BALANCING FIDELITY AND PERFORMANCE IN VIRTUAL WALKTHROUGH

Yixin Ruan, Jason Chionh, Zhiyong Huang, Lidan Shou, Kian-Lee Tan
Department of Computer Science, National University of Singapore, Singapore 117543
[ruanyixi, jchionh, huangzy, shoulida, tankl]@comp.nus.edu.sg

Abstract In a virtual reality (VR) system, users expect both high visual quality as well as a high constant frame rate when they interact with the system (walkthrough). However, realizing these two requirements for very large virtual environments (VE) that cannot fit in the main memory has not been adequately addressed in the literature. In this paper, we present a novel access method, called ViSA (Visibility and Spatial data Access method) that, given a region of the current view point, returns the set of 3D objects of VEs that are visible from, as well as those that are in, the region. Ideally, these 3D objects will be loaded into memory and rendered during walkthrough. However, to balance the visual fidelity (showing all visible objects) and constant high frame rate, we dynamically determine the number and set of objects that are to be loaded based on the current memory size and frame rate. In order to minimize I/O cost, we propose an optimized search technique. We have implemented the ViSA structure in a prototype walkthrough system and our experiments show that it can provide quality visual fidelity with an acceptable constant real time frame rate.

Keywords: Visual Fidelity, Visibility and Spatial Data, Virtual Environment, Secondary Storage Indexing

1. INTRODUCTION

Traditionally, in walkthrough applications, the Virtual Environment (VE) is stored in its entirety in the main memory. However, this assumption is no longer reasonable. For a realistic VE, secondary storage must be used. However, relying on the OS's virtual memory management for walkthrough gives unacceptable results because the OS is application independent.

One promising strategy that has been adopted in the literature is to exploit spatial index structures, e.g., R-trees (Guttman, 1984), for VE data (Pajarola et al., 1998; Shou et al., 2001). While these works can achieve high frame rates for real-time walkthrough, they have not adequately addressed the issue of visual fidelity. This is because spatial index structures are designed to cluster objects that are spatially close together. As such, a window query may miss "distant" objects that are visible from the viewpoint (i.e., those that do not intersect the query box) resulting in unsatisfactory visual quality. Clearly, using a sufficiently large window size that bounds the region for the maximum viewing distance of human eye sight is certainly not a practical solution.

In this paper, we capture two categories of information from the VE data. First, the VE space is split into regions, and for each region, we keep track of its *visibility set* (i.e., objects that are visible from at least one point within the region). (We assume a static environment so that visibility information can be pre-computed.) Second, the VE objects are organized into clusters so that those that are spatially close are in the same cluster. The first set of information is used to enhance the visual quality of the scene, while the second category is used to facilitate high frame rates. We present a novel access method, called ViSA (*V*isibility and *S*patial data *A*ccess method), to facilitate speedy retrieval of these two types of information. It is essentially an extended R-tree where nodes contain information on the spatial and visibility information of their entries. Given a window query, ViSA can therefore restrict the search space to regions that are close to the query region. The additional information along the paths of the nodes traversed provide the (pre-computed) visibility set that should be accessed for these candidate regions.

While the set of objects obtained from ViSA can guarantee visual fidelity (showing all visible objects), the system may not be able to sustain a constant and high frame rate. We propose a novel scheme to balance these two potentially conflicting user requirements. For each region, we pre-compute the contribution of each visible object to the scene. These objects can then be sorted using the contribution values in a non-ascending order. During walkthrough, visible objects are loaded according to this sorted order (i.e., objects with higher contribution values are loaded first) until some thresholds are violated. Two important thresholds can be used – available memory and the frame rate, i.e., if the available memory or frame rate drops below a predetermined value, then no further loading is performed. In order to minimize I/O cost, we propose an optimized search technique.

We have implemented a prototype walkthrough system that employs the proposed mechanisms. Our experiments on a large dataset show that the proposed mechanisms can facilitate high and constant rendering frame rates and excellent visual fidelity.

The rest of this paper is organized as follows. In the next section, we review some related work. In Section 3, we present the ViSA framework in detail. In Section 4, we describe an optimized search technique. Section 5 presents our experimental study and report our findings, and finally, Section 6 concludes the paper.

