Managing Gigabyte Virtual Environment Walkthrough

Lidan Shou, Jason Chionh, Zhiyong Huang, Yixin Ruan, Kian-Lee Tan

Department of Computer Science, National University of Singapore, Singapore 117543
Email: tankl@comp.nus.edu.sg

Abstract
In this paper, we study the problem of interactive walkthrough of a large virtual environment (VE) where the data representing 3D virtual objects can not reside completely in the main memory. We tap into the wealth of techniques (indexing, caching, prefetching) that have been widely used by the database community, and investigate how walkthrough semantics can be integrated into these techniques. In our approach, the VE data in the main memory are dynamically managed such that only those of the relevant 3D objects pertaining to the current user's viewpoint are loaded into the main memory. The objects in the VE are spatially indexed by a spatial index structure (R-tree) as opposed to splitting the VE into smaller rigid-sized grids. Queries are issued to the R-tree to retrieve only the relevant objects.
We propose a novel complement search algorithm to improve the standard search. In addition, we propose a cache replacement policy that considers the access pattern of walkthrough to facilitate effective memory management; and a prediction algorithm that computes the next likely position of the user in order to prefetch data. We implemented a prototype walkthrough system and evaluated it on a 1 GB synthetic dataset of a cityscape. Our results show that the proposed techniques deployed can lead to a high constant frame rate for the walkthrough.

1. Introduction

In a virtual reality (VR) system, one of the most basic requirements in supporting interactive walkthrough of the virtual environment (VE) is to achieve and maintain a high and constant rendering frame rate. However, for most VR systems, this requirement can only be realized for small VEs whose whole data set can reside in main memory. For more realistic VEs that are large in size (e.g., gigabytes (GB)), secondary storage must be used. In these environments, relying on system virtual memory management for walkthrough gives unacceptable frame rates.

In this paper, we investigate the use of database techniques for walkthrough of a large virtual environment (VE) whose data representing 3D virtual objects cannot reside completely in the main memory. In our approach, database techniques (such as indexing, caching and prefetching) are adapted to exploit the semantics of walkthrough in order to meet users' performance requirements.

To manage large VEs in relatively small-memory systems, we dynamically load into the main memory only those relevant 3D objects that pertain to the current user's viewpoint. This is realized through the use of a hierarchical spatial index structure, the R-tree structure, that clusters objects that are spatially close together. Queries issued to the R-tree retrieves only the relevant objects - those that are spatially close are likely to be regions that the users will also move into.

We propose a novel R-tree complement search algorithm to improve the standard search. This algorithm ensures that only the non-overlapped regions of successive queries are retrieved. In addition, we propose (1) a cache replacement policy that considers the access pattern of walkthrough to facilitate effective memory management. It keeps those nodes that are close to the current viewpoint in memory, while replacing those nodes that are distant from the current viewpoint when necessary; (2) a prediction algorithm that computes the next likely position of the user in order to prefetch data.

We implemented a prototype walkthrough system and evaluated it on a 1 GB synthetic dataset of a cityscape. Our results show that the proposed techniques deployed can lead to a high and constant frame rate for the walkthrough.

The rest of this paper is organized as follows. In the next section, we shall review some related works. Section 3 discusses how a large VE can be organized in the secondary storage and describes R-tree, a dynamic approach used in our work. In Section 4, we present techniques that optimize the I/O performance. We first present the novel complement search algorithm adopted to optimize the I/O performance of R-trees. The other two optimization techniques will be briefly discussed: the distance-priority-based replacement policy and a prefetching technique based on walkthrough semantics. Section 5 presents our experimental study and report our findings, and finally, Section 6 concludes the paper.

2. Related Work

The first group of work applied rigid spatial partitioning of VEs. In the UC Berkeley Soda Hall Walkthrough Project [5], Funkhouser et al. described a method to manage a large dataset that represents a 3D model of the Soda Hall. Their method partitioned the entire building into rigid subdivided cells using a variant of the k-D tree [3]. Rabinovich and Gotsman [9] used large terrains generated from a Digital Terrain Map (DTM) of Israel (consisting of 107 DTM points and 108 texels), divided their terrain into rigid tiles and subdivided all DTM points of each tile into a 3D octree. Davis et al. [4] used multiple linked quadtrees to subdivide a terrain.

