# Optimized Subdivisions for Preprocessed Visibility

Oliver Mattausch[*] Jiří Bittner[*,†] Peter Wonka[‡] Michael Wimmer[*]

[*]Vienna University of Technology [†]Czech Technical University in Prague [‡]Arizona State University

## Abstract

This paper describes a new tool for preprocessed visibility. It puts together view space and object space partitioning in order to control the render cost and memory cost of the visibility description generated by a visibility solver. The presented method progressively refines view space and object space subdivisions while minimizing the associated render and memory costs. Contrary to previous techniques, both subdivisions are driven by actual visibility information. We show that treating view space and object space together provides a powerful method for controlling the efficiency of the resulting visibility data structures.

**CR Categories:** I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Visible line/surface algorithms

**Keywords:** visibility preprocessing, potentially visible sets

## 1 Introduction

Visibility preprocessing is an important method for accelerating real-time walkthroughs of large scale virtual environments. Traditional visibility preprocessing algorithms assume that a *view space* is partitioned into a set of *view cells* and the *object space* is partitioned into a set of *objects*. In a preprocessing step, they determine for each view cell a potentially visible set of objects (PVS). At runtime, only the PVS stored with the view cell containing the viewpoint needs to be rendered, leading to potentially huge savings in rendering time. Visibility preprocessing is applicable even in today's dynamic environments since they still contain a predominant static part.

While there is a great body of literature on how to do the actual visibility calculation for a given view cell, the problem of how to subdivide view space into view cells has received only marginal attention so far [14]. What is even more surprising is that the problem of finding a good object space subdivision for preprocessed visibility has practically not been addressed. Traditional techniques would either use scene triangles, objects specified manually in the modeling phase, or objects defined by traditional object-space partitioning techniques.

It is important to note that both view space and object space subdivision are visibility dependent and also interdependent. The resulting set of objects directly influences the quality of the preprocessed visibility information. If the partitioning is too fine, the memory costs for storing preprocessed visibility will be very high. Additionally the setup costs for rendering the corresponding fine-grained objects on the GPU will become a burden. On the other hand if the partitioning is too coarse, visibility information will be inaccurate, leading to more geometry being rendered than necessary, and therefore slower frame rates.

[*]e-mail: {matt|wimmer}@cg.tuwien.ac.at
[†]e-mail: bittner@fel.cvut.cz
[‡]e-mail: wonka@asu.edu

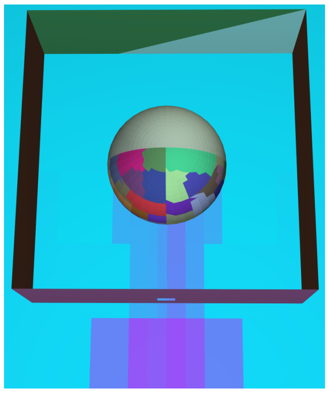


Figure 1: Example of combined view space and object space subdivisions. The scene consists of a sphere inside the room which can be seen only through a small hole. We show a cut through the 3d view space subdivision. The view space subdivision concentrates in the regions which see the sphere through the hole and therefore have higher render cost (low cost=blue, high cost=magenta). The object space subdivision focuses on the front of the sphere which is visible from more view points.

In this paper we present a technique which aims to automatically generate good view space and object space subdivisions based on visibility information. Once these subdivisions are constructed, visibility preprocessing can be carried out using any of the published PVS computation methods. The paper provides two main contributions: 1) This is the first paper to consider object space subdivision as a visibility problem and which provides a solution for visibility driven object splits. 2) This paper presents the first *integrated* visibility-based solution for view space subdivision and object space subdivision.

As a result, the average render cost in the scene can be reduced significantly compared to a naive view space and object space subdivision. We will show that our algorithm consistently improves the render cost at a given memory cost, while naive methods are very fragile with respect to the relative depths of object and view space subdivisions. Figure 1 shows a simple illustrative example: using a non visibility-aware object subdivision method, the spherical in the room would inevitably have been needlessly subdivided uniformly, whereas our method concentrates the subdivision to the front where actual visibility events take place.

## 2 Related Work

View space subdivision techniques have been used in visibility preprocessing from the very beginning, whereas object space subdivision are usually assumed to be already given. For an overview of the actual visibility preprocessing methods we refer the reader to the surveys of Cohen-Or et al. [5] or Bittner and Wonka [4].

### 2.1 View space subdivision

The first visibility preprocessing methods designed for indoor architectural environments [2, 18] partition the scene into cells roughly corresponding to rooms in the building. The cells are connected by portals which correspond to transparent boundaries between the cells. Airey et al. [2] construct a kD-tree taking into account simplified occlusion information. A similar technique was used by Teller and Séquin [18], and was later extended to an autopartition BSP tree [19]. As noticed by Teller [19], in general 3D scenes with non-axial polygons, the subdivision may result in cell fragmentation. This problem was addressed by Meneveaux et al. [15], who use clustering of splitting plane candidates when constructing optimized view cells and portals for building interiors. Further research in that direction aims at short portals [10] or builds on the watershed algorithm [7], while he manual construction of cells and portals during the modeling phase is still considered a valuable option especially for indoor maze-like scenes [11, 1].

