Efficient Approximate Visibility Query in Large Dynamic Environments

Leyla Kazemi[1], Farnoush Banaei-Kashani[1], Cyrus Shahabi[1], and Ramesh Jain[2]

[1] InfoLab Computer Science Department
University of Southern California, Los Angeles, CA 90089-0781
[2] Bren School of Information and Computer Sciences
University of California, Irvine, CA 92697-3425
{lkazemi,banaeika,shahabi}@usc.edu,jain@ics.uci.edu

Abstract. Visibility query is fundamental to many analysis and decision-making tasks in virtual environments. Visibility computation is time complex and the complexity escalates in large and dynamic environments, where the visibility set (i.e., the set of visible objects) of any viewpoint is probe to change at any time. However, exact visibility query is rarely necessary. Besides, it is inefficient, if not infeasible, to obtain the exact result in a dynamic environment. In this paper, we formally define an *Approximate Visibility Query (AVQ)* as follows: given a viewpoint v, a distance ε and a probability p, the answer to an AVQ for the viewpoint v is an approximate visibility set such that its difference with the exact visibility set is guaranteed to be less than ε with confidence p. We propose an approach to correctly and efficiently answer AVQ in large and dynamic environments. Our extensive experiments verified the efficiency of our approach.

1 Introduction

Visibility computation, i.e., the process of deriving the set of visible objects with respect to some query viewpoint in an environment, is one of the main enabling operations with a majority of spatial analysis, decision-making, and visualization systems ranging from GIS and online mapping systems to computer games. Most recently the marriage of spatial queries and visibility queries to spatio-visual queries (e.g., k nearest visible-neighbor queries and nearest surrounder queries) has further motivated the study of visibility queries in the database community [4, 11, 12]. The main challenge with visibility queries is the time-complexity of visibility computation which renders naive on-the-fly computation of visibility impractical with most applications.

With some traditional applications (e.g., basic computer games with simple and unrealistic visualization), the virtual environment is simple and small in extent, and hence, it can be modeled merely by *memory-resident* data structures and/or synthetic data. With such applications, a combination of hardware solutions (e.g., high-end graphic cards with embedded visibility computation modules) and memory-based graphics software solutions are sufficient for visibility analysis. However, with emerging applications the virtual environment is becoming large and is modeled based on massive geo-realistic data (e.g., terrain models and complex 3D models) stored on disk (e.g., Google Earth, Second Life). With these applications, all proposed solutions [10, 14, 15] inevitably leverage pre-computation to answer visibility queries in real-time. While such solutions perform reasonably well, they are all rendered infeasible with dynamic environments, as they are not designed for efficient update of the pre-computed visibility information.

In this paper, for the first time we introduce *approximate visibility query (AVQ)* in large dynamic virtual environments. Given a viewpoint v, a distance ε, and a confidence probability p, the answer to AVQ with respect to the viewpoint v is an approximate visibility vector, which is guaranteed to be in a distance less than ε from the exact visibility vector of v with confidence p, where visibility vector of v is the ordered set of objects visible to v. The distance ε is defined in terms of the cosine similarity between the two visibility vectors. Approximation of the visibility is the proper approach for visibility computation with most applications because 1) exact answer is often unnecessary, 2) exact answer is sometimes infeasible to compute in real-time due to its time-complexity (particularly in large dynamic environment), and 3) a consistent approximation can always converge to the exact answer with arbitrary user-defined precision. To enable answering AVQs in large dynamic environments, we propose a pre-computation based method inspired by our observation that there is a strong *spatial auto-correlation* among visibility vectors of distinct viewpoints in an environment, i.e., the closer two viewpoints are in the environment, often the more similar are their visibility vectors. Consequently, one can approximate the visibility vector of a viewpoint v by the visibility vectors of its close neighbors. Towards this end, we propose an index structure, termed *Dynamic Visibility Tree* (*DV-tree* for short), with which we divide the space into disjoint partitions. For each partition we pick a representative point and pre-compute its visibility vector. The partitioning with DV-tree is such that the distance between the visibility vector of any point inside a partition and that of the representative point of the partition is less than ε with confidence p. Therefore, with DV-tree we can efficiently answer an AVQ for viewpoint v (with distance ε and confidence p) by returning the pre-computed visibility vector of the representative point of the partition in which v resides. Figure 1 depicts an example of AVQ answering, which we discuss in more detail in Section 3. Accordingly, Figure 1a shows the exact visibility for viewpoint v while Figure 1b depicts the approximate result returned by DV-tree (i.e., visibility of the representative point of the partition in which v resides).

DV-tree is particularly designed to be efficiently maintainable/updatable in support of visibility computation in *dynamic* environments. In a dynamic environment, at any time a set of moving objects are roaming throughout the space, and consequently, the visibility vectors of some viewpoints may change. Accordingly, the partitioning of the DV-tree must be updated to reflect the visibility changes. However, this process is costly as it requires computing the visibility vectors of *all* viewpoints within each (and *every*) partition, to be compared with the visibility vector of the representative point of the corresponding partition. We have devised a two-phase filtering technique that effectively reduces the overhead of the DV-tree partition maintenance. In the first phase, termed *viewpoint filtering*, we effectively filter out the viewpoints whose visibility remains unchanged despite the recent object movements in the environment. For the remaining viewpoints (i.e., those that are filtered in), we proceed with the second phase, termed *object filtering*. In this phase, before computing the visibility vector for each of the remaining viewpoints, we effectively filter out all objects in the environment whose visibility status with respect to the viewpoint remains unchanged despite the recent object movements in the environment. After the two-phase filtering process, we are left with a limited number of viewpoints whose visibility must be computed with respect to only a limited number of objects in each case. Therefore, we can efficiently compute their visibility and also revise the corresponding DV-tree partitions accordingly, if needed.

