Walking Through A Very Large Virtual Environment In Real-time

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

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

Abstract

Users expect high and constant rendering frame rates when they interactively navigate in a Virtual Environment (VE). However, when the VE is too large to fit into the main memory, the frame rates can become unacceptable. In this paper, we combine walkthrough semantics and database techniques, such as indexing, caching and prefetching, to improve the performance of walkthrough of a very large VE. We implemented a prototype walkthrough system called REVIEW (REal-time VIrtual Environment Walkthrough) and evaluated its performance on a 1 GB synthetic data-set generated to simulate a large cityscape. Our results show that the proposed techniques are effective for generating constant frame rate and improving the visual effects.

1 Introduction

Many of today's applications exploit Virtual Reality (VR) to meet users' information needs. For example, in the construction sector, a Virtual Environment (VE) of the to-be-constructed building may be designed to allow interested buyers to "view" the apartments (in order for them to make better choices and decisions). As another example, a virtual museum can be built to attract more electronic visitors to "tour" the place. For such VE to be accepted by users, a VR system that supports interactive walkthrough must provide high and constant rendering frame rates [3].

Most of the earlier work has assumed that the VE can fit into the main memory. However, this assumption is no longer reasonable. First, a realistic VE typically consists of thousands of virtual objects, each of which is represented by hundreds of polygons, and may take up thousands of megabytes of storage space. Second, as users' expectation increases, we can expect more complicated models that capture fine details of the actual environment as closely as possible to be designed. This will lead to an explosion in the size of the model, even if it is a simple environment. Clearly, when the VE is too large to fit into the main memory, it becomes crucial to manage the main memory space effectively, otherwise, the frame rates can become unacceptable. Similarly, for the memory-resident objects, we need to restrict the amount of objects fed into the graphics engine to guarantee good performance.

In this paper, we present our approach to realize quality visual effects (i.e., constant and high frame rates) in the interactive walkthrough of a very large VE. In our solution, there are two distinct data representations of the VE. In the secondary storage, data are organized based on their spatial location in an R-tree index. In the main memory, loaded data are transformed into a *scene graph* that are fed into the graphics engine. During a walkthrough session, each user viewpoint is associated with two convex cells. The first cell, called *frustum cell*, bounds the view frustum and is used to control the amount of objects that should be passed to the graphics engine for rendering. The second cell, called *disk cell*, is used to determine the objects that should be loaded into memory. For simplicity, both cell types are axis-aligned boxes. The disk cell contains the frustum cell and is larger than it. Whenever the user moves such that its frustum cell falls out of the corresponding disk cell, objects belonging to a new disk cell will have to be fetched. Clearly, two consecutive disk cells often have significant overlaps. To minimize I/O cost, we employ three optimiza-

Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

tion strategies. First, we propose a complement search algorithm that retrieves only the non-overlapped regions. Second, we exploit the access patterns of a walkthrough to design a novel cache replacement policy for the R-tree nodes, namely distance-priority-LRU policy. Essentially, the policy keeps those nodes that are close to the current viewpoint in memory, while replacing those nodes whose bounding boxes are distant from the current viewpoint. Finally, we also deployed a prefetching technique to predict the position of the view cell that the user will be in.

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

We have implemented a prototype walkthrough system called REVIEW (REal-time VIrtual Environment Walkthrough) [9]. We have also evaluated its performance on a 1 GB synthetic data-set generated to simulate a large cityscape. Our extensive study shows that the proposed techniques are effective, and REVIEW can provide constant frame rate and quality visual effects.

There are several related research works in the literature. In [4], techniques for managing large amounts of data during an interactive walkthrough of an architectural model are proposed. However, for a very large environment whose objects are sparsely distributed, the organization of data discussed in the paper is costly and will have little advantage.