Most recent methods for PVS computation do not rely on cells and portals [5]. They compute PVSs by solving the from-region visibility problem while treating the scene geometry as occluders. These methods assume that the view cells are either defined by the user or use simple view space subdivisions like regular grids. However, there are several methods which create view cells during the PVS computation. Gotsman et al. [6] construct a 5D subdivision of view space in which they use sampled visibility to evaluate the efficiency of the candidate splitting planes. The visibility octree of Saona-Vázquez et al. [17] is constructed by a view space subdivision which terminates when reaching a predefined triangle budget or when visibility cannot be reduced by the associated conservative algorithm. Van de Panne and Stewart [20] designed a compression scheme for PVSs computed for a set of view cells. As a side-product of the compression, some cells get merged. Nirenstein and Blake [16] use a hierarchical view space subdivision which is terminated if the desired triangle budget is reached. The triangle budget is determined from the PVS computed for the view cell using adaptive visibility sampling.

Recently, Mattausch et al. [14] gave a deeper analysis of finding a good set of view cells based on actual visibility. Their method aims to minimize the estimated rendering cost for a given view space partition, however it does not address the problem of finding a good object space subdivision.

### 2.2 Object space subdivision

Object space subdivision is crucial for ray tracing, occlusion culling, and collision detection. There are many different techniques optimized for a particular target application. Common techniques include regular grids, octrees, kD-trees, B-kD trees, bounding volume hierarchies (BVH), and hierarchal grids. In particular kD-trees have become very popular for ray tracing acceleration, as they statistically provide the best performance over a large class of scenes [8]. One of the reasons for the success of kD-trees is that they can be easily optimized using a cost model for ray-object intersections called “Surface Area Heuristics” (SAH) [12]. This heuristics assumes a uniform distribution of rays with no occlusion and then performs a greedy optimization of the subdivision. The method achieves very good results for ray tracing applications, however its effect for optimizing the object space subdivision for preprocessed visibility is limited.

Baxter at al. [3] developed a method which clusters geometry in a bottom-up fashion and then refines the clusters by partitioning. This technique was successfully applied for online visibility computations, but it is not obvious how to use the method for optimizing subdivision for preprocessed visibility. Another clustering method for online visibility has been recently developed by Kortenjan and Schomaker [9]. They construct an object space subdivision so that the nodes of the subdivision maintain spatial locality as well as contain geometry of comparable size.

We are not aware of a method which would construct an object space subdivision that is optimized for storing preprocessed visibility. The known visibility preprocessing methods assume that the objects are either (1) identical to the triangles, (2) defined in the modeling phase, or (3) result from a subdivision with some local termination criteria. In the first case the memory consumption of the resulting PVSs for scenes with many triangles and view cells are prohibitive. The second case requires manual modeling; additionally it is not obvious how well the results of modeling will fit the visibility data. In the third case the success of the preprocessing strongly depends on setting the termination criteria of the subdivision: if the partitioning is too fine, the memory costs for storing preprocessed visibility will be very high. If the partitioning is too coarse, there might be significant reserves in the render cost reduction and the desired frame rate will not be reached.

## 3 Outline

The main idea of the proposed algorithm is to acquire coarse visibility information about the scene in a global sampling step, and then use this visibility information to partition view space and object space simultaneously. The proposed method thus consists of two main parts: visibility sampling and interleaved subdivision. *Visibility sampling* acquires visibility information which is represented as a set of maximal free line segments. These segments are then used to quickly determine visibility between the cells of the constructed view space and object space partitions. It is not our goal to compute an exact visibility solution. Instead we want to capture the visibility in the scene in an approximate fashion. The information gained from the sampling is then used to guide the view space and object space subdivision.

The *subdivision* starts with a single view cell representing the whole view space and a single object representing the whole scene geometry. Both the view and object space partitions are progressively refined by splitting either a view cell or an object into two parts. The main criterion driving the splits are the expected render and memory costs which are estimated using visibility samples. Each split attempts to reduce the render cost while keeping the associated memory cost increase as small as possible.

Candidates for splitting are chosen from all current view cells and object space cells. The candidates are stored in a priority queue, and at each step we pick the candidate which provides the best ratio of render cost reduction over memory increase. This candidate is then used to subdivide the associated view space or object space cell. The subdivision proceeds until the given termination criteria are reached. In particular the algorithm terminates if the local render cost reduction falls below a specified threshold, or a maximal memory budget for the whole visibility data is reached. As a result the algorithm delivers optimized view space and object space partitions which can then be fed into any from-region visibility preprocessing algorithm.