The second group of work applied dynamic spatial partitioning of VEs. Pajarola et al. [8] used an object-oriented database to store all objects in the VE and built the R-tree over the database. However, only results of network transfer and database access were reported. Kofler et al. [7] built the levels-of-detail (LOD) straight into the R-tree. This method tightly couples the rendering engine to the database and poses a problem of scalability in a multi-user environment.

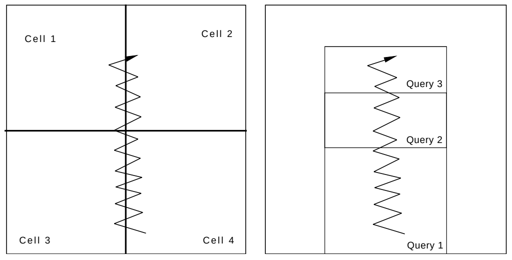
3. Organizing a Large VE

In this section, we first compare the rigid and dynamic spatial partitioning approaches for organizing a large VE in the secondary storage. Then, we describe the R-tree, a dynamic approach used in our work.

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

Figure 1(a) shows a simple case of a 2-dimensional environment being divided into 4 rigid cells. Assume that we only have enough main memory to load a single cell (the assumption is reasonable considering the size of a cell can be increased to the capacity of the memory).

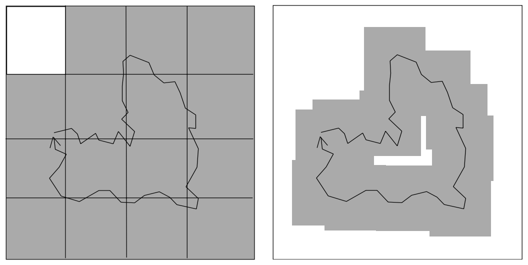
A swap has to be performed every time the user crosses a cell boundary. In Figure 1(a), 18 swaps have to be performed when the user crosses cell boundaries 18 times. Once the environment is rigidly subdivided, cell boundaries are always present at fixed positions in the environment and excessive swapping (and performance degradation) always occurs at these positions when the user crosses the cell boundaries often. Performance therefore becomes dependent on the physical position of the user path. Using a dynamic partitioning scheme, there are absolutely no fixed subdivisions of the environment. As such, we can issue queries centered to the user's current position to load necessary parts of the environment into main memory. As shown in Figure 1(b), under a dynamic partitioning scheme, it is possible to take only 1 swap in order to perform the same walkthrough. The swap occurs at Query 2. No swapping is required in Query 3 because we have enough main memory to store the results of both Query 2 and 3.



(a) Rigid spatial partitions (b) Dynamic spatial index

Figure 1: *Example to illustrate the number of swaps needed for rigid spatial partitions and dynamic spatial indexing.*

Suppose now we have enough memory to load all cells of the environment, the dynamic approach will require less memory for loading the environment. For a given user path, the shaded region in Figure 2(a) shows the area that is needed to load into memory for a rigid cell subdivision method. In comparison, the shaded area in Figure 2(b) shows the area needed to load into memory for the dynamic spatial indexing approach. From both figures, we see that the dynamic spatial partitioning approach requires less memory space to load the environment for the given user path. The rigid cell subdivision method also requires the splitting of objects if they lie across cell boundaries.



(a) Static spatial partition (b) Dynamic spatial indexing

Figure 2: *Example to illustrate the amount of data to be loaded into memory for rigid spatial partitions and dynamic spatial indexing.*

From the comparisons, we found the dynamic spatial partition is more suitable for the organization of a large VE.

Now, we start to describe the R-tree index [6] used in our work. As a typical spatial index structure, it facilitates speedy retrieval of objects according to their spatial locations. It is a height-balanced structure similar to the B-tree [2] with index records in its non-leaf nodes and pointers to objects in its leaf nodes. While the B-tree index 1D objects, the R-tree extends the indexing to n-dimensions.