2. RELATED WORK

Existing works have addressed the problem from two different perspectives – either from the visibility (Cohen-Or et al., 1998; Panne and Stewart, 1999; Yagel and Ray, 1995) or from the spatial proximity (Kofler et al., 2000; Pajarola et al., 1998; Shou et al., 2001) point of views. Detecting objects that are visible from at least one point in a 3D view cell is a non-linear 4D problem (Teller, 1992). Most of these works assume that the entirety of the VE can fit in the main memory.

A straightforward approach to indexing visibility information for large VEs is to decompose the VE space into regular grid cells, and maintain cell-to-cell visibility information. However, this method is shown to be inefficient (Shou et al., 2001). Works on spatial proximity have largely employed the R-tree to organize the data space to facilitate searching. In Figure 1, we illustrate a sample dataset with 8 objects (Minimum Bounding Rectangles (MBRs) are shown in Figure 1(a)), and the corresponding R-tree structure (Figure 1(b)). The works in (Pajarola et al., 1998; Shou et al., 2001) consider only the spatial proximity of objects, while (Kofler et al., 2000) adapted the R-tree structure to also store multiple level of details of an object. R-tree, however, is not adequate in addressing the issue of visual fidelity.

3. THE VISA STRUCTURE

In this section, we shall present the ViSA structure. After giving an overview of the structure, we shall present the basic search algorithm. Next, we shall examine how ViSA can be used to balance visual fidelity and constant high frame rate. We shall end this section by looking at how ViSA can be created.

3.1. The Big Picture

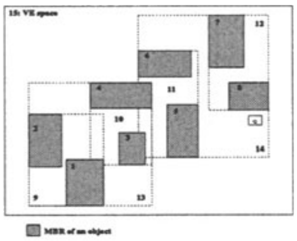
ViSA is a *V*isibility and *S*patial data *A*ccess method to facilitate speedy retrieval of visibility as well as spatial proximity information. For pedagogical reasons, we shall use the example in Figure 1(a) in our discussion. Figure 2(a) shows the cell-to-object visibility information of the MBRs of the objects. For example, objects 2, 3, 4 and 5 are visible from MBR 1. This means that if the viewpoint is in MBR 1, then (ideally) objects 2-5 should also be loaded to ensure high visual fidelity walkthrough. Figure 2(c) shows the ViSA structure for our example dataset and visibility information.

As can be seen from Figure 2(c), the core of ViSA is an R-tree structure that provides the spatial proximity information (compare the structure given by the box with the R-tree in Figure 1(b)). However, ViSA extends the R-tree in several ways. First, each leaf node contains entries of the form (MBR, OP, VP) where MBR is the minimum bounding rectangle of the object, OP is the pointer to the object, and VP (which is newly added) is a pointer to the list of objects visible from at least one position in MBR. As shown in Figure 2(c), MBR 1 points to objects 2-5.

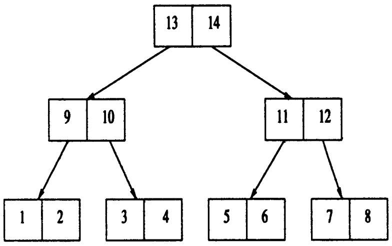
Each internal node contains entries of the form (MBR, NP, VP) where MBR is the minimum bounding rectangle that bounds all regions in its child node pointed by NP, and VP is a pointer (newly added) to the visibility information of MBR. Clearly, the visibility information of MBR is a superset of the visibility information of its children regions. To enhance performance, the visibility information of MBR is determined by excluding those of its children regions. However, since the region bounded by MBR can be very large, the visibility information can be very large. For example, consider MBR 14 in Figure 1(a), even if we have excluded MBRs 5, 6, 7 and 8, the region covered by MBR 14 is still large, and the visibility set of this space can be very large. Thus, if a query window, Q (see Figure 1(a)), intersects MBR 14, the entire visibility set corresponding to MBR 14 may have to be retrieved. We resolve this problem by splitting the region into smaller subregions. As an example, in Figure 2(b), the regions of MBRs 12 and 14 have been further decomposed into 4 and 7 subregions respectively. This decomposition is done such that each subregion contains approximately the same number of visible objects. Thus, VP points to a list of regions, each of which provides the visibility information for the region. Using the same query example where query window Q intersects MBR 14, we now only need to access visibility set corresponding to MBR 14.6 which is much smaller than MBR 14! The root node structure is similar to the internal node.