The rest of the paper is organized as follows: section 4 presents a theoretical framework for the method. Section 5 discusses the visibility sampling step. Section 6 presents the interleaved subdivision algorithm. The results are summarized in section 7. Finally, Section 7 concludes the paper.

## 4 Framework for interleaved object and view space subdivisions

The interleaved object and view space partitioning which is the core of the proposed method is driven by a cost model which is based on the estimate of the average rendering time and the memory costs needed for storing the visibility information. This section describes a theoretical framework for the rest of the paper. In particular it addresses the representation of the subdivisions, and evaluation of the render cost and the memory cost.

### 4.1 Representing the subdivisions

The input to the algorithm is a subset $VS \subseteq \mathbf{R^3}$ called view space, and a set $OS \subseteq \mathbf{N}$ of object identifiers representing the geometric primitives in the scene. The algorithm operates on *partitions* (or subdivisions) of $VS$ and $OS$, called $\mathscr{V}$ and $\mathscr{O}$ respectively. Initially, $\mathscr{V} = \{VS\}$ and $\mathscr{O} = \{OS\}$. Each step in the algorithm *splits* exactly one cell in either $\mathscr{V}$ or $\mathscr{O}$. An object space split of cell $O = O_1 + O_2$, for example, transforms the partition $\mathscr{O}$ into $\mathscr{O}' = (\mathscr{O} \setminus \{O\}) \cup \{O_1, O_2\}$.

In our implementation, we use an axis-aligned kD-tree to represent the view space subdivision, with the leaf nodes corresponding to the view cells. We decided to use a kD-tree as it is efficient and provides good render cost reduction [14]. To represent the object space subdivision, we use a bounding volume hierarchy. The hierarchy is a binary tree where each node recursively subdivides the associated triangles into two disjoint subsets represented by its two children. The leaves of the hierarchy correspond to the constructed scene objects. Each leaf (object) contains references to scene triangles. Note that each triangle is referred to only once and thus the memory consumption of the data structure is $O(n)$ (internal nodes of the hierarchy only add a constant factor). For each node we also keep an axis-aligned bounding box.

### 4.2 Render cost

The main factor in determining whether a split is beneficial or not is the effect such a split has on the *expected render cost*. For example, a view cell split often causes the child view cells to see less objects than the original cell, so that the render cost is lower in both cells. Similarly, an object space split can cause the resulting sub-objects to be present in the PVS of fewer view cells, similarly reducing the overall rendering costs.

The render cost $c_r$ of a given view space and object space partition $\mathscr{V}$ and $\mathscr{O}$ is given by the expected value of the rendering time over all view cells:

$$c_r(\mathscr{V}, \mathscr{O}) = \sum_{V \in \mathscr{V}} p(V) r(PVS_V), PVS_V \subseteq \mathscr{O} \qquad (1)$$

where $PVS_V$ is an approximate PVS of view cell $V$ consisting of objects from $\mathscr{O}$, $r(PVS_V)$ is a rendering time estimator [21] for this PVS, and $p(V)$ is the probability of the viewpoint being located in view cell $V$. Assuming that viewpoints will be distributed uniformly in the whole view space, $p(V)$ can be chosen as the ratio of the volume $Vol_V$ of the given view cell and the total volume $Vol_{tot}$ of the view space:

$$p(V) = \frac{Vol_V}{Vol_{tot}}.$$

Alternatively, the user can specify any probability density $d$ for viewpoint locations, so that areas where the user is more likely to move receive more attention in the view cell construction. $p(V)$ is then given by:

$$p(V) = \frac{\int_V d(V)}{\int_{VS} d(V)}.$$

The rendering time for a view cell is estimated from the rendering times for the objects in the PVS as seen from view cell $V$:

$$r(PVS_V) = \sum_{O \in PVS_V} \bar{r}(O, V).$$

The rendering time estimation function $\bar{r}(O,V)$ is difficult to establish exactly since it depends not only on the particular set of objects, their attributes and distance to the view cell $V$, but also on the actual implementation and hardware. On the other hand, the view cell subdivision should not be tied too much to a specific hardware, neither do we have an accurate PVS (it is only estimated from a coarse visibility sampling) to determine the absolute value of the rendering time. Therefore we propose to loosely calibrate an analytic rendering time estimation function [21] to a small number of target machines. Since current graphics hardware is CPU limited for small batches, the following function provides good results:

$$\bar{r}(O,V) = \max(a, bt_O, cp_O),$$

where $a$, $b$ and $c$ are positive constants, and $t_O$ and $p_O$ are the number of triangles and the number of projected pixels of object $o$ (estimated from some points in the cell) respectively.