The R-tree can be seen as a hierarchy of nested n-dimensional boxes. Every node of the R-tree contains many entries. The maximum number of entries a node can contain depends on the fan-out configuration of the R-tree. The leaf nodes contain entries of the form (MBR, ptr) where MBR is the minimum bounding box of the virtual object being indexed, and *ptr* is a pointer to the object being indexed. Note that *ptr* is an address when the node and objects are in memory, or a file name when the node and objects are stored in a file or on disk. The non-leaf nodes contain entries of the form (MBR, ptr) where MBR is the bounding box of all the bounding boxes of the entries of the lower level nodes and *ptr* is the pointer to the lower level node in the R-tree. In our implementation, we have also optimized the R-tree using the linear node splitting algorithm proposed in [1].

4. Optimizing Disk I/O

There are essentially three major bottlenecks in a walkthrough system: I/O - loading data (index and virtual objects) into memory; CPU - transforming data from disk-based format to in-memory format; and Graphics Processing Unit (GPU) - loading the graphics pipeline with data to be viewed.

In this paper, we focus on the I/O bottleneck. Our basic strategy is to associate the user's viewpoint with a *disk cell* which contains the view frustum. A disk cell corresponds to a region of space in the VE that contains objects that should be accessed from the secondary storage and brought into main memory for rendering. This approach has two main advantages. First, in an interactive walkthrough process, query frustums in consecutive rendering frames usually have significant overlaps, as the user's viewpoint moves smoothly. By using a cell whose size is larger than the view frustum, we can store the previous results in memory and to retrieve data merely for non-overlapped areas in the next rendering frames. Second, if the frustums of the next frames are totally bounded in the original box, there is no need to access data from secondary storage. This can also lead to higher frame rate. We propose a novel complement search algorithm to optimize the I/O performance. The details are described in subsection 4.1. Other I/O optimization techniques will be briefly described in subsection 4.2.

4.1. Complement Search Algorithm

As mentioned, a user's viewpoint is associated with a *disk cell*, Whenever the user moves out of its current disk cell, objects belonging to a new cell will have to be accessed. Figure 3 illustrates an example. Here, the user's frustum cell is initially within cell C1. When the user's frustum cell moves out of C1, data belonging to cell C2 has to be loaded. Intuitively, it doesn't make sense to load all objects belonging to cell C2 into memory, since there is a significant amount of overlap between cell C1 and C2. In fact, ideally, we should only load objects in the shaded region of C2. Similarly, if C3 becomes the current cell, then only objects in the shaded region of C3 need to be accessed.

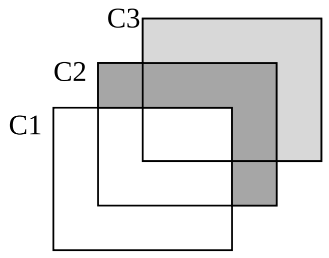


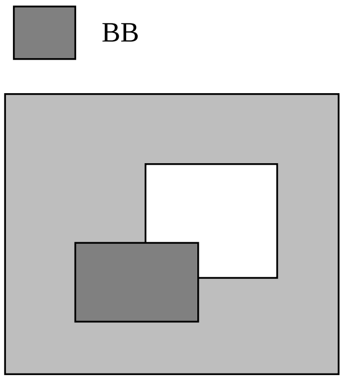
Figure 3: *An example to motivate complement search.*

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 R-tree as the original search algorithm employs only box-shaped regions.

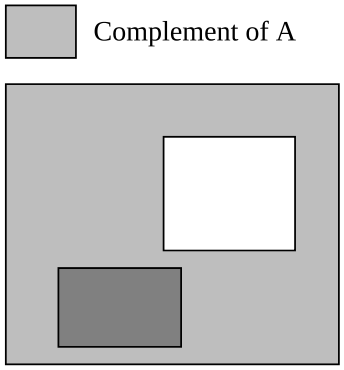
To minimize I/O, we propose a novel search method, called ComplementSearch, for R-tree that retrieves only objects in the non-overlapped regions. Essentially, the algorithm requires us to maintain a history of cells H = $\{C_1, C_2, \ldots, C_i\}$. Given that we want to load objects belonging to a new cell C, the problem becomes one of retrieving objects whose bounding boxes overlap C but do not overlap any of the cells in H. Referring to Figure 3 again, if C_3 is the current cell whose objects we want to retrieve, then objects that overlap C_3 but not C_1 and C_2 are the ones that we are interested in.