Finally, through extensive experiments, we show that our approach can efficiently answer AVQ in large dynamic environments. In particular, our experiments show that DV-tree result is more than 80% accurate in answering AVQ, while the update cost is tolerable in real scenarios. This validates our observation about the spatial auto-correlation in visibility vectors. Note that both the approximation error and the update cost can be interpreted as the visual *glitch* and frame rate delay, respectively, in visualization systems. In general, for a DV-tree with higher error-tolerance, larger partitions are generated. This results in more visual *glitches* as a viewpoint moves from one partition to another, since its visibility may encounter a noticeable change during this transition. However, less frame rate delay is expected, because the DV-tree update is less costly as compared to that of a less error-tolerant DV-tree. On the other hand, for a DV-tree with less error-tolerance, smaller partitions are generated, which results in less glitches, but higher frame rate delays. Thus, there is a trade-off between these two system faults. Our experiments also show that DV-tree significantly outperforms a competitive approach, HDoV-tree [15], in both query response time and update cost.

The rest of the paper is organized as follows. Section 2 reviews the related work. In Section 3, we formally define our problem, and successively in Section 4 we present an overview of our proposed solution. Thereafter, in Sections 5 and 6 we explain the processes of construction and update for our proposed index structure (DV-tree). Section 7 presents the experimental results. Finally, in Section 8 we conclude and discuss the future directions of this study.

2 Related Work

Visibility analysis is an active research topic in various fields, including computer graphics, computer vision, and most recently, databases. Below, we review the existing work on visibilty analysis in two categories: memory-based approaches for small environments and disk-based approaches for large environments.

2.1 Memory-based Approaches

In [1, 3, 9, 17], different approaches are proposed for fast and efficient rendering. The end goal of most of these studies is to develop efficient techniques to accelerate image generation for realistic walkthrough applications [6]. In addition, there are a few proposals [2, 5, 7, 16] from the computer graphics community on visibility analysis in *dynamic* environments. However, the aforementioned work assume the data are memory-resident, and therefore, their main constraint is the computation time, rather than disk I/O. This is not a practical assumption considering the immense data size with today's emerging applications with large virtual environments.

2.2 Disk-based Approaches

On the other hand, in database community, many spatial index structures (e.g., R-tree, quad-tree) are proposed for efficient access to large data. Here, the goal is to expedite the search and querying of *relevant* objects in databases (e.g., kNN queries) [13], where the relevance is defined in terms of spatial proximity rather than visibility. However, recently a number of approaches are introduced for efficient visibility analysis in *large* virtual environments that utilize such spatial index structures to maintain and retrieve the visibility data ([10, 14, 15]). In particular, [10, 14] exploit spatial proximity to identify visible objects. However, there are two drawbacks with utilizing spatial proximity. First, the query might miss visible objects that are outside the query region (i.e., possible false negatives). Second, all non-visible objects inside the query region would also be retrieved (i.e., numerous false positives). Later, in [15], Shou et al. tackle these drawbacks by proposing a data structure, namely HDoV-tree, which pre-computes visibility information and incorporates it into the spatial index structure. While these techniques facilitate answering visibility queries in large environments, they are not designed for dynamic environments where visibility might change at any time. They all employ a pre-computation of the environment that is intolerably expensive to maintain; hence, inefficient for visibility query answering in dynamic environments.

In this paper, we focus on answering visibility queries in virtual environments that are both large *and* dynamic. To the best of our knowledge, this problem has not been studied before.

3 Problem Definition

With visibility query, given a query point q the visibility vector of q (i.e., the set of objects visible to q) is returned. Correspondingly, with *approximate* visibility query for q the returned result is guaranteed to be within certain distance from the exact visibility vector of q, with a specified level of confidence. The distance (or alternatively, the similarity) between the approximate and exact visibility vectors is defined in terms of the cosine similarity between the two vectors. In this section, first we define our terminology. Thereafter, we formally define *Approximate Visibility Query (AVQ)*.

Consider a virtual environment $\Omega \subset R^3$ comprising of a stationary environment φ as well as a set of moving objects μ (e.g., people and cars). The stationary environment includes the terrain as well as the static objects of the virtual environment (e.g., buildings). We assume the environment is represented by a TIN model, with which all objects and the terrain are modeled by a network of Delaunay triangles. We consider both static and moving objects of the environment for visibility computation. Also, we assume a query point (i.e., a viewpoint whose visibility vector must be computed) is always at height h (e.g., at eye level) above the stationary environment φ.