In [6], a GIS system that allows users to "fly" over a large area was reported. A Level-Of-Detail-R-tree and progressive rendering techniques were deployed to speed up the interactive flying. To access objects overlapping the view frustum, the system needs to issue several queries to the database. In our system, however, the problem is solved with complement search and view frustum search algorithms.

Chim et al.[2] proposed a multi-resolution caching mechanism and investigated its effectiveness in supporting Internet based VR. Unfortunately, these schemes are tested in a simulation, instead of a real walkthrough system. In this paper, we will present prefetching results based on a real system and real user walkthrough sessions.

In [7], a desktop virtual reality interface to a geographic information system was reported. Only results of network transfer and database accesses were reported. No results of visual effect and rendering were included.

The rest of this paper is organized as follows. In Section 2, we shall present an overview of the system architecture, and discuss the issues to be addressed. Section 3 present the three optimization techniques to optimize I/O accesses. In Section 4, we present the view frustum search algorithm to optimize the graphics engine performance. Experimental results obtained from user walkthrough sessions in a desktop walkthrough system, are presented in Section 5, and finally, we conclude in Section 6 with directions for future work.

2 System Architecture and Issues

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

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

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

Figure 1 shows an overview architecture of the proposed walkthrough system. In our system, virtual objects are stored in files, and an R-tree index [5] is used to organize the virtual objects based on their spatial locality. In an R-tree index, the leaf nodes contain entries of the form (MBR, ptr) where MBR is the minimum bounding box of the virtual object being indexed, and *ptr* is a pointer to the object being indexed. Note that *ptr* is an address when the node and objects are in memory, or a file name when the node and objects are stored in a file or on disk. The non-leaf nodes contain entries of the form (MBR, ptr) where MBR is the

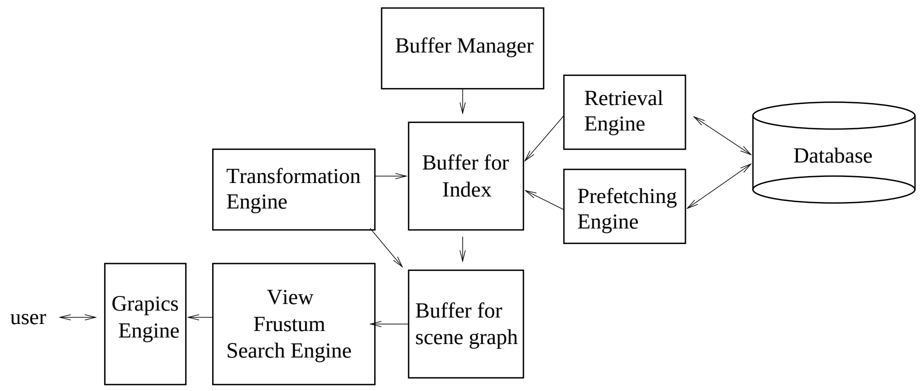


Figure 1: Architecture of proposed system.

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].

The data retrieved from the secondary storage are transformed into a *scene graph* [8] (using the Transformation Engine). The scene graph is a hierarchical structure that captures the virtual objects and their features such as locality, colors, textures and lightings. To better manage the main memory, it is also organized into two distinct pools - one for manipulating the data loaded from external storage, and the other for the scene graph. The scene graph is fed into the graphics engine for display.

Our basic strategy to the first and last problems is to associate the user's viewpoint with two different cells. The first cell, frustum cell, is a sufficiently small one that contains the view frustum. It serves as a search region to determine the in-memory objects that should be sent to the graphics engine. In this way, irrelevant data in the memory can be pruned away and only the visible data are passed to the graphics engine. This is realized by the View Frustum Search Engine.

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

To further improve performance, several other components have also been incorporated. We have a Prefetching Engine that predicts the future positions of the user viewpoint, and prefetches from the secondary storage those virtual objects. We have also designed a Buffer Manager that manages the main memory allocated to the R-tree index.