Figure 5 gives the algorithmic description for the complement search. One of the main operations is the **ComplementOverlap** operation between two regions. We define the complement overlap between these two regions as follows: given a cell A, the space not contained in A is the complement of A, which is denoted as $\bar{A}$. If a bounding box BB (of a virtual object or a group of objects) overlaps (or intersects) $\bar{A}$, then we say that BB complement overlaps A. In Figure 4, the bounding boxes of (a) and (b) complement overlap A, but that of (c) does not. In Figure 5, we use T to denote an R-tree node, and use E to denote an entry of the R-tree node.

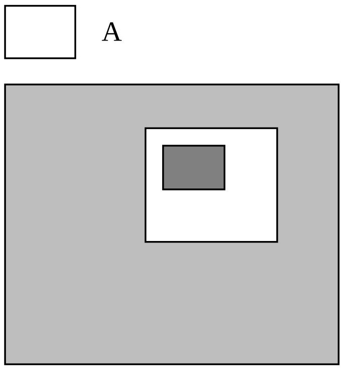
We now explain that the algorithm for complement overlap listed between lines 4-5 is necessary and cannot be replaced by the following pseudo-code using only the Overlaps operation:



(a) BB complement overlaps A



(b) BB complement overlaps A



(c) BB does not complement overlap A

Figure 4: *Complement overlap relations.*

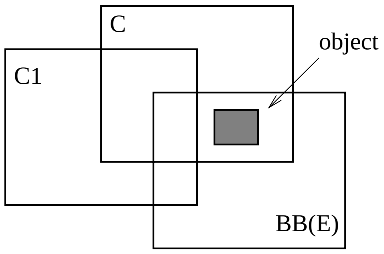
Algorithm ComplementSearch (T, C, H)

```
1.  if T is not a leaf node /* search subtrees */
2.      for each entry E of node T do
3.          if (BB(E) **Overlaps** C)
4.              if BB(E) **ComplementOverlaps** all of
                    $C_1, C_2, \ldots, C_i$ in H
5.                  Invoke **ComplementSearch** on the sub-tree
                        whose root is the node associated with E
6.  else /* search leaf node */
7.      for each entry E of node T do
8.          if (BB(E) **Overlaps** C)
9.              if BB(E) **Overlaps** none of $C_1, C_2, \ldots, C_i$ in H
10.                 E is a qualifying record
```

Figure 5: *The ComplementSearch algorithm.*

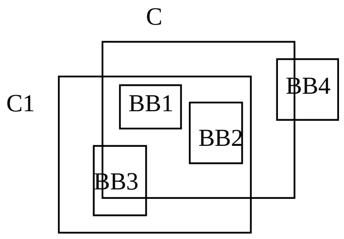
```
4'. If BB(E) **Overlaps** C and BB(E) does NOT **Overlaps**
        any of $C_1, C_2, \ldots, C_i$)
5'.     Invoke **ComplementSearch** on the tree whose root
            is the node associated with the entry E
```

This is because the **ComplementSearch** using lines 4' and 5' may miss some data. To illustrate the problem, we present a simple example shown in Figure 6(a). The black block represents an object in the database with a bounding box at the edges of the block. Assume (1) this bounding box belongs to a leaf node L, which is covered by an ancestor node with a larger bounding box BB(E) in the R-tree; (2) C1 and C are two consecutive cells that are accessed, C being the current cell. As illustrated in the figure, the non-leaf bounding box BB(E) overlaps both C1 and C, however, the object is contained in C but not in C1. So when the search algorithm traverses the R-tree and goes to the entry E, it finds BB(E) overlaps C and BB(E) overlaps C1, so the search process will not proceed to this entry, loosing the object in the final search results.