Definition 3.1: Given a viewpoint v, the 3D *shadow-set* of v with respect to an object $O \subset \Omega$, $S(v, O)$, is defined as follows:

$$S(v, O) = \{tr | tr \in \Omega,\ tr \notin O,\ \exists p \in tr\ s.t.\ \overline{vp} \cap O \neq \emptyset\} \tag{1}$$

i.e., $S(v, O)$ includes any triangle tr in Ω, for which a straight line $\overline{vp}$ exists that connects v to a point p on tr, and $\overline{vp}$ intersects with O.

Accordingly, we say a triangle t is visible to v, if t is not in the shadow-set of v with respect to any object in the environment. That is, t is visible to v, if we have:

$$t \in \{tr | tr \in \Omega - \bigcup_{\forall O \subset \Omega} S(v, O), dist(v, tr) \leq D\} \tag{2}$$

where *dist* is defined as the distance between v and the farthest point from v on tr, and D is the *visibility range*, i.e., the maximum range visible from a viewpoint.

Note that without loss of generality, we assume boolean visibility for a triangle. Accordingly, we consider a visible triangle as the one which is only fully visible. However, triangle visibility can be defined differently (e.g., a triangle can be considered visible even if it is partially visible) and our proposed solutions remain valid.

Definition 3.2: Given a viewpoint v and an object $O \subset \Omega$, we define the *visibility value* of O with respect to v as follows:

$$vis_O^v = \frac{\sum_{tr \in T_v \wedge tr \in O} Area(tr)}{\sum_{tr \in T_v} Area(tr)} \times \frac{1}{Dist(v, O)} \tag{3}$$

where T_v is the set of all triangles visible to v, and $Dist(.,.)$ is the distance between a viewpoint and the farthest visible triangle of an object. In other words, the visibility value of an object O with respect to a viewpoint v is the fraction of visible triangles to v which belong to O, scaled by the distance between v and O. Intuitively, visibility value captures how visually significant an object is with respect to a viewpoint. Thus, according to our definition of visibility value, nearby and large objects are naturally more important than far away and small objects in terms of visibility. In general, there are many factors (some application-dependent) that can be considered in determining the visibility value of objects (e.g., size of the object, the view angle). Developing effective metrics to evaluate visibility value is orthogonal to the context of our study, and hence, beyond the scope of this paper.

Definition 3.3: For a viewpoint v, we define its *visibility vector* as follows:

$$VV_v = (vis_{O_1}^v, vis_{O_2}^v, ..., vis_{O_n}^v) \tag{4}$$

Visibility vector of v is the vector of visibility values for all the objects $O_i \in \Omega$ with respect to v.

Definition 3.4: Given two viewpoints, v_1 and v_2, the visibility similarity between the two viewpoints is defined as the *cosine similarity* between their visibility vectors as follows:

$$sim(v_1, v_2) = cosim(VV_{v_1}, VV_{v_1}) = \frac{VV_{v_1}.VV_{v_2}}{||VV_{v_1}||\ ||VV_{v_1}||} \tag{5}$$

We say the two viewpoints v_1 and v_2 (and correspondingly their visibility vectors) are $\alpha-$*similar* if:

$$cosim(VV_{v_1}, VV_{v_1}) = \alpha \tag{6}$$

Equally, the *visibility distance* ε is defined based on the similarity α as $\varepsilon = 1-\alpha$. Alternatively, we say the two viewpoints are ε-distant ($\varepsilon = 1 - \alpha$).

Definition 3.5: AVQ Problem

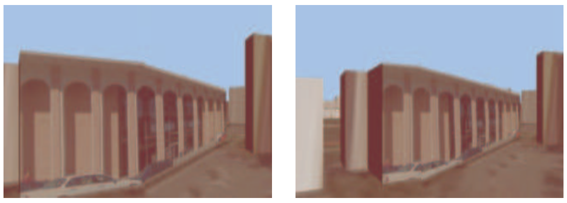
Given a query point q, a visibility distance ε, and a confidence probability p, the *Approximate Visibility Query* returns a vector $A \in \Omega$, such that the visibility vector VV_q of q and A have at least $(1 - \varepsilon)$-similarity with confidence p.

A visual example of AVQ query is shown in Figure 1. Given a query point q, ε=30%, and p=90%, Figure 1a depicts the exact visibility VV_q for q, whereas Figure 1b shows the AVQ result A, which approximates the visibility from viewpoint q with a user-defined approximation error.

4 Solution Overview

To answer AVQs, we develop an index structure, termed *Dynamic Visibility Tree* (*DV-tree* for short). Given a specific visibility distance ε and confidence p, a DV-tree is built to answer AVQs for any point q of the virtual environment Ω. The parameters ε and p are application-dependent and are defined at the system configuration time. Figure 2 depicts an example of DV-tree built for Ω. DV-tree is inspired by our observation that there exists a strong spatial auto-correlation among visibility vectors of the viewpoints. Accordingly, we utilize a spatial partitioning technique similar to quad-tree partitioning to divide the space into a set of disjoint partitions. However, unlike quad-tree that uses spatial distance between objects to decide on partitioning, with DV-tree we consider visibility distance between viewpoints to decide if a partition should be further partitioned into smaller regions. Particularly, we continue partitioning each region to four equal sub-regions until every viewpoint in each sub-region and the representative point of the sub-region (selected randomly) have at least (1-ε)-similarity in visibility with confidence p (see Figure 2). Once DV-tree is constructed based on such partitioning scheme, we also pre-compute and store the visibility vector of the representative point for every partition of the tree. Subsequently, once an AVQ for a query point q is received, it can be answered by first traversing DV-tree and locating the partition to which q belongs, and then returning the visibility vector of the partition's representative point as approximate visibility for q. For example, in Figure 2 the query point q is located at partition P_{31}. Thus, the answer to AVQ for the query point q is the visibility vector $VV_{r_{P_{31}}}$ of the representative point $r_{P_{31}}$ of the partition P_{31}.