3.2. The Basic ViSA Search Algorithm

Figure 3 shows the algorithmic description of the basic ViSA search algorithm. The algorithm is very similar to the R-tree search algorithm, except for lines 5-7 and 12-13. Given a query window, the search begins from the root node, and traverses down the tree. For each internal node of ViSA examined (lines 1-7), the Bounding Boxes (BB) of the entries within the node are first checked against the query window (line 2). If the BB of an entry E overlaps the query window, it means that objects covered by the BB may be spatially close to or within the query window. As such, the search will continue with the child node associated with E (line 4). Next comes the differences from R-tree. Since ViSA internal nodes also keep track of visibility information of the space covered by the BB of E, the VP pointer is used to retrieve the list of regions not covered by E's children nodes. For those regions that intersect the query window, we return the pointer to the list of the visible objects (lines 5-7). In this way, all potentially visible objects along the path can be accessed.

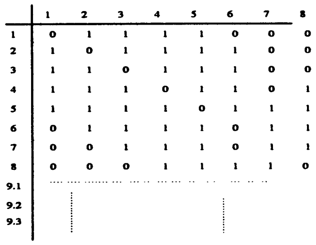


(a) A sample dataset.



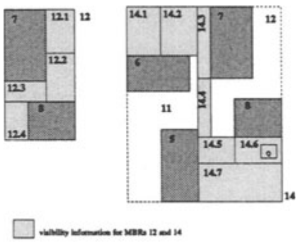
(b) The corresponding R-tree.

Figure 1. An R-tree example.

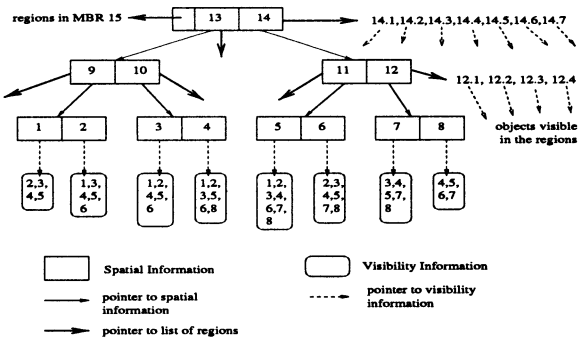


	1	2	3	4	5	6	7	8
1	0	1	1	1	1	0	0	0
2	1	0	1	1	1	1	0	0
3	1	1	0	1	1	1	0	0
4	1	1	1	0	1	1	0	1
5	1	1	1	1	0	1	1	1
6	0	1	1	1	1	0	1	1
7	0	0	1	1	1	0	1	1
8	0	0	0	1	1	1	1	0
9.1								
9.2								
9.3								

(a) Visibility information.



(b) Partitioning of MBR.



(c) The ViSA structure.

Figure 2. A ViSA example.

Now, for a leaf node, similar process is performed (lines 9-12). If the BB of an entry E of the leaf node overlaps the query window, then we know that E is a qualifying record. So, both the pointer to the object corresponding to E and the pointer to the list of objects visible from E are returned. Finally, we need to merge the query results of steps 7 and 12. The reason is that the same object can be queried more than once, e.g., one object can be visible from two different regions.

As in the R-tree search algorithm, those entries that do not intersect with the query window can be pruned away. Thus, ViSA facilitates speedy retrieval of spatial and visibility data. In order to minimize I/O cost, we propose an optimized search algorithm that will be described in Section 4.

Algorithm Search (T, C)
Input: T: pointer to node, C: Query window

```
1.   if T is not a leaf node /* search subtrees */
2.       for each entry E of node T do
3.          if (BB(E) Overlaps C)
4.             Invoke Search on the sub-tree whose root is the node
               associated with E
5.             for each region V obtained via pointer VP associated with E
6.                if (V Overlaps C)
7.                   Return VP of node T, the pointer to list of visible objects
8.   else /* search leaf node */
9.       for each entry E of node T do
10.         if (BB(E) Overlaps C)
11.            E of node T is a qualifying record
12.            Return the SP and VP of node T
13.  Merge the results of steps 7 and 12
```

Figure 3. The search algorithm.

3.3. Visual Fidelity vs High Frame Rate

Given a query window Q, we can obtain the set of objects that are in Q as well as objects that are visible from any point in Q. However, depending on the memory size and the computational power of the system, it may not always be possible to sustain high frame rates as well as visual fidelity. Under such a situation, our approach is to provide as much visible data as possible as long as the frame rate does not degrade beyond a certain predetermined threshold, i.e., when the frame rate falls below a certain prescribed threshold, we will sacrifice the visual quality by dropping some visible objects.

