) replacement policy, 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.

While the DoV cache replacement policy may be efficient in exploiting the *visual coherence* that exists between consecutive visibility queries, the data of some of the nodes with 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," in the second scheme, the DoV value stored in the cache is adjusted by a function of 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. We shall refer to this variant of the DoV cache replacement policy as the *DoV-Time (DT)* policy. In our experiments, we set κ to 0.001.

6 Performance Study

We have implemented the HDoV-tree as a component of a prototype visualization system, VISUAL. VISUAL is a virtual reality walkthrough system implemented on a Pentium 4 PC running RedHat 7.2 that also facilitates visibility queries on specific viewpoint. In this section, we present representative results of an experimental study to evaluate the performance of the HDoV-tree. Readers are referred to [11] for the complete set of results including a study on the storage overhead of the three storage schemes.

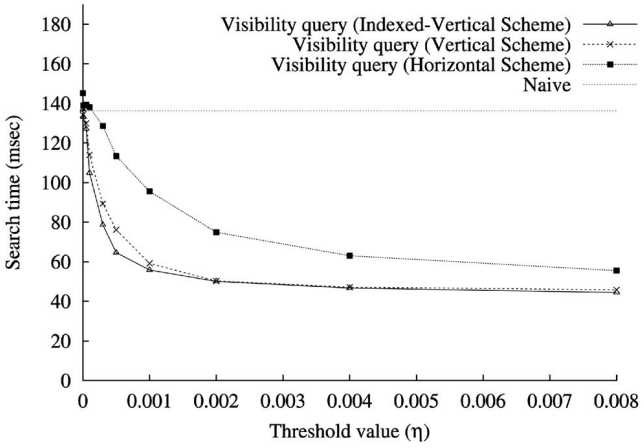


Fig. 8. Search time with different η values.

6.1 Experiment 1: Disk-Based System

In the first set of experiments, we study VISUAL for large data sets that do not fit into the main memory. The data sets we used are synthetic city models containing numerous buildings and bunny models. The raw data sets excluding the visibility data vary in sizes from 400 MB to 1.6 GB. The default data set used has a raw size of approximately 1 GB. The precomputation phase for the largest data set takes about 1.02 seconds to compute the DoV values in one cell. The time to reinstantiate the HDoV-tree in each cell will be accounted for in the runtime searching. For this set of experiments, we only evaluate the threshold-based search algorithm since the polygon-budget scheme is expected to access more nodes.

6.1.1 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 (see Section 6.1.2 for a comparative study).

We tested 10,000 visibility queries at random viewpoint positions obtained from the precomputed cells. Fig. 8 shows the results of the search time as η (DoV threshold) varies from 0 to 0.008. 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 coarser internal LoDs are allowed in the result set. Since the coarser 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

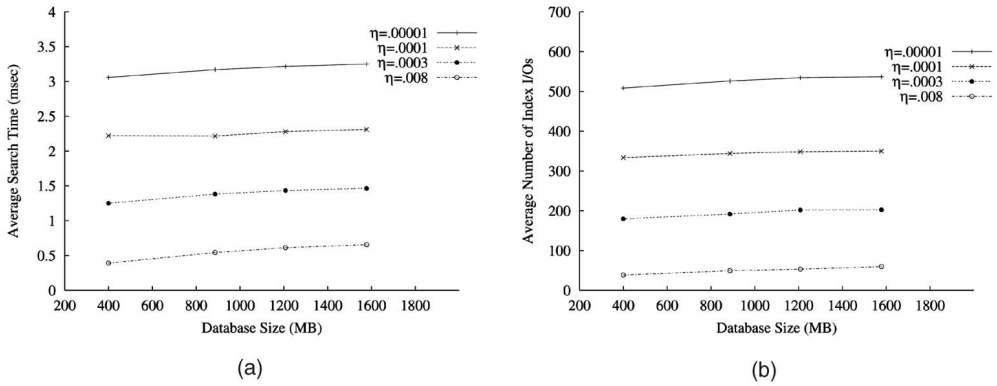


Fig. 9. Scalability of the visibility query (effect of data set size). (a) Average search time and (b) I/O cost.

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 marginally better 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, for the remaining experiments, we shall present the results for the indexed-vertical scheme only.