However, in a dynamic environment the visibility (i.e., the visibility vectors of the viewpoints) may change as the objects move around. Accordingly, in order to guarantee correct AVQ answering, the pre-computed partitioning with DV-tree must be updated to maintain the distance between the representative point of the partition and the rest of the viewpoints within the partition. To enable efficient update, we propose a two-phase filtering technique which significantly reduces the update cost of DV-tree. In particular, the first phase prunes the unnecessary viewpoints whose pre-computed visibility data do not require update (Section



a) Exact visibility (VV_q) b) Approx. visibility (A)

Fig. 1. Comparing approximate visibility (AVQ with ε=30% and p=90%) with exact visibility for a viewpoint q

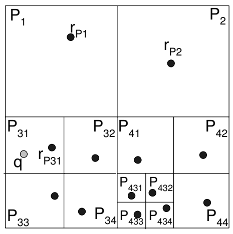
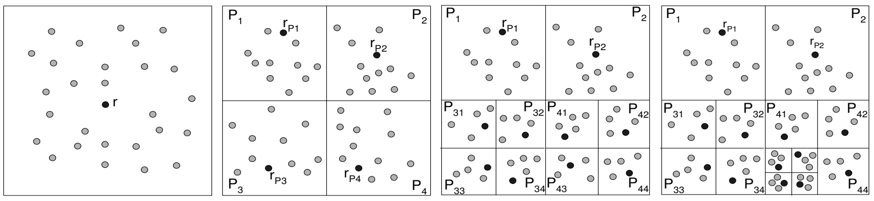


Fig. 2. DV-tree for Ω

6.1), while the second phase prunes the set of objects that can be ignored while computing visibility for the remaining viewpoints (Section 6.2). Sections 5 and 6 discuss the construction and update processes for DV-tree, respectively.

5 Index Construction

In order to construct DV-tree with visibility distance ε and confidence p for a given environment, we iteratively divide the region covered by the environment into four smaller subregions, and for each partition we pick a random representative point, until with a confidence p all the viewpoints inside a partition reside in a visibility distance of less than ε, with the representative point of the partition (Figure 3). With this index construction process, at each step of the partitioning we need to compute the visibility vectors of all the viewpoints of a partition, and then find their visibility distance with the representative point. However, the computation cost for such operation is overwhelming. This stems from the fact that each partition should always satisfy the given ε and p for the approximation guarantee, and such task is hard to accomplish when all the viewpoints of a region are considered. More importantly, continuous maintenance of DV-tree requires repeated execution of the same operation. To address this problem, during each iteration of the DV-tree construction algorithm, instead of calculating the visibility distance for all the viewpoints of a partition, we calculate the visibility distance only for a chosen set of random sample points. In Figure 3, these sample points are marked in gray. Obviously, using samples would result in errors, because the samples are a subset of points from the entire partition. To achieve a correct answer for AVQ, we incorporate this *sampling error* into the approximation that in turn results in a probabilistic solution with confidence p.



a) Step 1 b) Step 2 c) Step 3 d) Step 4

Fig. 3. Illustrating four steps of the DV-tree construction algorithm

In order to compute the sampling error, we first define a few notations. Given a partition with representative point r and assuming a visibility distance ε, we define the probability of success P for the partition as the probability that any random point q inside the partition has a visibility distance less than ε with r. In other words, we have:

$$Pr\Big\{sim(r,q) \geqslant (1-\varepsilon)\Big\} = P \tag{7}$$

During the DV-tree construction, for a given visibility distance ε and confidence p, we say a partition satisfies the given ε and p if its probability of success P is equal to or larger than p (i.e., $P \geqslant p$).

Now, suppose we take n random sample points with replacement from the set of viewpoints inside a partition. We denote N as the number of sample points whose visibility distance to r is less than ε. Our goal is to find out the least value to expect for N, denoted by $\tilde{N}$, such that $P \geqslant p$ is guaranteed with high probability.

Lemma 1. *Given a partition with P as its probability of success, suppose we take n random sample points with replacement from the partition. Then, the random variable N follows a binomial distribution $B(n, P)$ where n is the number of trials and P is the probability of success.*

Proof. The proof is trivial and is therefore omitted due to lack of space. □

Theorem 1. *Given a partition with P as its probability of success and n as the number of samples, the normal distribution $N(nP, nP(1-P))$ is a good approximation for N, assuming large n and P not too close to 0 or 1.*

Proof. Since N has a binomial distribution (i.e., $B(n, P)$), according to the central limit theorem, for large n and P not too close to 0 or 1 (i.e., $nP(1-P) > 10$) a good approximation to $B(n, P)$ is given by the normal distribution $N(nP, nP(1-P))$. □

Consequently, we can approximate the value of $\tilde{N}$ with the normal distribution for n samples and a given *confidence interval* λ. Throughout the paper, we assume a fixed value for the confidence interval λ (i.e., $\lambda = 95\%$). Moreover, any value can be selected for n, as long as the constraint of Theorem 1 is satisfied. To illustrate, consider the following example, where for a given partition with $\varepsilon = 0.2$ and $p = 0.8$, we take 100 random sample points. Using the normal distribution $N(80, 16)$, with $\lambda = 95\%$, we have $\tilde{N} = 87$. This indicates if 87 out of 100 sample viewpoints have visibility distance of less than 0.2 with r, we are 95% confident that $P \geqslant p = 0.8$ holds for that partition.