### 4.3 Memory cost

While fine-grained partitions can reduce the rendering cost, they also increase memory costs. Therefore, memory costs need to be controlled. The memory cost $c_m$ of a given set of view cells $\mathscr{V}$ and objects $\mathscr{O}$ is given by:

$$c_m(\mathscr{V}, \mathscr{O}) = M_V + M_O + M_{PVS}, \qquad (2)$$

where $M_V$ is the memory needed to store the view cells, $M_O$ is the memory needed to store the objects, and $M_{PVS}$ is the memory for storing the PVSs, given as follows:

$$M_{PVS} = \sum_{V \in \mathscr{V}} \sum_{o \in PVS_V} m_e \qquad (3)$$

based on the elementary cost $m_e$ of storing one object identifier.

### 4.4 Optimization approach

We cast the partitioning problem as an optimization problem. Let $\{(\mathscr{V}_i, \mathscr{O}_i) : i \in \mathbf{N}\}$ be the set of all possible view space and object space subdivisions resulting from splitting (we assume that there is only a finite number of possible split positions for each cell, and therefore this set is also finite). Then we look for a partition $(\mathscr{V}, \mathscr{O})$ with $c_r(\mathscr{V}, \mathscr{O}) = \min_i(c_r(\mathscr{V}_i, \mathscr{O}_i))$ and $c_m(\mathscr{V}, \mathscr{O}) \leq maxmem$, i.e., we look for the partition with the least render cost given a maximum allowed memory cost *maxmem*.

Enumerating all possible partitions would be prohibitively costly, and we therefore follow an idea borrowed from the well-known Knapsack problem [13]: a common greedy solution to the Knapsack problem always adds the element with the highest "value per size" ratio to the Knapsack. If we interpret the render cost decrease $dR$ of a split as its "value", and the memory cost increase $dM$ of a split as its "size", we can follow the same strategy.

We place all potential split candidates in a *priority queue* and calculate the priority of each candidate by the described ratio, i.e.,

$$p = \frac{dR}{dM}. \tag{4}$$

The optimization algorithm therefore proceeds by repeatedly removing the split with the highest priority from the queue and applying it to the current partition. For the newly generated cells, new potential split candidates are added to the priority queue. This process is repeated until the maximum memory cost is reached. How the potential split candidates are obtained is described in Section 6.1.

Note that the algorithm does not distinguish between object space and view space splits—they are treated equally. The *memory cost increase* consists of a constant overhead of the split (new view cell or new object) on one hand and the number of new PVS entries following from the split on the other hand. For example, when splitting an object, the PVS size of all view cells that see both subobjects will be increased by $m_e$. The evaluation of the *render cost reduction* is described in the next section.

### 4.5 Evaluating the render cost reduction

The crucial part of the algorithm is to evaluate the render cost reduction $dR$ resulting from subdividing either view space or object space.

**View space splits** The render cost of a single view cell is defined as:

$$c_r(V) = p(V)r(PVS_V) \tag{5}$$

When subdividing a view cell $V$ by a splitting plane, the render cost reduction is given by the difference of the render cost of the new view cells and the old one:

$$dR(V) = c_r(V_b) + c_r(V_f) - c_r(V) \tag{6}$$

where $V_b$ and $V_f$ are back and front fragments of the view cell $V$ with respect to the splitting plane. The view space split is illustrated in Figure 2.

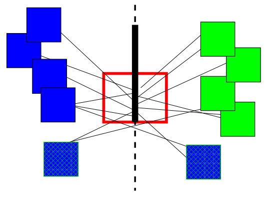


Figure 2: Illustration of a view space split. A view cell (red) is split by a plane which is aligned with certain scene occluder (black). The PVS breaks into two parts seen from the new view cells by the visibility samples. Each part consists of objects seen only by one view cell (blue and green) and objects seen by both view cells (two hatched objects in the bottom).

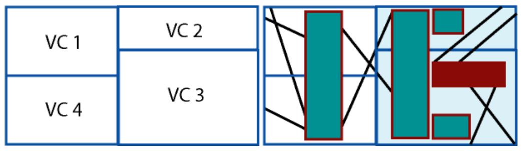


Figure 3: Scene with four view cells (left) and five objects (right). Four objects are shown in blue and one object is shown in red. The render cost of the highlighted object is given by its render time estimate weighted by the sum of area (volume) of all view cells that see the object (VC2 and VC3 shown in light blue).

**Object space splits** To evaluate the render cost reduction when splitting an object, we first need to know from which view cells the object and its new fragments can be seen. We denote the set of view cells which can see an object $O$ as $\overline{PVS}_O$. The expected render cost of an object is then expressed as:

$$c_r(O) = \sum_{V \in \overline{PVS}_O} \overline{r}(O,V)p(V) \tag{7}$$

where p(V) is the probability of view point located in view cell $V$ and $\overline{r}(O,V)$ is the render cost of the object $O$ seen from view cell $V$ (see Figure 3).

When subdividing an object $O$, the render cost reduction is given by:

$$dR(O) = c_r(O_b) + c_r(O_f) - c_r(O) \tag{8}$$

where $O_b$ and $O_f$ are the back and front fragments of the object. The object space split is illustrated in Figures 3 and 4.