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

3 Optimizing I/O Performance

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

3.1 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 2 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 have to be loaded. Intuitively, it doesn't make sense to load all objects belonging to cell C2 into memory, since there is a significant amount of overlap between cell C1 and C2. In fact, ideally, we should only load objects in the shaded region of C2. Similarly, if C3 becomes the current cell, then only objects in the shaded region of C3 need to be accessed.

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.

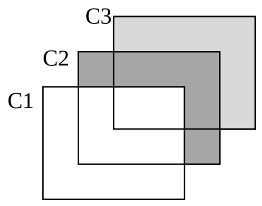


Figure 2: An example to motivate complement search.

In this section, we shall propose a novel search method for R-tree that retrieves only objects in the non-overlapped regions. We refer to the proposed search algorithm as the CSearch(Complement Search) algorithm. 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 2 again, if C_3 is the current cell whose objects we want to retrieve, then objects that overlap C_3 but not C_1 and C_2 are the ones that we are interested in.

Figure 4 gives the algorithmic description for the complement search. One of the main operations is the **COverlap** (Complement Overlap) operation between two regions. We define the complement overlap between these two regions as follows: given a cell A, the space not contained in A is the complement of A, which is denoted as $\bar{A}$. If a bounding box BB (of a virtual object or a group of objects) overlaps (or intersects) $\bar{A}$, then we say that BB complement overlaps A. In Figure 3, the bounding boxes of (a) and (b) complement overlap A, but that of (c) does not. In Figure 4, we use T to denote an R-tree node, and use E to denote an entry of the R-tree node.

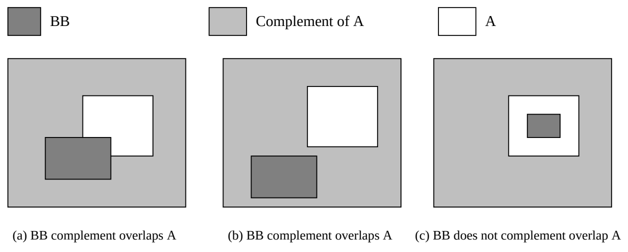

BB
Complement of A
A
(a) BB complement overlaps A
(b) BB complement overlaps A
(c) BB does not complement overlap A

Figure 3: Complement overlap relations.

The algorithm CSearch 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 will only happen when the bounding box overlaps the current cell C, thus CSearch will not access more R-tree nodes than the original algorithm. The algorithm will terminate the recursive search at the R-tree nodes with bounding boxes which are completely contained in all the $i + 1$ cells (i history cells + current cell). When the size of the cell is large enough as compared

```
Algorithm CSearch (T, C, H)

if T is not a leaf node /* search subtrees */
  for each entry E of node T do
    if (BB(E) Overlaps C)
      if BB(E) COverlaps all of $C_1, \ldots, C_i$ in H
        Invoke CSearch on the sub-tree
        associated with entry E
else /* search leaf node */
  for each entry E of node T do
    if (BB(E) Overlaps C)
      if BB(E) Overlaps none of $C_1, \ldots, C_i$ in H
        E is a qualifying record
```

Figure 4: The CSearch algorithm.

to the average size of scene objects, and the overlapping between two cells of consecutive query operations is also large, CSearch can stop searching at high level nodes in the R-tree, saving a large percentage of disk accesses. At the same time, CSearch retrieves all data objects inside the current cell in one traversal of the R-tree, without accessing those that already have been retrieved in the past frames, minimizing the result set of objects that have not been accessed.

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

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

As a user "steps" out of a cell boundary, a complement search returns a new result set. The complement search algorithm guarantees that the result sets will have no overlap. However, as the object buffer in the main memory gets filled up, old objects should be freed to make space for objects of new cells. Once the buffer is full, we need to remove the previous results and to keep only objects of the current cell in the buffer. That is, as shown in Figure 5, only objects in C_i need to be retained in the buffer. Unfortunately, since $C_1, C_2, \ldots, C_{i-1}$ may overlap with C_i, to remove the results of these cells may also remove objects in the overlapped regions (shown as the shaded area in the figure). A direct method to deal with this problem is to delete from the buffer the objects that do not overlap the latest cell, while maintaining the objects overlapping it. As shown in Figure 5, objects overlapping the latest cell C_i are kept in memory. After this operation, the object buffer contains only the objects of cell C_i, of which the user is currently walking out.