(a) BB(E) overlaps C1 and C

(using 4' and 5', the object will not be retrieved.)



(b)

Figure 6: *An example to illustrate complement search works.*

To address this problem, the search algorithm has to be conservative when processing non-leaf nodes. Our solution is to check whether BB(E) overlaps C and the complement of C1, as in lines 4-5 of the pseudo-code. If BB(E) overlaps C, it means that the bounding box is partially or completely contained in C. If BB(E) also complement-overlaps C1 at the same time, it means that the bounding box is not completely contained in C1, so the sub-tree under entry E needs to be searched further.

When the search process traverses to the leaf nodes of the R-tree, it is easy to know if the data object has been accessed by previous cells using the overlap algorithm. If BB(E) overlaps C1, so data object pointed to by C(E) must have been queried for C1 or other cells in the current history list, thus C(E) does not need to be accessed again. Otherwise, BB(E) overlaps C but does not overlap C1, so C(E) is a qualified object.

The algorithm ComplementSearch is conservative when accessing non-leaf nodes to guarantee that no objects in the new cell would potentially be lost. As the pseudo-code shows, it is obvious that complement-overlap checking only happens when the bounding box overlaps the current cell C, thus ComplementSearch will not access more R-tree nodes than the original algorithm. The algorithm will terminate the recursive search at the R-tree nodes with bounding boxes which are completely contained in all the $i+1$ cells (i history cells + current cell), as shown in Figure 6(b). When the size of the cell is large enough as compared to the average size of scene objects, and the overlapping between two cells of consecutive query operations is also large, ComplementSearch can stop searching at high level nodes in the R-tree, saving a large percentage of disk accesses. At the same time, ComplementSearch retrieves all data objects inside the current cell in one traversal of the R-tree, without accessing those that already have been retrieved in the past frames, minimizing the result set of objects that have not been accessed.

If we denote the cells that a user accesses as $C_1, C_2, C_3, \ldots$, based on the ComplementSearch algorithm, the retrieval engine will issue the following queries to the database (R-tree): $C_1, C_2 - C_1, C_3 - (C_1 \cup C_2), C_4 - (C_3 \cup C_2 \cup C_1), \ldots, C_i + 1 - (C_i \cup \ldots \cup C_2 \cup C_1)$ and so on. As a comparison, a traditional method would issue queries $C_1, C_2, C_3, \ldots$, to the database. For a complement search like $C_{i+1} - (C_i \cup \ldots \cup C_2 \cup C_1)$, we can remove any cells from $\{C_1, C_2, \ldots, C_i\}$, if the bounding boxes of all their objects do not overlap C_{i+1}. Such cells have no effect on the query result because objects overlapping them cannot overlap C_{i+1}. Thus, before sending the query to the database, a filtering operation can be conducted on the cell list, so those cells not interfering the current cell do not need to be considered in the ComplementSearch 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.

As a user "steps" out of a cell boundary, a complement search returns a new result set. The complement search algorithm guarantees that the result sets will have no overlap. However, as the object buffer in the main memory gets filled up, old objects should be freed to make space for new cells. That is, as shown in Figure 7, only objects in C_i need to be retained in the buffer. Unfortunately, since $C_1, C_2, \ldots, C_{i-1}$ may overlap with C_i, to remove the results of these cells may also remove objects in the overlapped regions (shown as the shaded area in the figure). A direct method to deal with this problem is to delete from the buffer the objects that do not overlap the latest cell, while maintaining the objects overlapping it. After this operation, the object buffer contains only the objects of cell C_i, of which the user is currently walking out.