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 1,000 random viewpoints as the experimental query set, and performed the same 1,000 visibility queries on the data sets. The average search time and the average number of I/Os are shown in Figs. 9a and 9b, 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 increase only marginally with increasing data set sizes. The I/O cost only increases in very small amounts as the database size increases. The increase in search time is almost negligible. We also note that the variance of search time (in ms) changes little for different data sets, as shown in Table 1. These results demonstrate the scalability of the proposed scheme.

6.1.2 On Interactive Walkthrough

In this experiment, we evaluate the HDoV-tree's performance in interactive walkthrough applications. For a continuously moving viewpoint, there is often some spatiotemporal coherence to be exploited, i.e., two neighboring cells often share a number of visible objects. The search algorithm is extended to a "delta" search algorithm that does not retrieve objects accessed in the previous queries. We evaluated our walkthrough system, VISUAL, that implements HDoV-tree, against the REVIEW system [9]. Recall 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.

TABLE 1
Variance of the Visibility Query

Database Size (MB)	η=.00001	η=.0001	η=.0003	η=.008
400	0.489	0.503	0.458	0.490
887	0.521	0.480	0.534	0.532
1208	0.534	0.526	0.526	0.514
1578	0.521	0.569	0.531	0.588

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.

Fig. 10a shows the results of time spent on rendering each 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, as shown in the figure, the rendering frame time is very different. 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. In addition, REVIEW may retrieve objects that are within the query box but not visible to the viewpoint, wasting the I/O resources. On the contrary, the user of the VISUAL system can experience smoother walkthrough.

Fig. 10b 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 percent faster than that with $\eta = 0.0003$. This is expected since a larger η implies coarser representations are retrieved. However, as shown in Fig. 11, the visual fidelity is not much compromised. Comparing Figs. 11a and 11b, it is clear that REVIEW misses some objects. These are objects that are more than 100m away from the query box. Looking at Fig. 11c, it is clear that VISUAL not only provides better visual fidelity than REVIEW, but the loss in visual fidelity is not obvious. Comparing Figs. 11c and 11a, we note that a threshold size of 0.001 can provide good visual fidelity.

To measure the visual fidelity quantitatively, we can perform image segmentation on the images being rendered and compare the number of objects with that in the ground truth images, taking the visibility importance into con-

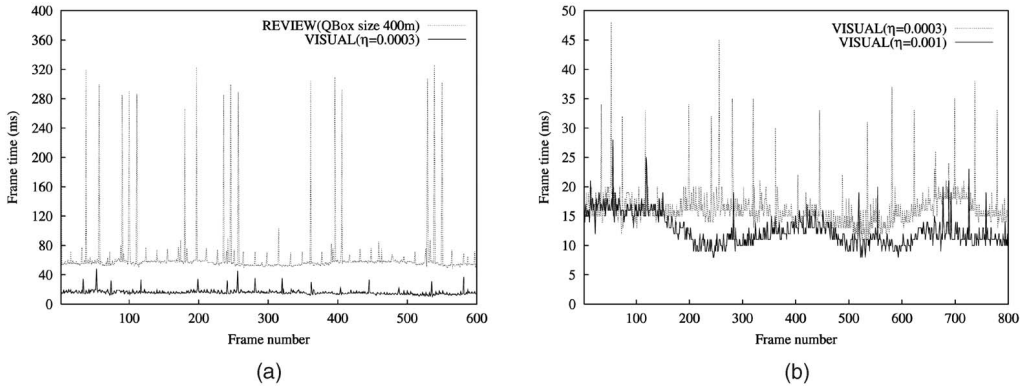


Fig. 10. Comparison of frame time. (a) VISUAL ($\eta = 0.001$) versus REVIEW and (b) VISUAL ($\eta = 0.001$) versus VISUAL ($\eta = 0.0003$).