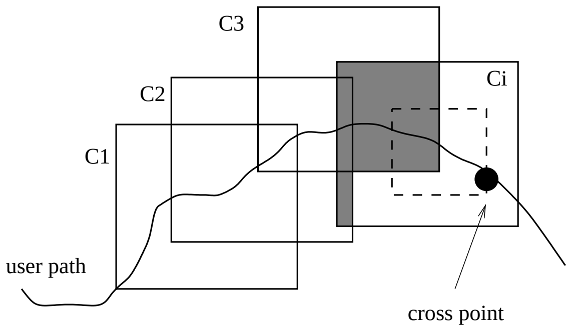


Figure 5: Objects to be kept in memory.

3.2 Distance-priority-based Replacement Policy

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

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

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

We note that a smaller P value means higher priority.

Our cache replacement policy is based on the *block-distance* between two axis-aligned boxes, $A = \{[X^a_{min}, X^a_{max}], [Y^a_{min}, Y^a_{max}], [Z^a_{min}, Z^a_{max}]\}$ and $B = \{[X^b_{min}, X^b_{max}], [Y^b_{min}, Y^b_{max}], [Z^b_{min}, Z^b_{max}]\}$, which is defined as follows:

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

where $dist_x = min\{|p_a - p_b|, p_a \in [X^a_{min}, X^a_{max}], p_b \in [X^b_{min}, X^b_{max}]\}$, $dist_y$ and $dist_z$ are defined similarly.

It is well known that the effectiveness of a cache replacement policy depends largely on the access pattern. In a walkthrough application, a user normally walks across the whole scene and turns left, right or backward sometimes. Since the R-tree index is organized to represent the spatial subdivision structure, when a user walks out of a high-level bounding box at some time and is already quite distant from it, he(she) is not likely to access it or its descendants in the near future. Based on this observation, the replacement can be made based on the following information:

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

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

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

According to the above points, we define a function

$$f = f(P, QN, DIST(Q, BB)),$$

where P is the initial priority, QN is an integer number representing how long the entry resides in the cache, and DIST(Q, BB) is the block-distance of the current query box (or cell) Q and the bounding box BB of the node. The f value is computed for each entry in the cache considered to be replaced. The function is defined in such a way that an entry with the highest f value will be replaced. There are many ways to define the function f to meet the above conditions. For simplicity in our implementation, we use the following definition:

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

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

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

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

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

3. For nodes on the same level, the more recent 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.

3.3 Prefetching Algorithm

When approaching the boundary of a cell, the user is likely to move out of the cell soon. So before the user goes out of the cell, we need to prefetch data of another cell using a different thread. As shown in Figure 6, C_i is the current cell. C_2 is the prefetched cell for C_1 and C_3 is the prefetched cell for C_2. As C_1 and C_2 have large overlap, complement search algorithm can be applied during the prefetching of C_2. C_3 can also be prefetched complementing C_1 and C_2.

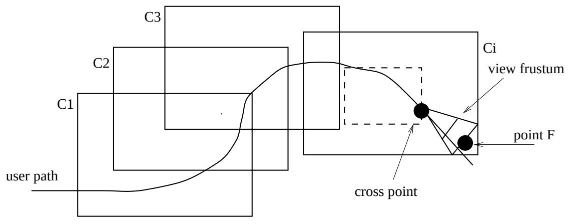


Figure 6: Prefetching objects.