Essentially, given a region, we associate with each object in its visibility set a contribution value. An object's contribution value reflects its importance to the rendered image. In other words, it is more valuable to load an object with a larger contribution value than one with a lower contribution value. Thus, all we need is to sort the objects in non-ascending order of their contribution values. During a walkthrough session, visible objects with higher contribution values are loaded first, followed by those with smaller values. In this way, whenever the frame rate drops below the threshold value, we stop loading the visibility objects.

3.4. Creation of the ViSA Structure

To create a ViSA structure from a VE is a fairly complex and time consuming task. Fortunately, this process is done offline, and only need to be done once for a static VE. As described in the previous subsections, the major data structures of ViSA are (1) the hierarchical cells that represent the spatial adjacency of 3D objects (referred by SP) and (2) the visible set of each node and regions (referred by VP).

The creation is split into three steps. First, we construct the R-tree component of ViSA. This process is straightforward by using the R-tree construction scheme proposed in (Guttman, 1984). However, we need to allocate space for all the additional visibility pointers that are needed in ViSA. The process is essentially an iterative inserting process: for each object and its MBR in the scene, we traverse a single path down the tree from the root to the leaf and insert the entry into the leaf node. At each level of the traversal, we choose the child node whose MBR needs the least enlargement to contain the MBR of the object. In our implementation, we adapted the linear node split algorithm of (Ang and Tan, 1997). This algorithm minimizes both the coverage and the overlap of the cells of SP. It is very efficient with linear time complexity for the number of 3D objects in a VE.

Second, we determine the regions in which visibility set should be computed. We traverse the R-tree component bottom up, i.e., from the leaf levels first, followed by the level above the leaves and so on. At each leaf node, for each data object, its MBR corresponds to the region. For each internal node, the space that is not captured by the children nodes are split into regions. In our current work, we have adopted a simple strategy: we split the space into regions whose size must be smaller than the largest MBR of the children nodes.

Third, we compute the visible set for the region defined by each node in the hierarchy. The resulting objects of the visible set will be recorded in the visible table (referred by VP). To pre-compute the visibility information, we change the model set to the k-D tree structure. For each node region R, some nearby large convex objects will be selected as the occluders. The k-D tree hierarchy of bounding boxes is then tested against the z-buffer to find which nodes are hidden by the occluder. The hidden sets for eight vertices are intersected, generating the hidden set of the occluder. And finally, all the hidden sets of all occluders are unioned to make up the global hidden set occluded by the occluder set. In such a way, we can get the visibility set for each region.

4. THE OPTIMIZED SEARCH ALGORITHM

In a large VE walkthrough system, in order to minimize I/O cost, the user's viewpoint is typically associated with a *disk cell* that contains the view frustum (Shou et al., 2001). For simplicity, in our experiments, we used a box-shaped cell (i.e., typical Window query) as a disk cell. The disk cell serves two purposes. First, if the frustums of subsequent frames are totally bounded in the disk cell, there is no need to access data from secondary storage. This can lead to higher frame rate. Second, whenever a user moves such that its view frustum is no longer bounded by the disk cell, new data has to be accessed from the secondary storage. The disk cell (corresponds to a query window) serves to define the region of space in the VE that contains objects that should be accessed from the secondary storage and brought into main memory for rendering.

However, in a walkthrough environment, consecutive disk cells, say $C_1, C_2, \ldots, C_i$, issued to the retrieval engine to access data from secondary storage may contain significant overlap. Consider a user's frustum cell is initially within cell C_1. When the user's frustum cell moves out of C_1, data of cell C_2 has to be loaded. Intuitively, we should only load objects that are in C_2 but not in C_1. Similarly, if C_3 becomes the current cell, then only objects in C_3 that are not in C_2 and C_1 should be accessed. This is also true for the visible sets V_1, V_2, and V_3. Unfortunately, the non-overlapped areas of cells are usually concave geometries, so it is difficult to describe such a region in each retrieval operation. It is also difficult to search for objects overlapping such a concave area in ViSA as the original search algorithm employs only box-shaped regions.