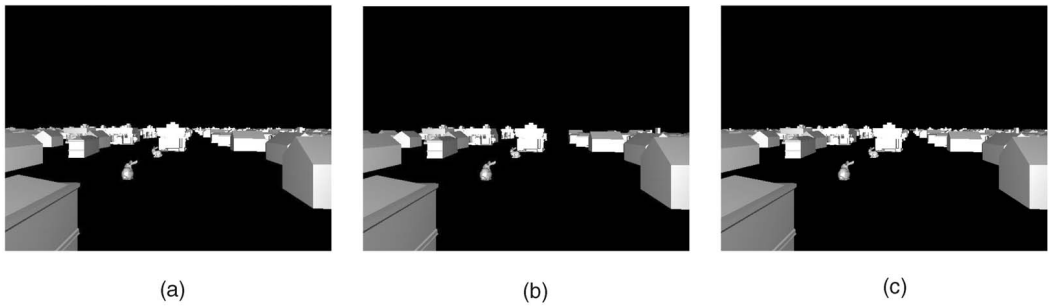


Fig. 11. Comparison of visual fidelity. (a) Original models, (b) REVIEW, and (c) VISUAL ($\eta = 0.001$).

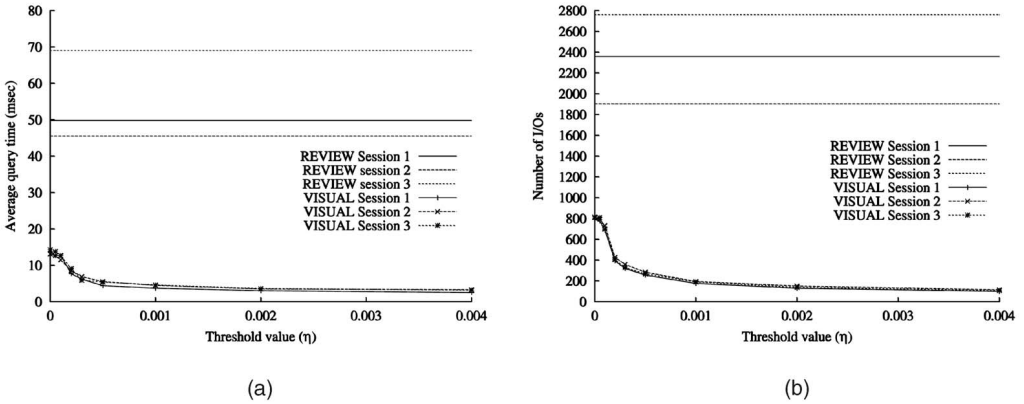


Fig. 12. Search performance in different walkthrough sessions. (a) Average query time of different sessions and (b) average number of I/Os of different sessions.

sideration as well. With such an approach, we define the *fidelity factor*, Fid, as follows:

$$Fid = \frac{1}{\sum_{i=1} w_i \cdot (N^i_{original} - N^i_{rendered})},$$

where $N^i_{original}$ and $N^i_{rendered}$ denote the number of objects in segment i of the ground truth image and the rendered image, respectively, and w_i is the weight to reflect the importance of segment i to the user. Clearly, $Fid \in (0, \infty)$, and the larger the Fid, the better the fidelity of the rendered image (compared to the ground truth image). We have segmented the sample scene in Fig. 11, and computed the Fid using a weight of 0.2 for the center of the image and 0.1 otherwise. As expected, the VISUAL system ($Fid = 0.909$) has a larger Fid value than the REVIEW system ($Fid = 0.192$).

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. Fig. 12a shows the average search time in each query for different walkthrough sessions. Fig. 12b 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 2 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.

TABLE 2
Results of Frame Time

	η	Avg Frame Time(ms)	Variance of Frame Time
VISUAL	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

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.

6.1.3 On Caching

In this experiment, we study the performance of the two proposed replacement policies. We use the Least-Recently-Used (LRU) policy as a reference. We note that the size of a node of the HDoV-tree is relatively small compared to the heavy-weighted object data. Therefore, the results show only the cost to traverse the HDoV-tree, and excludes the cost to retrieve the objects.

We first evaluated the cache performance for spatially continuous viewing cells. This is the case of a walkthrough environment, as the cells can be cascaded into a chain of the cells along the walking path. Fig. 13 shows the cache hit rates in 4,000 queries for the three cache replacement policies with various threshold values ($\eta =$ 0.00001, 0.0001, 0.0005). These threshold values are small enough for the traversal algorithm to access a large number of nodes in the HDoV-tree. For large threshold values, too few nodes are being accessed and the three schemes perform equally well. We observe that 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 percent. 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 scheme when the cache size is relatively large. Hence, the DT cache replacement policy is the best 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 occupies more than 7,500 pages.