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

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

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

where velocity is the current velocity vector of the user and k is an adjusting factor. If a user moves fast 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 viewpoint. If the user turns away from the predicted direction, the view frustum should still be within the predicted cell. However, if the user's direction is not correctly predicted, the user will move out of the predicted cell in a short time. As a consequence, the frequency of queries increases under such circumstance. Fortunately, as the new cell has significant overlap with the current cell, with our proposed CSearch algorithm, only a small number of objects need to be retrieved. If the user walks back into the old cells again, it is not necessary to issue a new query.

4 Optimizing GPU Performance

Object data returned from the disk cell are stored in an object buffer in the main memory. However, it is not practical to send all these data to the GPU (Graphics Processing Unit) or graphics pipeline, as the object buffer may contain a large number of objects that are not visible in the current frame. Culling the object buffer using the view frustum (see Figure 7) will improve the performance of the GPU because fewer objects will be transmitted and rendered.

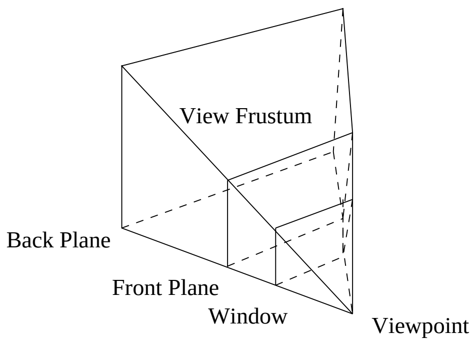


Figure 7: Culling the object buffer using the view frustum.

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

Let H be a frustum cell that bounds the view frustum. Essentially, what we want is to send only objects in H to the graphics pipeline. To determine whether a 3-dimensional box R overlaps H, we need to consider the four possibilities shown below:

1. $R \cap H = \phi$

2. $R \subseteq H$

3. $H \subseteq R$

4. $R \cap H \neq \phi, R \not\subset H, H \not\subset R$

We need to check face-face intersection between H and R. Since each face of H splits the whole 3D space into two half spaces, we can first check whether all vertices of R (H) are in the same *positive half space* where the normal of the face points to. If they are, as shown in Figure 8, R (H) does not intersect any face of H (R). When the size of each object is much smaller compared to the size of H, most bounding boxes in the lower level of R-tree do not overlap H. So the algorithm can be accelerated.

If all vertices of R (H) are not in any of the positive half spaces, then if there is one vertex of R (H) inside H (R), corresponding to case (2) or (3), the two volumes overlap. Otherwise, we need to check intersection face by face. If one intersection is found, it means that the two overlap (case 4). Otherwise, there is no overlapping (case 1).

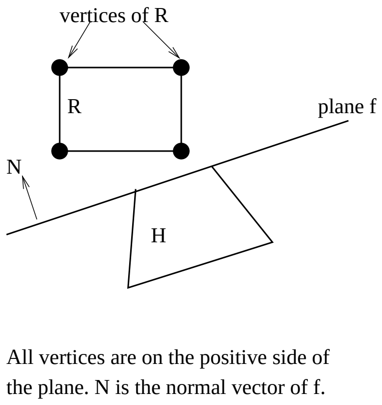

vertices of R
R
plane f
N
H
All vertices are on the positive side of the plane. N is the normal vector of f.

Figure 8: Intersection checking with vertices of R.

The culling algorithm is shown in Figure 9. The input of the algorithm is as the following: A frustum cell H having s polygons, denoted as $p_1, p_2, \ldots, p_s$, and t vertices, denoted as v_1, v_2, ..., v_t; an n-dimensional rectangle R, with vertices r_1, r_2,..., r_8, and faces f_1, f_2, ..., f_6.

This algorithm is applied repeatedly to check if the bounding boxes of the objects overlap the frustum cell. For object data stored in memory, we first check whether the frustum cell overlaps the cell-level bounding boxes. If it is true, the search process will go on with all objects in the result set corresponding to that cell. Otherwise, the whole cell does not need to be rendered. As a result, a large portion of objects in memory can be filtered, saving a lot of burden on the graphics engine.