In (Shou et al., 2001), a novel complement overlap concept was introduced to reuse overlap information to retrieve only objects in the non-overlapped regions. ComplementOverlap between two regions are defined as follows: given a cell A, the space not contained in A is the complement of A, which is denoted as $\bar{A}$. If a bounding box BB (of a virtual object or a group of objects) overlaps (or intersects) $\bar{A}$, then we say that BB complement overlaps A. ComplementOverlap can be exploited in our context as follows:

- Given that we want to load objects of a new cell C, we can use a history of cells HS = $\{C_1, C_2, \ldots, C_i\}$ to filter away (spatial) objects that are already loaded in memory by the cells in HS. Thus, the problem becomes one of retrieving objects whose bounding boxes overlap C but do not overlap any of the cells in HS.

- Similarly, given that we want to load objects that are visible from a new cell C, we can use a history of regions whose visibility objects are already loaded in memory, HV = $\{V_1, V_2, \ldots, V_i\}$, and use them to filter away visible objects that are already loaded in memory.

Referring to our example again, if C_3 is the current cell whose objects we want to retrieve, then (1) objects that overlap C_3 but not C_1 and C_2 and (2) visible objects of C_3 but not visible from C_1 and C_2 are the ones that we are interested in.

We incorporated this idea to the basic ViSA search algorithm. As it works on the spatial and visible information of a disk cell, we refer to it as the SVC-search (*S*patial-*V*isual *C*omplement search). Figure 4 gives the algorithmic description for the SVCsearch. In the figure, we use T to denote an ViSA node, use E to denote an entry of the ViSA node, and use SP and VP to denote the references (pointers) to objects inside and visible objects of the ViSA node region as described in Section 3. As shown, the main difference between this algorithm and the basic search algorithm in Figure 3 is the additional ComplementOverlaps statements that appear in lines 4 and 8.

If we denote the cells that a user accesses as $C_1, C_2, C_3, \ldots$, based on the SVCsearch algorithm, the retrieval engine will issue the following queries to the database (ViSA): C_1, $C_2 - C_1$, $C_3 - (C_1 \cup C_2)$, $C_4 - (C_3 \cup C_2 \cup C_1), \ldots$, $C_{i+1} - (C_i \cup \ldots \cup C_2 \cup C_1)$ and so on. As a comparison, a traditional method would issue queries $C_1, C_2, C_3, \ldots$, to the database. For a SVC search like $C_{i+1} - (C_i \cup \ldots \cup C_2 \cup C_1)$, we can remove any cells from $\{C_1, C_2, \ldots, C_i\}$, if the bounding boxes of all their objects do not overlap C_{i+1}. Such cells have no effect on the query result because objects overlapping them cannot overlap C_{i+1}. Thus, before sending the query to the database, a filtering operation can be conducted on the cell list, so those cells not intersecting the current cell do not need to be considered in the SVCsearch algorithm. In our prototype walkthrough system, the number of cells in the history to be maintained is fewer than twenty in most cases. With such short cell lists, the CPU cost on the extra ComplementOverlap and Overlap testing is negligible. The above argument also applies to the ComplementOverlap operation on visibility data.

Besides exploiting the concept of complement overlap, we also keep track of the list of objects that are in memory. In this way, by comparing the IDs of the objects, we can avoid loading a memory-resident object. Since the number of objects to be compared is small, the computation cost is negligible compared to the I/O savings in loading the actual objects.

As a user "steps" out of a cell boundary, a SVCsearch returns a new result set. The algorithm guarantees that the result sets will have no overlap. However, as the object buffer in the main memory gets filled up, old objects should be freed to make space for new cells. Our solution is to delete from the buffer

Algorithm SVCsearch (T, C, HS, HV)
Input: T: pointer to node, C: Query window, HS: History of spatial search cells, HV: History of visible regions

```
1.  if T is not a leaf node /* search subtrees */
2.      for each entry E of node T do
3.          if (BB(E) Overlaps C)
4.              if BB(E) ComplementOverlaps all of $C_1, C_2, \ldots, C_i$ in HS
5.                  Invoke SVCsearch on the sub-tree whose root is the node
                    associated with E
6.                  for each region V obtained via pointer VP associated with E
7.                      if (V Overlaps C)
8.                          if V ComplementOverlaps all of $V_1, V_2, \ldots, V_i$ in HV
9.                              Return VP of node T, the pointer to list of visible
                                objects
10. else /* search leaf node */
11.     for each entry E of node T do
12.         if (BB(E) Overlaps C)
13.             if BB(E) Overlaps none of $C_1, C_2, \ldots, C_i$ in HS
14.                 E of node T is a qualifying record
15.                 Return the SP and VP of node T
16. Merge the query results of steps 9 and 15
```

Figure 4. The SVCsearch algorithm.

the objects of the cells that do not overlap the latest cell, while maintaining the objects whose cells overlap it.