We also studied how the replacement schemes perform for visibility queries. Fig. 14 shows the results of the cache hit rates for 4,000 random visibility queries in the viewing regions. The threshold values are the same as those that are used in the previous experiment.

As expected, the hit rates of the random queries are slightly less than those in the previous experiment. But, they display similar patterns with regard to the cache size. The DT cache replacement policy also performs slightly better than the DoV policy. Like the previous experiment, the LRU policy also performs at either of its two

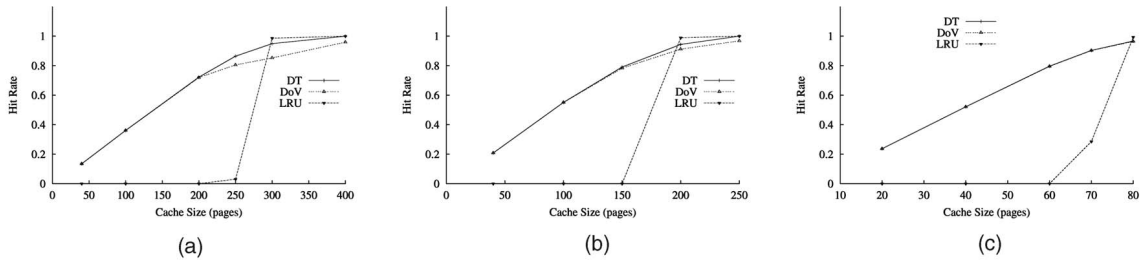


Fig. 13. Cache hit rate for *Continuous Queries*. (a) $\eta = 0.0001$, (b) $\eta = 0.0001$, and (c) $\eta = 0.0005$.

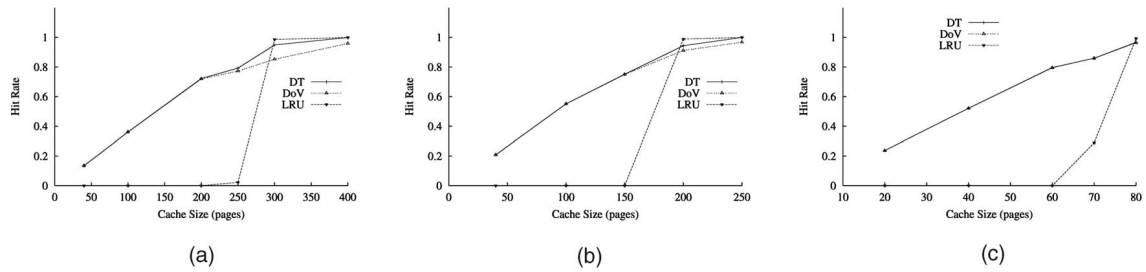


Fig. 14. Cache hit rate for *Random Queries*. (a) $\eta = 0.0001$, (b) $\eta = 0.0001$, and (c) $\eta = 0.0005$.

TABLE 3
Runtime Performance of the Same Walkthrough Path by Different DoV Thresholds

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

extremes—being 0 or very close to 100 percent. When the cache size is relatively large, the curves of the DT policy is very close to those of the LRU scheme.

6.2 Experiment 2: On Memory-Based HDoV-tree

In this section, we evaluate the performance of the HDoV-tree when the data sets fit in the memory. The data set used to evaluate the memory-based HDoV-tree is very similar to those used in the disk-based experiments, and contains 4 million polygons. This allows us to see how HDoV-tree will perform on existing virtual environments that typically employ simpler models that fit in the memory. We evaluate the memory-based HDoV-tree using the average frame time and number of polygons being rendered. For comparison, we use the naive (cell, list of objects) scheme (denoted $CONV$). Our CONV scheme uses the same spatial subdivision as the HDoV structure does. We also denote the threshold-based search scheme running with a DoV threshold of η as $VSC(\eta)$, and the polygon budget rendering scheme running with a polygon budget of n as $PB(n)$.