5 Experimental Results

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

Algorithm ViewCulling

```
for (each face $p_i$ of H)
    if (all vertices of R are on the positive side of $p_i$)
        Return FALSE
if (any vertices of R is inside H)
    Return TRUE;
else if (any vertices of H is inside R)
        Return TRUE;
    else
        for (each face $p_i$ of H)
            if ($p_i$ intersects one of $f_1$, $f_2$, ..., $f_6$)
                Return TRUE
            return FALSE
```

Figure 9: The view culling algorithm.

We generated a synthetic data-set to simulate a large cityscape. There are about 900,000 virtual objects/boxes, requiring about 200 MB of hard-disk space (inclusive of R-tree index), and more than 1 GB of memory space if fully loaded into main memory (in scene graph format). The distance between objects is 10 to 30 meters long, which is quite similar to realistic cases of a city.

The parameters of the view frustum include the following. The eyesight of a user, or the *depth* of the view frustum, is set to be one kilometer long, consistent with the real walkthrough. The horizontal and vertical *field-of-views* are both set to 60 degrees, which is a standard value in graphics applications.

To test the effectiveness of various techniques depicted in this paper, we ran experiments under different system configurations. For clarity of the description, we will use the following abbreviations:

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

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

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

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

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

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

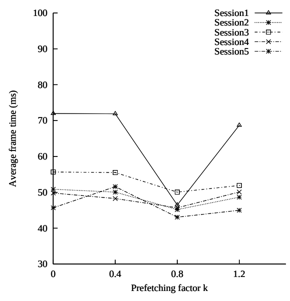
1. The default prefetching factor k, except NP, is 0.8
2. The default index cache size is 1MB
3. The default index cache replacement policy is *distance-priority-LRU*, with the weight factors set to $\alpha = \beta = \gamma = 0.333$.

5.1 Tuning The Parameters in REVIEW

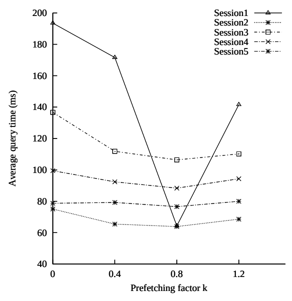
We note that there are several parameters in REVIEW that has to be tuned. First, in the prefetching algorithm, its effectiveness depends largely on the semantics of the user walkthrough and the algorithm itself. To fine-tune the prefetching factor k in the prefetching algorithm, we used several user sessions to find an optimal k value. Figure 10 shows the results. These user sessions have different motion patterns. The fast, slow, turning, backward, and normal patterns were tested in the five sessions respectively. In the figure, each curve represents an individual walkthrough session.

Frame time is defined as the cycle time between two consecutive rendering operations. The time for database query, memory data manipulation, rendering and other overheads are all included in frame time. A real-time walkthrough requires the frame time shorter than 50ms, i.e., the frame rate higher than 20 frames per second. In Figure 10(a), all sessions have a minimal average frame time around the point where $k = 0.8$. As different sessions have quite different motion patterns, the effect of k is also quite different. In session 1, since the user moves rapidly, it is crucial to make an accurate prediction of the position of the new cell. If k is too small, the user will move out of the new cell more frequently, generating more prefetches. On the contrary, if the prediction is too far away from the current position, as less overlap can be obtained between two consecutive cells, the query time will increase and so will the frame time. As to the other sessions, since the speeds of the user's motion are relatively small, the curves of the average frame time and query time display similar trends. The average query times are shown in Figure 10(b). The query times also have a minimum value around the point where $k = 0.8$.