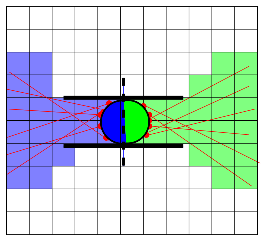


Figure 4: Illustration of an object space split. A spherical object inside a tube-like structure is subdivided into two parts shown in blue and green. In this example the view cells which see the object (identified by visibility samples) break into two disjoint sets.

## 5 Visibility Sampling

Similarly to [14], we gain information about global visibility in the scene by sampling. This section describes how the visibility samples are created and how they are used to estimate PVSs.

### 5.1 Creating visibility samples

Each sample is a line segment which associates all points on the line segment with visibility of the object(s) on its endpoints. Every sample is obtained by casting a ray which starts on an object surface and goes towards the view cells which see the object. The ray origins are distributed uniformly over the object surface, and the ray directions use q cosine weighted distribution over the object surfaces. The resulting visibility samples are line segments which are bound by the ray origin on one side and the intersection point determined by the ray casting algorithm on the other side. If the ray caster reports no intersection, we clip the ray to the view space bounding box. The visibility sample also stores the triangle from which the sample was generated. Thus it provides information from which points in the view space this triangle can be seen.

In the further steps of the algorithm, each sample is associated with an object resulting from the object space subdivision as well as a list of view cells resulting from the view space subdivision.

### 5.2 Estimating PVSs using visibility samples

Every leaf of the view space and object space hierarchies is associated with a set of visibility samples. For a view space node, this list consists of rays that intersect the corresponding view cell. For an object space node, this list consists of rays which are associated with triangles contained in the node. The PVS of a view space node ($PVS_V$) is enumerated as the union of all objects seen by the rays associated with the node. The PVS of an object space node ($\overline{PVS_O}$) is enumerated as the union of all view cells intersected by the associated rays.

In the beginning of the subdivision there are many visibility samples per node and the accuracy of the PVS estimation is relatively high. As the subdivision proceeds, there are less and less samples per node, and consequently the accuracy of the PVS estimation drops. As a result the estimated PVSs in the leafs can be significantly smaller than the real PVSs which would be obtained with an exact visibility solver.

To compensate for this visibility undersampling, we compute an *undersampling factor* for each subdivided node based on the number of rays intersecting the node and the size of the associated estimated PVS. This factor expresses the credibility of the PVSs computed for child nodes using the associated rays. We use this factor to correct the PVS size estimate as well as the render cost estimate for the nodes. The correction is done by blending the values computed using rays with the values determined for the parent node. If the number of rays is significantly larger than the PVS size (every object is sampled by many rays), we use the PVS and render cost without any correction. However, if the number of rays is comparable to the PVS size (every ray sees a different object) we use the PVS size and the render cost determined in the parent node. For cases between these two extremes we linearly interpolate between the values determined by the rays and those computed for the parent node.

Note that this correction is only an estimate, and therefore the total memory cost of the PVS calculated by the actual visibility solver can be larger than the bound *maxmem*.

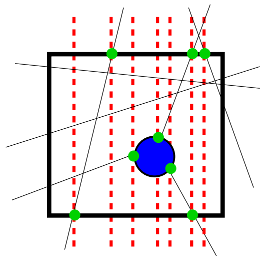


Figure 5: Establishing the split candidate for a view space node. The split planes are placed at the end-points of visibility samples inside the view cell and intersections of visibility samples with the view cell boundary. Then we pick the plane with highest priority as a split candidate for this node. The example shows only the split positions evaluated for one axis.

## 6 Subdivision

This section describes in detail the selection of splitting planes, determination of their processing order, and the computation of the actual splits of view space or object space nodes.

### 6.1 Establishing split candidates

Whenever a new (object space or view space) cell is generated in the subdivision, we establish a new splitting plane candidate for this cell and add it to the priority queue. Within the cell, we aim to find the split plane with the highest priority according to Eq. 4. Both dR and dM functions have discontinuities at places of visibility changes (changes in PVS). The visibility changes can occur only at the end-points of the visibility samples clipped to the current node, i.e., either at an end-point of a visibility sample due to an object within the view cell, or at the intersection of a visibility sample with the view cell boundary. In order to achieve a high chance of separating the PVS, we compute these points and evaluate the split priorities for the corresponding split plane positions in all three axes. The splitting plane candidate for this node is then established at the position with highest priority. The selection of split plane candidate is illustrated in Figure 5.

To efficiently calculate the priorities for all positions, we use a sweeping plane algorithm [14]. We sort the positions and sweep the splitting plane along these positions. At every position of the plane, we compute the PVS incrementally by counting the references to objects seen by rays on both sides of plane. The priority for each position is then computed from the PVS using the formulas given in Section 4.5.