Figure 4 illustrates the DV-tree that corresponds to the final partitioning depicted in Figure 3d. Each node P_I is of the form $(parent, P_{I1}, P_{I2}, P_{I3}, P_{I4}, Internal)$, where P_{I1}, P_{I2}, P_{I3}, and P_{I4} are pointers to the node's children, *parent* is a pointer to the node's parent and *Internal* captures some information of the current node, which is required for the DV-tree maintenance as we explain in Section 6.

6 Index Maintenance

When an object moves from one location to another, not only the visibility vector of the representative point of a partition but also that of each viewpoint inside the partition might change. This affects the visibility distance of viewpoints inside the partition to the representative point, which consequently could invalidate the approximation guarantee. As a result, DV-tree may return incorrect answers for AVQ. Thus, in order to guarantee correct result, we must update the DV-tree partitioning accordingly. Note that we assume a discrete-time model for the dynamics in the environment; i.e., an object located at location A at time t_0, may move to location B at time $t_0 + 1$. In this section, we propose a two-phase filtering technique which significantly reduces the cost of update: *viewpoint filtering* at phase 1 (Section 6.1) and *object filtering* at phase 2 (Section 6.2). After applying the two-step filtering, the visibility vectors of some of the viewpoints are updated. For each of these viewpoints, the visibility distance with the representative point of its partition might change. Consequently, the representative point might no longer remain as a *correct* representative of the viewpoints inside a partition. Accordingly, the partition must be revised such that a correct AVQ answer is guaranteed. This revision can be either by splitting or by merging the partitions (similar to quad-tree split and merge operations) to guarantee ε and p for the revised partitions. While splitting is necessary for correct query answering, merging only improves the efficiency of query answering. Therefore, to maintain DV-tree we split the partitions (when needed) immediately and merge the partitions in a lazy fashion (i.e., *lazy merge*). In the rest of this section, we explain our two-step filtering technique in more details.

6.1 Viewpoint Filtering

As discussed earlier, in order to efficiently maintain DV-tree, we exploit the fact that examining the visibility of all viewpoints is unnecessary for DV-tree maintenance. The viewpoints are categorized into the following two groups. The first group are those viewpoints that are not included in the sampling during DV-tree construction, and hence, any change in their visibility does not affect the DV-tree maintenance. These viewpoints are filtered out by sampling. This filtering step is performed only once during the DV-tree construction, and therefore we refer to it as *offline viewpoint filtering.* On the other hand, for the set of viewpoints that are sampled, maintaining visibility of *all* the samples with each object movement is unnecessary, because with each object movement only visibility of a subset of the sampled viewpoints changes. Thus, the second group of viewpoints are filtered out from the set of sampled viewpoints because their visibility cannot be affected by a particular object movement. We refer to this step as *online viewpoint filtering.* Below we elaborate on both of these viewpoint filtering steps.

Offline Viewpoint Filtering By maintaining visibility of the sample viewpoints, we need to guarantee that our DV-tree is still properly maintained. Recall from Section 5 that N has a binomial distribution, if n samples are *randomly* selected. During an update of a partition, we need to guarantee that our sampled viewpoints which are stored in the visibility distance histogram VDH, remain valid random samples, and therefore, allow avoiding any resampling. This problem has been studied in [8], where given a large population of R tuples and a histogram of s samples from the tuples, to ensure that the histogram remains a valid representative of the set of tuples during tuple updates, one only needs to consider the updates for the sampled-set s. This guarantees that the histogram holds a valid random sample of size s from the current population. Accordingly, during object movements, when visibility vector of a viewpoint p inside the partition changes, if p is one of the sample points in VDH, its visibility vector is updated. Otherwise, the histogram remains unchanged. In both cases, the randomness of the sample points stored in VDH is guaranteed.

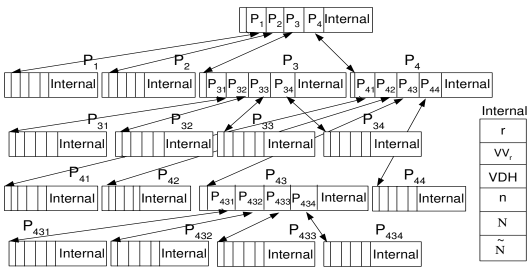


Fig. 4. DV-tree

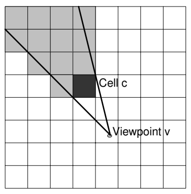


Fig. 5. Potential occluded set of the cell c with respect to viewpoint v ($POS_c(v)$)

Online Viewpoint Filtering As discussed earlier, once an object moves from location A to location B, only visibility vectors of a group of *relevant* viewpoints are affected. These are the viewpoints to which an object is visible when the object is either in location A or location B. With our online viewpoint filtering step, the idea is to pre-compute the set of relevant viewpoints for each point of the space. Accordingly, we propose *V-grid.* With V-grid, we partition the environment into a set of disjoint cells, and for each cell we maintain a list of moving objects that are currently within the cell, as well as a list of all leaf partitions in DV-tree (termed *RV-List*), that either their representative point or any of their sample points is a relevant viewpoint for that cell. When an object moves from one cell to another, we only need to update the visibility vectors of the relevant viewpoints of the two cells.