5. EXPERIMENTAL STUDY

We implemented the ViSA system that employs all the proposed techniques. The system was built upon a PC with Pentium III and 128 megabytes of memory, running Windows NT4. We generated a synthetic dataset to simulate a large cityscape. There are about 100,000 virtual buildings requiring about 100 MB of harddisk space (inclusive of ViSA index), and more than 500 MB of memory space if fully loaded into main memory (in scene graph format).

We compare the ViSA system against the REVIEW system (Shou et al., 2001). REVIEW is the predecessor of ViSA designed to facilitate walkthrough in large VEs. REVIEW employs R-tree to index the VE data. However, it does not handle visibility data beyond those within a query region, i.e., given a query region, nothing outside of the region can be viewed. Currently, both systems support the same replacement and prefetching policies that are designed in REVIEW based on walkthrough semantics. The distance-priority-based replacement policy keeps those nodes that are close to the current viewpoint in memory, while replacing those nodes whose bounding boxes are distant from the current viewpoint. The prefetching scheme computes the position of the view cell that the user will be in based on the velocity in which the user is moving. Both schemes have been shown to perform better than other non-semantic-based schemes.

The metrics of measuring the quality of a walkthrough are the frame time and the smoothness of the walkthrough. Frame time is defined as the cycle time between two consecutive rendering operations. The time for database query, memory data manipulation, rendering and other overheads are all included in frame time. The smoothness of the walkthrough can be represented by how much each frame time varies from the average frame time. A walkthrough with a small average frame time and a small variance is considered to be of good quality.

Both the average frame time and the frame time variance of the ViSA and REVIEW system are listed in table 1. We can see the frame time and time variation of ViSA is close to those of REVIEW. With increasing query cell size ratio (defined as the ratio of the disk cell size over the view frustum cell size), the frame time and its variance become worse. This is because when the size of the querying box increases, more objects are loaded into memory. The average frame time of ViSA is about 3.5ms more than that of REVIEW and its average variance is about 17.3ms more than that of REVIEW. Thus, ViSA's performance is acceptable especially since it can provide much better visual fidelity than REVIEW (as we shall see shortly).

Cell size ratio	ViSA ft (ms)	ViSA vt (ms)	REVIEW ft (ms)	REVIEW vt (ms)
1.0	25.89	28.53	22.52	10.24
1.1	29.76	30.57	26.51	12.94
1.2	34.20	32.30	30.92	14.04
1.3	39.18	32.78	35.67	15.92
1.4	44.32	35.84	40.68	18.24
1.5	49.37	35.26	45.88	18.51
1.6	54.35	35.37	50.57	19.15

Table 1. Comparisons of ViSA and REVIEW on the average and variance of frame time (ft and vt) for different query cell size.

Figures 5(a) and 5(b) show the results on the rendering time for each frame when a user path is applied to ViSA system and REVIEW system. The results show that the ViSA has longer rendering time and less smooth frame rate because the visible objects were loaded and rendered. However, the performance is only marginally lower. At some frames (20 of total 500 frames), the frame time is very large for both systems because of the queries to the secondary storage.

Another straightforward approach to indexing visibility information for large VEs is to decompose the VE space into regular grid cells (REGULAR in short). We now compare the rendering time of REGULAR (Figure 5(c)) and ViSA (Figure 5(a)). REGULAR's average frame time (70.9ms) is longer than ViSA's (25.9ms). The variance of frame time of REGULAR is 343.6ms, longer than that of ViSA(28.5ms) as shown in table 1. It is due to REGULAR's longer average disk access time.

As expected, the fidelity of ViSA is significantly higher than REVIEW in walkthrough. The missing visible objects in REVIEW can be 0% (all the visible objects are occluded by the objects in the region) to ∞ (no objects in the region, e.g., a football ground, but many visible objects around, "blind spots"). To visually compare, Figure 6 displays three pairs of snapshots of display of ViSA and REVIEW using the same viewpoint for a same VE. It is clear from the figure that some visible buildings are only shown in the results of ViSA.