### 6.2 Processing split candidates

The split candidates established for every leaf of the current subdivision have to be processed in the order of descending priority. The basic algorithm described so far would just use a single priority queue of the candidates, pick the candidate with best priority, split the corresponding node and put split candidates for the newly created nodes into the priority queue. However, a view space split induces a change of the priorities of object space split candidates and vice versa. The affected candidates are those which could see the node that was split, i.e., those which are connected with that node with at least one ray. The priorities of the affected candidates have to be reevaluated and their position in the priority queue has to be updated. In the extreme case, a split of a single view space or object space node can affect all of the split candidates from the other domain.

In order to cope with this situation, we exploit the following observation: when performing a split, the priority of other candidates in the queue remains valid if they are from the same domain as the split. This follows from the fact that the cost evaluation of disjoint nodes of the same domain is independent. We use this observation to reduce the number of recomputations of split candidate priorities. The optimized algorithm maintains separate priority queues for split candidates of view space and object space domains and proceeds as follows:

Initially we take the split candidate with the highest priority by comparing the fronts of the view space and object space priority queues. We take at least $n_{min}$ splits from the same domain *without reevaluation* of split candidate priorities. Then we compare the priority of the current split candidate with the best split of the other domain. If the priority of the current split is lower (condition 1), we recalculate the priorities of the candidates and decide whether to switch the domains. Otherwise we continue subdividing in the current domain until condition 1 is met or we reach $n_{max}$ splits (condition 2). $n_{max}$ is a safety criterion ensuring that reevaluation is made with sufficient frequency. As a result the number of steps without reevaluating the split candidate priorities is in the range $(n_{min}, n_{max})$.

Even when processing the splitting planes in batches, the update of all affected split candidates per batch is expensive. The deeper the subdivision, the more candidates have to be updated. To further reduce the number of recomputations of priorities, we only update a subset of split candidates. This subset is chosen randomly from the set of affected candidates.

We found that using a range of $(n_{min}, n_{max}) = (100, 900)$, and updating 2000 candidates per batch gives stable results. The final ratio of view space splits to object space splits is nearly independent from the range of $n_{min}$ and $n_{max}$. The chosen settings are quite conservative, so taking less repair candidates will speed up the algorithm without significantly impacting the efficiency of the algorithm.

### 6.3 Splitting a node

The algorithm for the actual split of the node depends on the domain of the node. If a view space node is subdivided, we split the corresponding cell into two disjoint parts and create two new view cells (leaves of the kD-tree). Then we distribute the associated visibility samples to the new view cells using a line segment / box intersection tests. The visibility samples are used to compute the split candidates for the newly created nodes and their priorities. These split plane candidates are then inserted into the view space priority queue.

If an object space node is subdivided, we partition the triangles of the subdivided cell (object) into two disjoint sets based on the position of their center of mass with respect to the splitting plane. Then we distribute the visibility samples according to the position of the triangles stored with the rays in the child nodes. Similarly as for the view space splits, the visibility samples are used to compute the split plane candidates for the newly created nodes as well as their priorities. These split candidates are then inserted into the object space priority queue.

## 7 Results

We have evaluated the proposed method on two different scenes. The scenes are depicted in Figure 9). The first scene (Vienna) represents 8km$^2$ of the city of Vienna. The second scene (Arena) is a model of Sazka-Arena—a multipurpose stadium in Prague (courtesy of Digital Media Production a.s.). The Vienna scene consists of houses with windows and balconies, pavements, and roads. The Arena scene represents both the interior and the exterior of the stadium. The interior has high depth complexity (apart from the large open spaces, the building contains about 1000 rooms) and includes many instances of coarsely modeled objects such as chairs and tables. The Vienna scene contains 1M triangles, the Arena scene 1.5M triangles.

For both scenes we measured the dependence of the render cost estimate on the memory cost of the constructed subdivisions. As a reference for comparison, we used a method which initially subdivides object space using the surface area heuristics up to a specified number of objects, and then applies the visibility-driven view cell construction [14] to subdivide view space. We used different termination criteria of the initial object space subdivision (10k and 60k objects) to observe the dependence of the reference method on this parameter.

For both test scenes the subdivisions were constructed with 3M, and 6M visibility samples, respectively. We terminated the subdivisions when they reached a specified memory budget (35MB). After the construction of the subdivisions, we also evaluated the quality of the subdivision by casting a large number of visibility samples and recomputing the render cost and memory cost curves. We used 160M evaluation samples for both scenes. The results of these measurements are summarized in Figures 6 and 7. Our new method is labeled INT (interleaved) and the reference method is labeled SEQ (sequential) together with a number expressing the number of objects in the initial object space partition. We used a 2.2 GHz Dell Inspiron 9300 notebook with 2GB RAM for computing the results.

To analyze the behavior of the compared methods, we measured the ratio of the total number of view cells and objects during the subdivision process (Figure 8-(left)). This ratio is monotonically growing for the reference methods, whereas for the new method it adapts to the visibility properties of the scene. Interestingly, after a steady falloff the ratio increases again at the end of the subdivision.