Our intuitive assumption for pre-computing the relevant viewpoints of a cell with V-grid is that moving objects of an environment (e.g., people, cars) have a limited height, while the static objects such as buildings can be of any height. We denote the maximum height of a moving object as H. In order to build V-grid, we impose a 3D grid on top of the stationary environment φ, where each cell of the grid is bounded with the height H on top of φ. Our observation is that any viewpoint that cannot see the surface of a cell, cannot see anything inside the cell either. Accordingly, all the viewpoints to which any point on the surface of a cell is visible, are considered as the relevant viewpoints of the cell.

6.2 Object Filtering

For each of the relevant viewpoints of the cell in which a movement occurs (i.e., when a moving object enters or leaves the cell), we need to recompute the visibility vector of the viewpoint. However, to compute the visibility vector for each relevant viewpoint, we only need to consider the visibility status of a subset of the environment objects that are potentially occluded by the cell in which the moving object resides before/after the movement. Below, we define how we can identify such objects.

Definition 6.2.1: Given a grid cell c, and a viewpoint v, let c_{xy} and v_{xy} be the 2D projection of c and v on the xy plane, respectively. We define the *potential occluded set* of c with respect to the viewpoint v (denoted by $POS_c(v)$) as the set of all cells, where for each cell $\widehat{c}$, there exists a line segment from v_{xy} to a point s in $\widehat{c}_{xy}$ which intersects c_{xy} in a point p such that the point p lies between the two points v_{xy} and s. The definition can be formulated as follows:

$$POS_c(v) = \{\widehat{c} | \exists s \in \widehat{c}_{xy}, \overline{sv_{xy}} \cap c_{xy} \neq \emptyset, \exists p \in \{\overline{sv_{xy}} \cap c_{xy}\}, |\overline{pv_{xy}}| < |\overline{sv_{xy}}|\} \qquad (8)$$

where $|\overline{pv_{xy}}|$ and $|\overline{sv_{xy}}|$ are the lengths of line segments $\overline{pv_{xy}}$ and $\overline{sv_{xy}}$, respectively.

Figure 5 depicts an example of the potential occluded set of a viewpoint v with respect to a cell c as the gray area. We only need to recompute the visibility values of all the objects that reside in the POS of c with respect to v.

7 Performance Evaluation

We conducted extensive experiments to evaluate the performance of our solution in compare with the alternative work. Below, first we discuss our experimental methodology. Next, we present our experimental results.

7.1 Experimental Methodology

We performed three sets of experiments. With these experiments, we measured the accuracy and the response time of our proposed technique. With the first set of experiments, we evaluated the accuracy of DV-tree answers to AVQ. With the rest of the experiments, we compared the response time of DV-tree in both query and update costs with a competitive work. In comparing with a competitive work, since no work has been found on visibility queries in large dynamic virtual environments, we extended the HDoV approach [15] that answers visibility queries in large static virtual environments, to support dynamism as well. With HDoV-tree, the environment is divided into a set of disjoint cells, where for each cell the set of visible objects (i.e., union of all the objects which are visible to each viewpoint in the cell) are pre-computed, and incorporated into an R-tree-like structure. This spatial structure captures level-of-details (LoDs) (i.e., multi-resolution representations) of the objects in a hierarchical fashion, namely internal LoDs. Consequently, in a dynamic environment, not only the pre-computed visibility data of every cell should be updated, the internal LoDs should be updated as well. In this paper, we do not take into account the objects LoDs[3]. Conclusively, to perform a fair comparison with HDoV-tree, we only consider the cost of updating the pre-computed visibility data of every cell, while ignoring the cost of traversal and update of the tree hierarchy.

Because of the I/O bound nature of our experiments, we only report the response time in terms of I/O cost. Our DV-tree index structure is stored in memory, while the visibility data associated to each node of DV-tree (i.e., *Internal*

[3] Note that employing LoDs is orthogonal to our approach, and can be integrated into DV-tree; however, it is out of the scope of this paper.

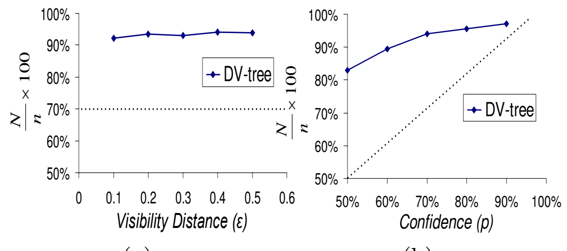
of the node) is stored on disk. Thus, the traversal of DV-tree for point location query is memory-based, and the I/O cost considers only accessing the *Internal* of a node. Moreover, our V-grid is in memory as well, since RV-list of each cell holds only pointers to the relevant partitions.