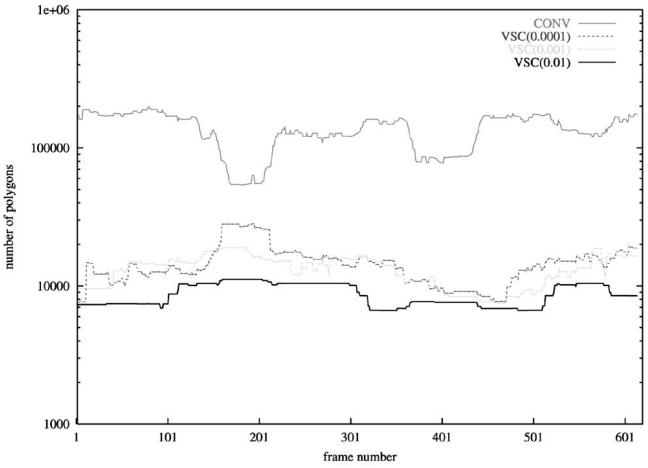


Fig. 15. Number of polygons rendered in each frame.

We first compare $VSC(\eta)$ with different η values to $CONV$. We run the same walkthrough sessions as that used in the disk-based experiments 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 3. As the table shows, the average frame time of VSC is much smaller than that of CONV. The number of polygons rendered and nodes accessed are also much smaller compared to CONV. As η increases, the frame time becomes shorter, and vice versa. This is consistent with our goal in designing the VSC algorithm.

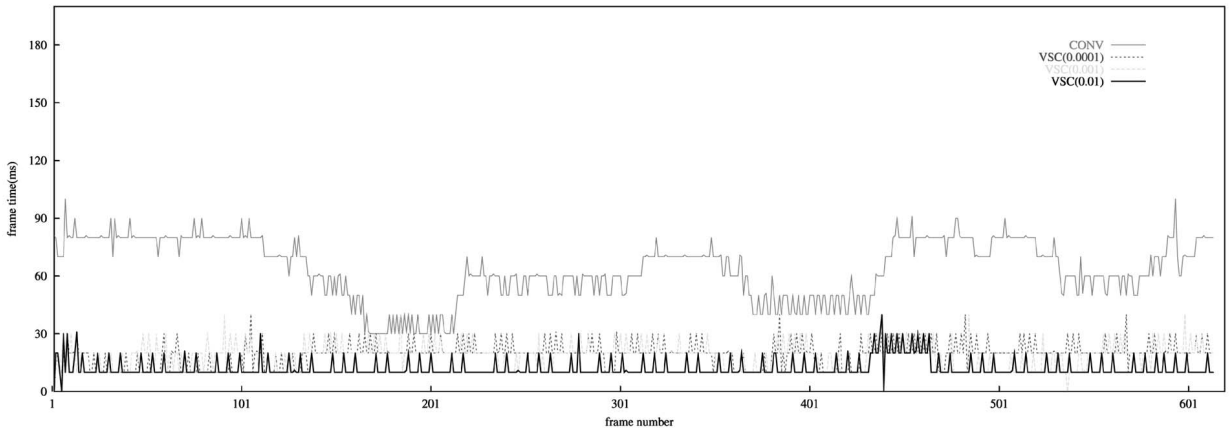
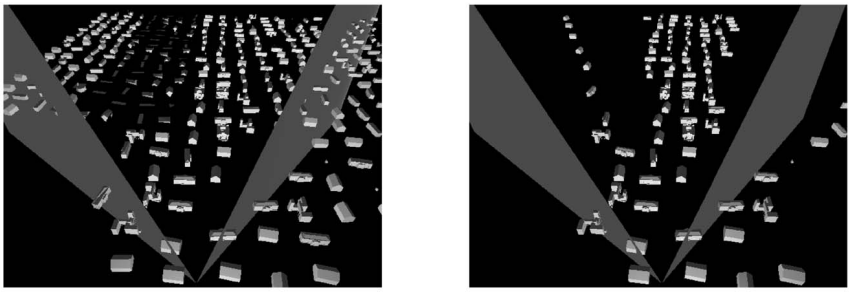


Fig. 16. Frame time of the same walkthrough (VSC mode versus CONV mode).