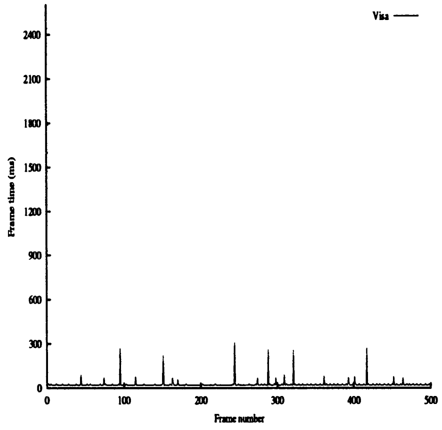
In the next experiment, we study the effect of trading off visual fidelity for performance. We use the cell size ratio 1.0 for this experiment. Figure 7 shows how the percentage of visible objects loaded affect the average frame time of the walkthrough. In other words, a point (x, y) in the figure means that (100-x)% of the objects were dropped, while $x\%$ were loaded into memory.

We can see that in Figure 7, as we decrease the percentage of visible objects information, the ViSA's frame time decreases. The frame time of REVIEW remains constant as REVIEW does not consider visibility information, but rather, only the spatial information. The curve of ViSA will not reach the horizontal line of REVIEW. The reason is that there is a constant additional overhead for ViSA to do the computation for the visibility part even if there is no visible object to be loaded.

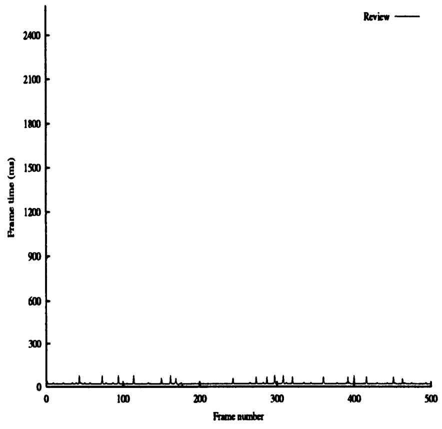
The fidelity for different dropping rates also can be visually compared. One set of three snapshots is shown in Figure 8. The visual fidelity decreases as the dropping of more visible objects in the visible set. In Figure 8(a), all distant cylinder towers and lower blocks are shown. In Figure 8(b), two most distant cylinder towers and some lower blocks are not shown. In Figure 8(c), only one most close cylinder tower and lower blocks are shown.

6. CONCLUSION

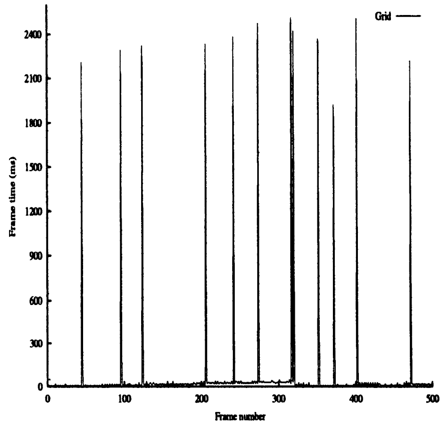
In this paper, we have presented a novel access method to organize large virtual environments that cannot fit into the main memory. The proposed structure, ViSA, stores both visibility and spatial proximity information, and facilitates optimized search and speedy retrieval of objects that are visible from and



(a) ViSA



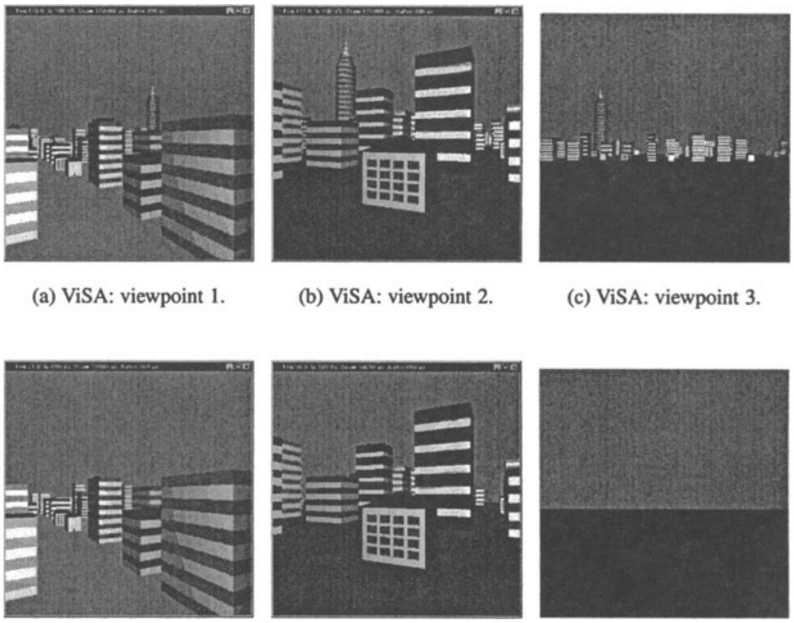
(b) REVIEW



(c) REGULAR

Figure 5. Experiment results of rendering time for each frame: the spikes indicate time and frame number for disk accesses.