We used a synthetic model built after a large area in the city of Los Angeles, CA as our data set with the size of 4GB, which contained numerous buildings. We also had a total number of 500 objects, with the maximum height of 1.5 meter (i.e., $H = 1.5m$) moving in the environment. For the movement model, we employed a network-based moving object generator, in which the objects move with a constant speed. Also, we picked 120 random samples during each iteration of DV-tree construction. For visibility computation, we set the visibility range D to 400 meters. Moreover, for visibility query, we ran 1000 queries using randomly selected query points, and reported the average of the results.

The experiments were performed on a DELL Precision 470 with Xeon 3.2 GHz processor and 3GB of RAM. Our disk page size was 32K bytes. Also, we employed an LRU buffer, which can hold 10% of the entire data.

7.2 Query Accuracy

With the first set of experiments, we evaluated the query accuracy of DV-tree. Given a DV-tree with visibility distance ε and a confidence p, we run n queries. Assuming N is the actual number of results satisfying ε, we expect $\frac{N}{n} \geq p$. We first varied the values for ε from 0.1 to 0.5 with p set to 70%. Figure 6a illustrates the percentage of queries satisfying ε for different values of ε. As Figure 6a depicts, for all cases this ratio is in the range of 92% to 95%, which is much higher than our expected ratio of 70% (shown with a dashed line in Figure 6a). Next, we varied the values for p from 50% to 90% with ε set to 0.4, and measured the percentage ratio of $\frac{N}{n}$ accordingly. As Figure 6b illustrates, for all cases this ratio is in the range of 82% to 97%. The dashed line in the figure shows the expected results. According to Figure 6b, the actual results are always above the dashed line. This demonstrates the guaranteed accuracy of our query result.



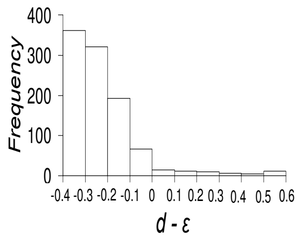


Fig. 6. Query accuracy

Fig. 7. Frequency distribution

The above results show the overall accuracy for a given ε, and p, but they do not represent the distribution of displacement of the query result's distance from the exact answer as compared with ε. For $\varepsilon = 0.4$, and $p = 70\%$, Figure 7 demonstrates the frequency distribution of this displacement (i.e., $d - \varepsilon$). The area under the curve represents the total number of queries, out of which we expect 70% to have a negative displacement. Figure 7 shows 94% query results satisfy this condition. The figure also shows that a large number of query results (68%) are only in distance of less than 0.2 from the exact result.

To summarize, this set of experiments confirms two issues. First, our use of sampling technique during DV-tree construction results in correct AVQ answering. Second, our initial observation about the spatial auto-correlation among the visibility vectors of the viewpoints is valid. This was also illustrated in Figure 1.

7.3 Query Response Time

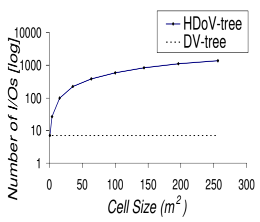
With the second set of experiments, we compared the query response time of DV-tree with that of the extended HDoV-tree by varying the cell size in HDoV-tree. We set the values of ε and p to 0.4 and 70%, respectively. With DV-tree, the query response time is in terms of number of I/Os for retrieving visibility vector of the representative point of the partition to which the query point belongs, whereas with HDoV-tree the query response time is in terms of number of I/Os for retrieving union of the visibility vectors of all viewpoints inside the cell where query point is located. Thus, as the cell size grows, the query response time increases as well. Figure 8 illustrates such results, where the number of I/Os are shown in logarithmic scale. As the figure shows, in all cases DV-tree outperforms HDoV-tree during query answering, except for the case of HDoV-tree with the smallest cell size (i.e., one viewpoint per cell), where the query response time of HDoV-tree is identical to that of DV-tree.

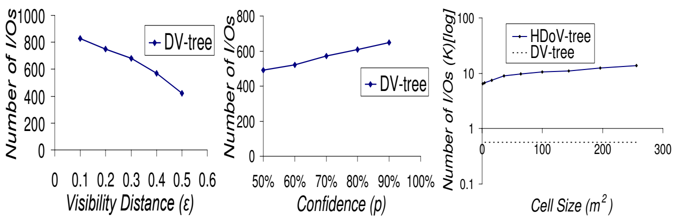
Note that with DV-tree the query result is retrieved as the visibility vector of one viewpoint, which is restricted to the objects inside the viewpoint's visibility range. Conclusively, the query response time is independent of the values chosen for ε and p, as well as the data size.

7.4 Update Cost

The final set of experiments investigates the update cost for maintaining DV-tree. We set the number of moving objects to 50. First, we evaluate the DV-tree update cost by varying the values for ε, where p is fixed at 70%. As Figure 9a depicts, the update cost is in the range of 400 to 800 I/Os. The results show a decrease in I/O cost for large values of ε. The reason is that as the value of ε increases, larger partitions are generated, which leads to fewer updates. Thereafter, we fixed ε at 0.4, and evaluate the DV-tree update cost by varying the values for p. As Figure 9b illustrates, the number of I/Os varies between 500 to 650. Thus, the update cost slightly increases with the growing value of p. The reason is that higher value for p requires more number of viewpoints to satisfy ε in a partition (i.e., N), which results in maintaining more number of viewpoints.