(a) (b)

Fig. 17. Snapshots of VSC and CONV schemes. (a) Bird's-eye view (VSC 0.001) and (b) bird's-eye view (CONV).

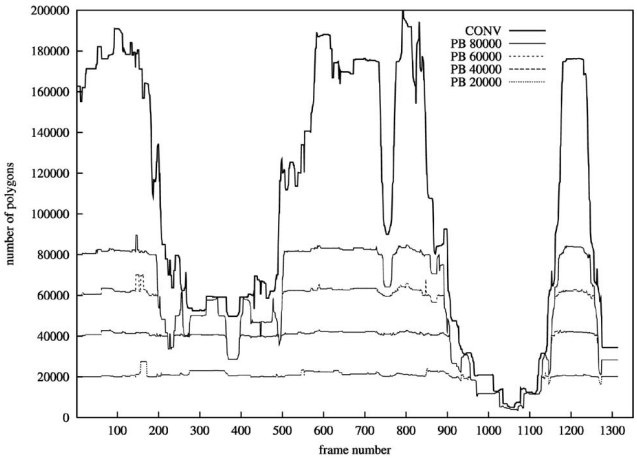


Fig. 18. Number of polygons rendered in Polygon Budget Mode.

Fig. 15 shows the number of polygons rendered in each frame. The VSC scheme greatly reduces the number of polygons being rendered. This is because the use of coarse internal LoDs reduces the number of nodes accessed. Fig. 16 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 Fig. 15.

Fig. 17 shows the bird's eye view of a snapshot of the walkthrough. Note the very low details of the internal nodes in Fig. 17a. Although Fig. 17a appears to contain more objects than Fig. 17b, there are actually fewer geometries. We also observe that when the threshold becomes larger, the visual quality will degrade, as LoDs at higher level nodes will be retrieved. A threshold below 0.01 can achieve fairly good visual quality while keeping high frame rates. For VSC(0.001), the system can gain a speedup of 3.5 times as compared to the CONV.

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.

Fig. 18 shows the number of polygons rendered under various polygon budgets. As shown, 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 Fig. 18, 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.

Fig. 19 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

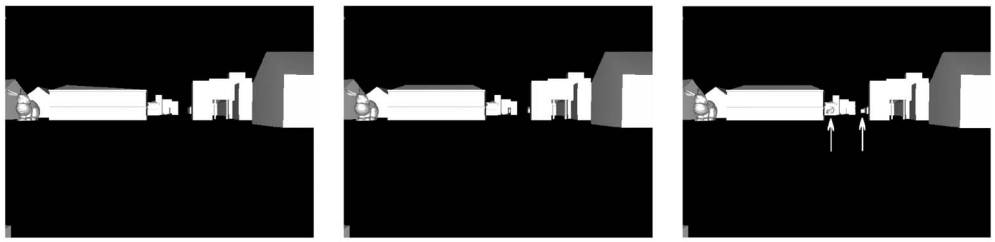


Fig. 19. Snapshots of PB modes. Note the low detail of the building on the left in (a), the missing bunny and the missing building in the center of the screen in (a) and (b). (a) PB 20000, (b) PB 40000, and (c) PB 80000.

TABLE 4
Comparison of VSC and PB Algorithms

	The DoV threshold method (VSC)	The polygon budget method (PB)
Objective	The threshold controls the fidelity directly: A smaller (larger) threshold will achieve a higher (lower) fidelity.	The budget controls the frame time directly: A more (less) powerful computer will have bigger (smaller) budget value to achieve the same frame time.
Relationship between the two methods	A smaller (larger) threshold will lead to a larger (smaller) frame time.	A smaller (larger) budget will lead to a lower (higher) fidelity.
User's point of view	The threshold value needs to be tuned for a specified data set to compromise between the visual fidelity and the frame time.	The system (CPU, caches, memory, graphics card, hard disk, etc.) determines the polygon budget of the real time frame rate (24 frame per second equivalent to about 0.04 second frame time).

system obtains a speedup of 2.7 times as compared to the CONV walkthrough.

To summarize, 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 occupy. 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. Table 4 compares the two approaches at a high level.