The middle and rightmost plots in Figure 6 and Figure 7 show the evaluation of the subdivisions using 160M samples. We can see that the number of samples has significant influence on the results. Our method profits from increasing the number of samples from 3M to 6M in both scenes. On the other hand, the reference methods are sensitive to the larger number of samples only in Arena. It is hard to tell which would be the optimal number of samples for a certain scene. Our experiments indicate that with 6M samples, satisfactory results can be achieved for typical scenes such as the ones shown.

The results show that with a sufficient number of visibility samples, the proposed method provides a consistent reduction of the average render cost at any given memory cost when compared to the reference methods. The render cost reductions for selected memory budgets are summarized in Table 1. Compared to the reference methods we obtained speedups of 16–68% for Vienna and 32–153% for Arena. Note that the speedup is based on the *average* render cost, a quantity which is smoothed over all view space, whereas locally the speedup can be much higher. For example, both scenes contain some large irreducible view cells which prevent large variations in the average render cost, therefore the speedups shown are quite significant. This fact is illustrated in Figure 8-(right), which depicts the histogram showing the distribution of render cost over the view

| Scene | Method | Time [m] | Memory [MB] | View cells | Objects | Render cost | Speedup |
|---|---|---|---|---|---|---|---|
| Vienna | SEQ-10k | 25 | 60 | 11000 | 10000 | 81999.2 | 1.67 |
| | SEQ-60k | 21 | 60 | 3000 | 60000 | 58149.2 | 1.18 |
| | INT | 212 | 60 | 3106 | 60894 | 49221.9 | – |
| Arena | SEQ-10k | 33 | 80 | 15500 | 10000 | 24631.4 | 2.53 |
| | SEQ-60k | 29 | 80 | 5500 | 60000 | 12825.7 | 1.32 |
| | INT | 106 | 80 | 5305 | 57195 | 9745.91 | – |

Table 1: Summary of results for Vienna and Arena scenes.

space volume for Vienna. It can be seen that for the new method, significantly more volume is covered by lower render cost. We can see from Table 1 that our method requires several times more computation time than the sequential methods. However, to find a good subdivision using a sequential method, it would have to be evaluated several times for different numbers of objects, which would be even more costly. Most of the computation time of our method is spent evaluating local split planes for the view space splits, the cost of which highly depends on the number of visibility samples. A viable alternative is to use only mid splits, which speeds up computations a lot.

We have presented a new tool which allows constructing an optimized subdivision of view space and object space for preprocessed visibility. The method treats view space and object space partitioning together and it progressively refines both subdivisions while minimizing the associated render and memory costs. Contrary to previous techniques, both subdivisions are driven by the actual visibility in the scene. This allows a better adaptation to the visibility distribution within the view space and the object space.

In the future we would like to improve our model of render cost. In particular we want to integrate the cost of state changes due to materials and textures in the algorithm. Further we want to deal with the issue of determining how many samples to use, by implementing an adaptive version that starts with a low number of samples and casts more samples on demand. We are also working on an algorithm which uses the scalability of the subdivisions for rapid sampling-based global visibility precomputation.

## Acknowledgments

This work has been supported by the EU under the project no. IST-2-004363 (GameTools) and by the Ministry of Education, Youth and Sports of the Czech Republic under the research program LC-06008 (Center for Computer Graphics). The Arena scene is a courtesy of Digital Media Production a.s.

## References

[1] Timo Aila. Surrender umbra: A visibility determination framework for dynamic environments. Master's thesis, Helsinki University of Technology, 2000.

[2] John M. Airey, John H. Rohlf, and Frederick P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. In *1990 Symposium on Interactive 3D Graphics*, pages 41–50. ACM SIGGRAPH, March 1990.

[3] William V. Baxter III, Avneesh Sud, Naga K. Govindaraju, and Dinesh Manocha. GigaWalk: Interactive walkthrough of complex environments. In Simon Gibson and Paul Debevec, editors, *Proceedings of the 13th Eurographics Workshop on Rendering (RENDERING TECHNIQUES-02)*, pages 203–214, Aire-la-Ville, Switzerland, June 26–28 2002. Eurographics Association.

[4] Jiří Bittner and Peter Wonka. Visibility in computer graphics. *Environment and Planning B: Planning and Design*, 30(5):729–756, sep 2003.

[5] D. Cohen-Or, Y. Chrysanthou, C. Silva, and F. Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics.*, 2002.

[6] Craig Gotsman, Oded Sudarsky, and Jeffrey A. Fayman. Optimized occlusion culling using five-dimensional subdivision. *Computers and Graphics*, 23(5):645–654, October 1999.

[7] Denis Haumont, Olivier Debeir, and Franois Sillion. Volumetric cell-and-portal generation. *Computer Graphics Forum*, 22(3):303–312, September 2003.