spatially close to the viewpoint. This approach can successfully balance the fidelity and performance for walkthrough. We plan to extend this work in several directions. First, we would like to further study data structures that can organize the visibility data (instead of using a visibility table which can become a bottleneck for large number of objects). Second, we will take Level of Detail (LOD) into consideration in our future work. We plan to integrate this into the successor of ViSA.



(a) ViSA: viewpoint 1. (b) ViSA: viewpoint 2. (c) ViSA: viewpoint 3.

(d) REVIEW: viewpoint 1. (e) REVIEW: viewpoint 2. (f) REVIEW: viewpoint 3.

Figure 6. Visual comparisons of fidelity for ViSA and REVIEW. It shows that ViSA has higher visual fidelity for walkthrough. For viewpoints 1 and 2, some buildings, e.g., a cylinder tower, are missing from the REVIEW system. For viewpoint 3, all objects are missing, i.e., a "blind spot" is encountered during walkthrough.

ACKNOWLEDGMENTS

Yixin Ruan and Lidan Shou are supported by the University Research Scholarship. Initial discussion with Beng Chin Ooi prompted us to look into the issue of visibility data.

REFERENCES

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

Cohen-Or, D., Fibich, G., Halperin, D., and Zadicario, E. (1998). Conservative visibility and strong occusion for viewspace partitioning of densely occluded scenes. *Computer Graphics Forum*, 17(3):243–254.

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

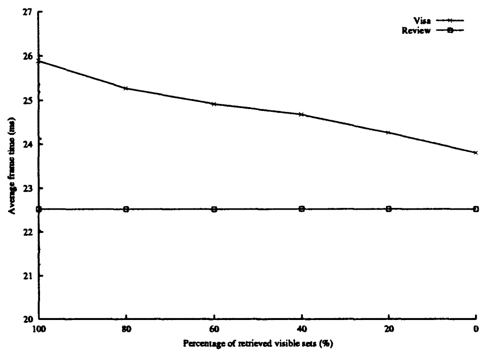
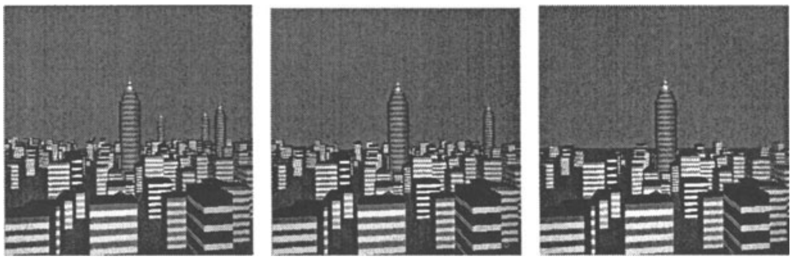


Figure 7. The average frame time of the walkthrough in ViSA, dropping the visible objects, against the REVIEW.



(a) No dropping of the visible objects.

(b) Dropping 20% of the visible objects.

(c) Dropping 40% of the visible objects.

Figure 8. Visual illustration of fidelity for ViSA when dropping the visible objects with small contribution values in order to maintain the frame time.

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

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

Panne, M. and Stewart, A. (1999). Effective compression techniques for precomputed visibility. In *Proceedings of Eurographics Workshop on Rendering*, pages 305–316.

Shou, L., Chionh, C., Huang, Z., Ruan, Y., and Tan, K. L. (2001). Walking through a very large virtual environment in real-time. In *Proc. VLDB 2001*, pages 401–410.

Teller, S. (1992). Computing the antipenumbra of an area light source. In *Proceedings of SIGGRAPH'92*, pages 139–148.

Yagel, R. and Ray, W. (1995). Visibility computation for efficient walkthrough of complex environments. *Presence*, 5(1):45–60.