7 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 performance and visual fidelity. The HDoV-tree is essentially an R-tree that contains visibility data and LoDs. We also examined two novel search algorithms and proposed three storage structures for the HDoV-tree. We have discussed two caching replacement policies that are based on the DoV values. We have implemented the HDoV tree in a prototype walkthrough system called VISUAL, and conducted extensive experiments to evaluate its performance. Our results show that HDoV-tree can provide excellent visual fidelity efficiently.

References

[1] P.K. Agarwal, S. Har-Peled, and Y. Wang, "Occlusion Culling for Fast Walkthrough in Urban Areas," *Proc. Eurographics 2001 (short presentation)*, Sept. 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," *Proc. ACM Symp. Interactive 3D Graphics*, pp. 199-206, 1999.

[3] J. Bittner, V. Havran, and P. Slavík, "Hierarchical Visibility Culling with Occlusion Trees," *Proc. Computer Graphics Int'l Conf. (CGI '98)*, pp. 207-219, June 1998.

[4] T.A. Funkhouser, C.H. Sequin, and S.J. Teller, "Management of Large Amounts of Data in Interactive Building Walkthroughs," *Proc ACM SIGGRAPH Symp. Interactive 3D Graphics*, pp. 11-20, Mar. 1992.

[5] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. 1984 ACM SIGMOD Int'l Conf. Management of Data*, pp. 47-57, 1984.

[6] M. Kofler, M. Gervautz, and M. Gruber, "R-Trees for Organizing and Visualizing 3D GIS Databases," *J. Visualization and Computer Animation*, vol. 11, pp. 129-143, 2000.

[7] V. Koltun, Y. Chrysanthou, and D. Cohen-Or, "Hardware Assisted Culling Using a Dual Ray Space," *Proc. Eurographics Rendering Workshop*, pp. 204-213, 2001.

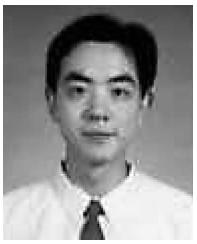
[8] C. Saona-Vázquez, I. Navazo, and P. Brunet, "The Visibility Octree: A Data Structure for 3D Navigation," *Computers & Graphics*, vol. 23, pp. 635-643, 1999.

[9] L. Shou, C. Chionh, Y. Ruan, Z. Huang, and K.L. Tan, "Walking through a Very Large Virtual Environment in Real-Time," *Proc. 27th Int'l Conf. Very Large Data Bases*, pp. 401-410, 2001.

[10] L. Shou, Z. Huang, and K.L. Tan, "Hdov-Tree: The Structure, the Storage, the Speed," *Proc. 19th Int'l Conf. Data Eng.*, 2003.

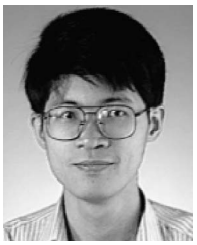
[11] L. Shou, "Querying Large Virtual Models for Interactive Walkthrough," PhD thesis, Nat'l Univ. Singapore, 2002.

[12] S.J. Teller and C.H. Sequin, "Visibility Preprocessing for Interactive Walkthroughs," *Computer Graphics (Proc. SIGGRAPH '91)*, vol. 25, no. 4, pp. 61-69, 1991.



Lidan Shou received the PhD degree from the School of Computing, National University of Singapore, in 2002. He is now working as a senior software architect at the Handsome Electronics Corporation, China. His research interests cover spatial database, computer graphics, virtual reality, and real-time distributed systems.

Zhiyong Huang received the BEng and MEng degrees in computer science and engineering from Tsinghua University of Beijing, China, in 1986 and 1988, respectively. He received the PhD degree in computer science from EPFL, Switzerland, in 1997. He is currently an assistant professor in the Department of Computer Science, National University of Singapore. His research interests include visualization, computer graphics, and multimedia databases. He is a member of the IEEE and ACM SIGGRAPH Singapore Chapter.



Kian-Lee Tan received the BSc (Hons) and PhD degrees in computer science from the National University of Singapore, in 1989 and 1994, respectively. He is currently an associate professor in the Department of Computer Science, National University of Singapore. His major research interests include query processing and optimization, database security, and database performance. He has published more than 100 conference/journal papers in international conferences and journals. He has also coauthored three books. He is a member of the ACM and the IEEE Computer Society.