Next, we need to tune the parameters used in the replacement policy. Figure 11 illustrates the results of index cache performance with different cache replacement policies under various cache sizes. The figure shows that the *distance-priority-LRU* scheme performs better than LRU when cache size is smaller



(a) Average frame time



(b) Average query time

Figure 10: The effect of the prefetching factor k to system performance

than 20 Megabytes. For cache size between 5 to 15 Megabytes, the best setting is the DPLRU, where $\alpha = \beta = \gamma = 0.333$. As the cache is implemented with software, it takes more than *O(1)* time for finding a cache entry, as opposed to hardware implementation. Therefore, the overhead caused by the software implementation of the cache offsets the performance improvement at large cache sizes. This explains the increase in query time at cache sizes larger than 15 Megabytes.

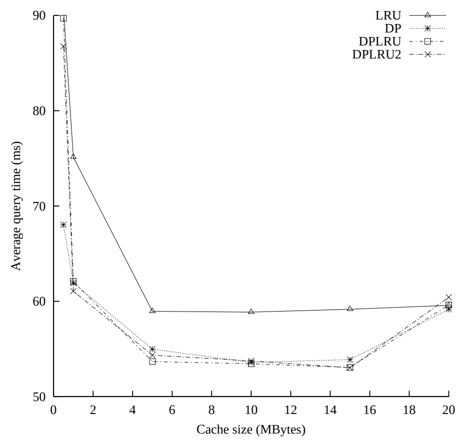


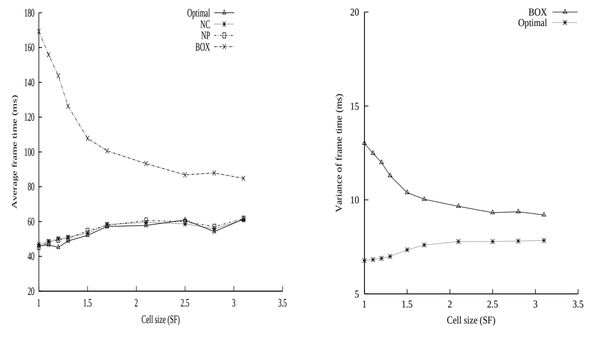
Figure 11: Cache performance with various cache sizes

5.2 Performance Improvements of REVIEW

5.2.1 Results of rendering frames

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

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



(a) Average frame time

(b) Variances of avg. frame time

Figure 12: Comparison of average frame time

As shown in Figure 12(a), for different cell sizes, the average frame time of the traditional system is much longer than the optimized system. For the traditional system, as the cell size decreases, more queries have to be issued to the database. Moreover, as the overlaps of the cells used in the query are not considered, the average frame time increases as the cell size decreases. In contrast, for the configurations which implement complement query interface, i.e. *Optimal*, *NC*, and *NP*, 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 size than the BOX system. From the figure, we can also see that the rendering frame rate of REVIEW is higher than that of the BOX. As the cell size increases, the average frame time of the BOX system decreases. But this does not mean that the walkthrough quality of the BOX system increases. The reason for the decrease in frame time is that fewer queries are issued to the database.

As the query boxes become larger, the search time per query is also longer. User will experience a serious "pause" during each query. Hence, the walkthrough effect is not better. This is confirmed in Figure 12(b). In the figure, the variance of the average frame time of the BOX is larger than that of the Optimal configuration. This means that the frame time of the BOX system varies more than that of the Optimal system, giving a choppy visual effect. In contrast, the frame time of the Optimal configuration has lower variation and gives a more constant frame rate. The results also show that caching and prefetching have less effects on the average frame time than the complement search algorithm.

Figure 13 shows the results on the rendering time for each frame when a user path is applied to an Optimal system and a BOX system. Both of them use the same cache size of 1 Megabytes. The results show that the Optimal system has shorter rendering time and much smoother frame rate. Since the complement search algorithm returns smaller result set, the Optimal system also needs less time to transfer the result set into the scene graph structure, so the change in rendering frame time is much smaller for each query.