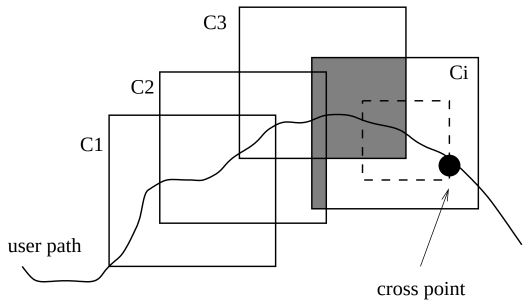


Figure 7: *Objects to be kept in memory.*

4.2. Other I/O Optimization Techniques

The first technique is the distance-priority-based replacement policy for R-tree index. Because of the limited memory size (compared to the large VE), index nodes that are cached in the memory may have to be replaced. In the R-tree, since a search process starts from the root node (at level 0), it is obviously beneficial to keep the root node in memory all the time. It guarantees that:

1. High-level nodes have a higher tendency to remain in the cache.
2. Nodes which have not been accessed for a long time or distant from the current viewpoint will have a higher preference to be replaced;
3. For nodes on the same level, the later a node is accessed, the more likely it is to reside in memory.

Considering the walkthrough of a large virtual environment, this distance-priority-LRU policy is expected to be superior to the traditional LRU scheme since a node that is currently distant from the user is not likely to be accessed in the near future. On the contrary, if a user takes a circular path and moves near to an area which was accessed long time ago, the distance-priority-LRU algorithm will detect that the node is near to the user and should be kept in the cache. However, under the LRU policy, this node will be assigned a low priority, since it has not been accessed for a long time, and may be removed from the cache.

The second technique is a prefetching algorithm for the cells the user is most likely to walk in. By observation, the user will be more likely to move in the direction of the current view. So it is reasonable to set the central point, $F(x,y,z)$, of the far clip plane of the view frustum as the center of the new cell. An inertial term $I = k \times velocity$ is added to F to produce the final result:

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

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

5. Experimental Study

We implemented the REVIEW (a *RE*al Time *VI*rtual *E*nvironment *W*alkthrough System) that employs all the proposed techniques. The system was built upon a Silicon Graphics Octane workstation running IRIX 6.5, with 400 megabytes of memory. Since the memory size is large, we set an upper limit of memory size to 30MB for the system.

We generated a synthetic dataset to simulate a large cityscape. There are about 900,000 virtual objects/boxes (Figure 8), requiring about 200 MB of harddisk space (inclusive of R-tree index), and more than 1 GB of memory space if fully loaded into main memory (in scene graph format). The distance between objects is 10 to 30 meters long, which is quite similar to realistic cases of a city. Another dataset of a residential area used in the experiments is shown in Figure 9.

The parameters of the view frustum include following: The eyesight of a user, or the *depth* of the view frustum, is set to be one kilometer long for the rendering, being consistent with the human eyes. The horizontal and vertical *field-of-views* are both set to 60 degrees.

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

The experiments were conducted in two groups. The first group of experiments allows us to fine-tune our configurations to find the optimal pre-fetch factor, cache size and cache policy. The second group illustrates the performance improvements of REVIEW to the traditional system. Due to space constraints, we shall not present the results of the tuning experiments.