Next, we compare the update cost of DV-tree with that of HDoV-tree. Similar to the previous experiments, we set ε to 0.4, and p to 70%, and vary the cell size for HDoV-tree. Figure 9c illustrates the number of I/Os for both data structures in a logarithmic scale. The significant difference between the two proves the effectiveness of our two-step filtering technique. For HDoV-tree with a small cell size, the number of cells whose visibility should get updated is large. As the cell size grows, the update cost slightly increases. The reason is that although visibility of less number of cells should be updated, retrieving the entire visible objects of a cell is still very costly.





(a) (b) (c)

Fig. 8. Query response time **Fig. 9.** Update cost

Note that the accuracy evaluation in Section 7.2 is a good indicator that a DV-tree with a higher error tolerance allows for reasonable visualization of large-scale and dynamic environments with minor glitches. As Figure 9c depicts, each update of DV-tree with $\varepsilon = 0.4$ and $p = 70\%$ requires 500 I/Os on average. With every I/O taking 10ms on average, the update cost can be estimated as 5 seconds. Note that 5 second update cost is very practical as one can apply many techniques to hide this short delay. For instance, considering the case where trajectory of the moving objects are known a priori, location of the moving objects in the next 5 seconds can easily be predicted. This results in a smooth real-time rendering of the environment. Moreover, for the cases where the objects are constrained to move on predefined paths (e.g., road networks), one can prefetch from disk all the partitions whose visibility might change, and therefore, avoid the extra I/O cost for DV-tree update.

8 Conclusion and Future Work

In this paper, we introduced the novel concept of approximate visibility query in large dynamic environments. Accordingly, we proposed DV-tree for correct and efficient AVQ answering. To enable efficient maintenance for DV-tree, we also proposed a two-phase filtering technique that significantly reduces the update cost for DV-tree. With our experiments, we showed that our observation about the spatial auto-correlation of the visibility vectors is valid. We also demonstrated the overall superiority of our approach as compared to other approaches.

The focus of this paper has been on formal definition of the visibility problem and to propose a framework to address this problem through the use of approximation and DV-tree. For future work, we aim to explore optimization techniques which improve the query efficiency and maintenance cost of DV-tree. For example, while we chose to use a quad-tree-based approach to perform the spatial partitioning, other partitioning techniques exist that might result in a more optimal design for DV-tree. Furthermore, in this paper we make no assumption about movement of the objects. Another direction for our future work would be to use object models with known trajectory or restricted movement options (e.g., cars on roads), and exploit this knowledge to improve the DV-tree update cost.

9 Acknowledgement

This research has been funded in part by NSF grants IIS-0534761, and CNS-0831505 (CyberTrust), the NSF Center for Embedded Networked Sensing (CCR-0120778) and in part from the METRANS Transportation Center, under grants from USDOT and Caltrans. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. J. M. Airey, J. H. Rohlf, and F. P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. *SIGGRAPH*, 24(2):41–50, 1990.
2. H. Batagelo and W. Shin-Ting. Dynamic scene occlusion culling using a regular grid. *Computer Graphics and Image Processing*, pages 43–50, 2002.
3. J. Bittner, V. Havran, and P. Slavk. Hierarchical visibility culling with occlusion trees. In *CGI*, pages 207–219. IEEE, 1998.
4. L. Bukauskas, L. Mark, E. Omiecinski, and M. H. Bhlen. itopn: incremental extraction of the n most visible objects. In *CIKM*, pages 461–468, 2003.
5. S. Chenney and D. Forsyth. View-dependent culling of dynamic systems in virtual environments. In *SI3D*, pages 55–58, 1997.
6. D. Cohen-or, Y. Chrysanthou, C. T. Silva, and F. Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2003.
7. C. Erikson, D. Manocha, and V. B. III. Hlods for faster display of large static and dynamic environments. In *I3D*, pages 111–120, 2001.
8. P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. *ACM Trans. Database Syst.*, 27(3):261–298, 2002.
9. N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *SIGGRAPH*, pages 231–238, 1993.
10. M. Kofler, M. Gervautz, and M. Gruber. R-trees for organizing and visualizing 3d gis databases. *Journal of Visualization and Computer Animation*, 11(3):129–143, 2000.
11. K. C. K. Lee, W.-C. Lee, and H. V. Leong. Nearest surrounder queries. In *ICDE*, page 85, 2006.
12. S. Nutanong, E. Tanin, and R. Zhang. Visible nearest neighbor queries. In *DASFAA*, pages 876–883, 2007.
13. H. Samet. *Foundations of Multidimensional and Metric Data Structures*. 2005.
14. L. Shou, J. Chionh, Z. Huang, Y. Ruan, and K.-L. Tan. Walking through a very large virtual environment in real-time. In *VLDB*, pages 401–410, 2001.
15. L. Shou, Z. Huang, and K.-L. Tan. Hdov-tree: The structure, the storage, the speed. *ICDE*, pages 557–568, 2003.
16. O. Sudarsky and C. Gotsman. Output-senstitive rendering and communication in dynamic virtual environments. In *VRST*, pages 217–223, 1997.
17. S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. *SIGGRAPH*, 25(4):61–70, 1991.