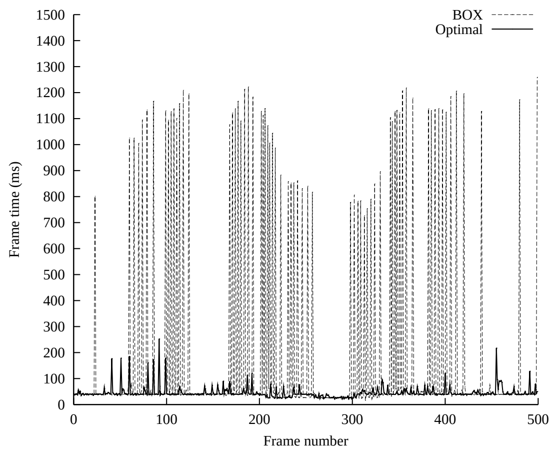


Figure 13: Rendering time for each frame

5.2.2 Results of search cost

Figure 14 shows the arithmetic average search time of each query in user walkthrough sessions in different sized databases. It is apparent that the Optimal system outperforms the BOX system in databases of different sizes.

In Table 1, the average disk accesses per query are shown for five different walkthrough sessions. The disk I/Os of the Optimal system varies from 9% to 21% of those of the BOX system. Therefore, it is apparent that the Optimal system performs better than the BOX in disk I/Os.

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

Table 1: Disk I/Os Per Query

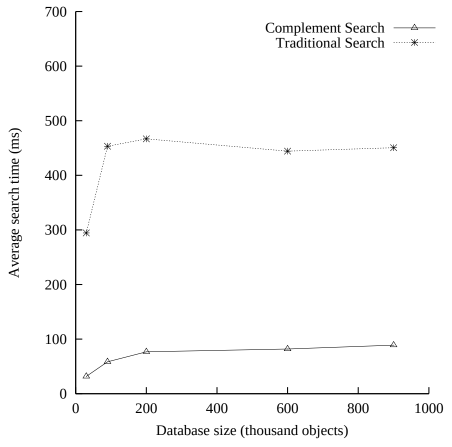


Figure 14: Average search time

5.2.3 Results of optimizing GPU performance

Table 2 shows the results of the view frustum culling algorithm discussed in section 4. The Optimal system runs this algorithm to remove irrelevant objects before sending the rest to the graphics engine. Therefore, the algorithm tries to remove as many objects as possible to reduce the workload on graphics subsystem. The left column of the table contains various cell sizes, while the right one shows the respective average percentage of objects that are culled away before the rendering. The data in the table illustrate that a large percentage of object data in memory can be filtered and need not to be sent to the graphics engine. The percentage increases as the cell size increases. This is because when the cell size increases, more irrelevant data are retrieved into the memory, so the algorithm can find more irrelevant objects in the memory buffer.

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

Table 2: Results of view frustum culling

6 Conclusion

In this paper, we reexamined the issue of designing effective walkthrough system for a very large virtual environment that cannot fit into the memory. Our solution is to address the various bottleneck individually. We implemented REVIEW, a prototype walkthrough system and evaluated its performance on a very large virtual environment. With these techniques, the system can sustain constant real-time frame rate and achieve better visual effects.

As for the future work, we plan to extend the REVIEW system to incorporate the Levels-of-Details(LODs) in the virtual scene, as well as the visibility information.

7 Acknowledgment

This project is partially funded by the University Research Grant RP982694.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] R. Pajarola, T. Ohler, P. Stucki, K. Szabo, and P. Widmayer. The Alps at your fingertips: Virtual reality and geoinformation systems. In *Proc. ICDE International Conference on Data Engineering*, pages 550–557, 1998.

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

[9] L. Shou, C. Chionh, Y. Ruan, Z. Huang, and K. L. Tan. REVIEW: A real-time virtual walkthrough system (Demo). In *Proc. ACM SIGMOD 2001*, Santa Barbara, CA, May 2001.