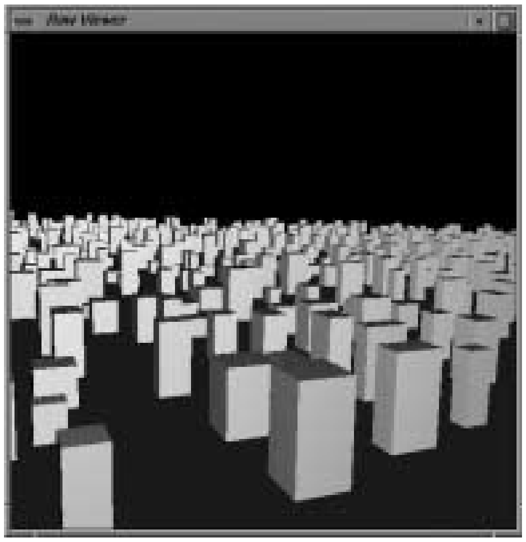


Figure 8: *A snapshot of a large cityscape.*

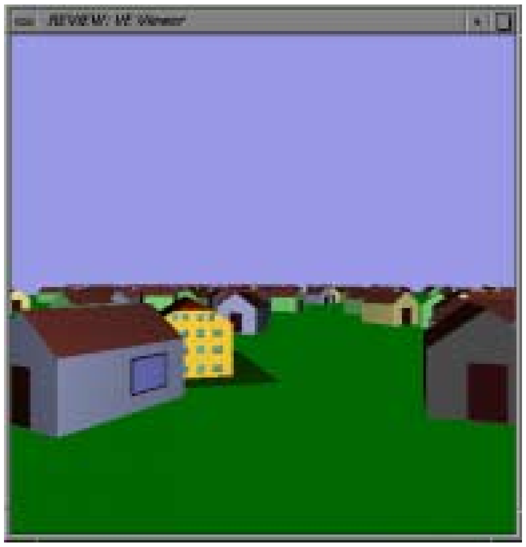


Figure 9: *A snapshot of a residential area.*

The user positions and orientations are recorded during different walkthrough sessions. These user sessions have different motion patterns. The fast, slow, turning, backward, and normal patterns were tested in the five sessions respectively. In the experiment, one recorded user session is used as the user path for all systems.

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. A real-time walkthrough requires the frame time shorter than 50ms, i.e., the frame rate higher than 20 frames per second. The smoothness of the walkthrough can be represented by how much each frame time varies from the average frame time. A walkthrough with a small average frame time and a small variance is considered of good quality.

Both the average frame time and the frame time variance of the REVIEW system are smaller than those of the BOX system. In addition, the frame time of REVIEW meets the requirement of real-time walkthrough as shown in table 1.

For the BOX, as the cell size decreases, more queries have to be issued to the database. Moreover, as the overlaps of the cells used in the query are not considered, the average frame time increases as the cell size decrease. In contrast, for the REVIEW, the queries will only return data in non-overlapped areas. Therefore, as cell size decreases, the average frame time does not increase. This shows that the REVIEW system is less sensitive to changes in cell sizes than the BOX system. As the cell size increases, the average frame time of the BOX system decrease. This does not mean that the walkthrough quality of the BOX system increases. The reason for the decrease in frame time is that fewer queries are issued to the database. The variance of the average frame time of the BOX is larger than that of the REVIEW. This means that the BOX system gives a choppy visual effect.

Cell	**REVIEW**		**BOX**	
ratio	**ft (ms)**	**vt (ms)**	**ft (ms)**	**vt (ms)**
1.0	46.04	6.77	169.30	13.01
1.1	46.61	6.82	155.87	12.48
1.2	45.2	6.89	143.81	11.99
1.3	48.86	6.99	126.33	11.29
1.5	52.08	7.34	107.87	10.39
1.7	57.18	7.60	100.64	10.03
2.1	57.75	7.78	93.32	9.66
2.5	60.92	7.78	86.86	9.32
2.8	54.26	7.81	87.92	9.37
3.1	61.46	7.84	84.80	9.20

Table 1: *Comparisons of REVIEW and BOX on the average and variance of frame time (ft and vt) for different cell sizes. The cell ratio is defined as the ratio of the disk cell size over the the view frustum cell size.*

6. Conclusion

In this paper, we have re-examined the issue of designing effective walkthrough system for large VEs that cannot fit into the memory. We have proposed a complement search algorithm for R-tree, a dynamic indexing structure for organizing the large VE on the secondary storage, together with a distance-priority-based cache replacement policy and a prefetching technique based on walkthrough semantics. We implemented REVIEW, a prototype walkthrough system and evaluated its performance on a large VE. Our results show that the proposed techniques can sustain relatively constant real-time frame rate and improve the visual effects.

References

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

2. R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of 1970 ACM SIGFIDET Workshop on Data Description and Access*, pages 107–141, 1970.

3. J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communication of the ACM*, pages 509–517, 1975.

4. W. Ribarsky D. Davis D., T. Y. Jiang and N. Faust. Intent, perception and out-of-core visualization applied to terrain. In *Proceedings of the IEEE Visualization'98*, pages 455–458, 1998.

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

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

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

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

9. B. Rabinovich and C. Gotsman. Visualization of large terrains in resource-limited computing environments. In *Proceedings of the IEEE Visualization'97*, pages 95–99, 1997.