[8] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[9] Michael Kortenjan and Gunnar Schomaker. Size equivalent cluster trees (sec-trees) realtime rendering of large industrial scenes. In *Afrigaph '06: Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 107–116, New York, NY, USA, 2006. ACM Press.

[10] A. Lerner, D. Cohen-Or, and Y. Chrysanthou. Breaking the walls: Scene partitioning and portal creation. In *Pacific Graphics*, 2003.

[11] David Luebke and Chris Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In Pat Hanrahan and Jim Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 105–106. ACM SIGGRAPH, April 1995.

[12] J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(6):153–65, 1990. criteria for building octree (actually BSP) efficiency structures.

[13] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons Inc., New York, 1990.

[14] Oliver Mattausch, Jiří Bittner, and Michael Wimmer. Adaptive visibility-driven view cell construction. In Wolfgang Heidrich and Tomas Akenine-Moller, editors, *Rendering Techniques 2006 (Proceedings of the Eurographics Symposium on Rendering 2006)*, pages 195–206. Eurographics, Eurographics Association, June 2006.

[15] Daniel Meneveaux, Kadi Bouatouch, Eric Maisel, and R. Delmont. A new partitioning method for architectural environments. *Journal of Visualization and Computer Animation*, 9(4):195–213, 1998.

[16] S. Nirenstein and E. Blake. Hardware accelerated aggressive visibility preprocessing using adaptive sampling. In *Rendering Technqiues 2004*, pages 207–216, 2004.

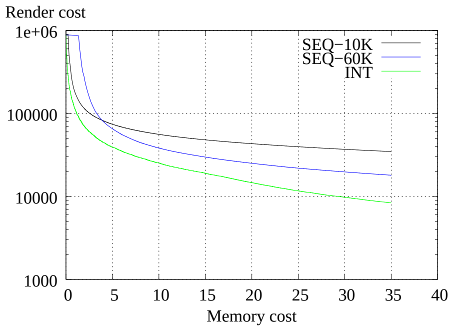
[17] C. Saona-Vázquez, I. Navazo, and P. Brunet. The visibility octree: a data structure for 3D navigation. *Computers and Graphics*, 23(5):635–643, October 1999.

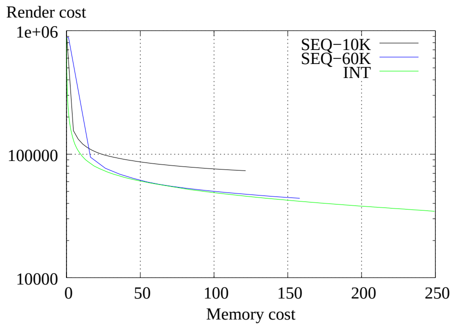
[18] S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. In *Proceedings of SIGGRAPH '91*, pages 61–69, July 1991.

[19] Seth Jared Teller. *Visibility Computations in Densely Occluded Polyhedral Environments*. PhD thesis, Dept. of Computer Science, University of California, Berkeley, 1992. Also available as Technical Report UCB//CSD-92-708.

[20] Michiel van de Panne and A. James Stewart. Effective compression techniques for precomputed visibility. In *Rendering Techniques*, pages 305–316, 1999.

[21] Michael Wimmer and Peter Wonka. Rendering time estimation for real-time rendering. In *Rendering Techniques*, pages 118–129, 2003.





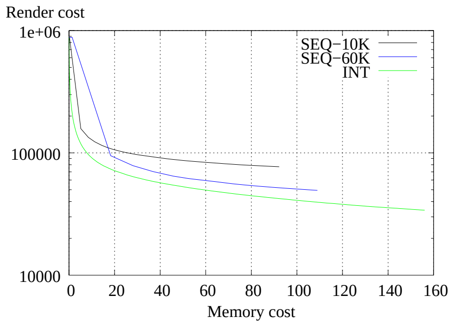
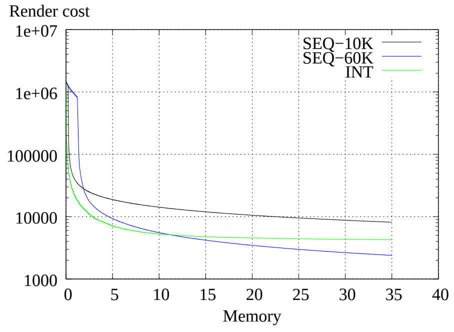
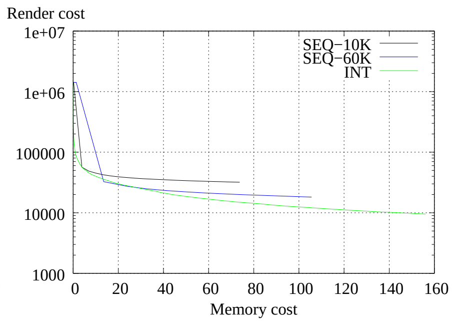


Figure 6: Render cost vs. memory cost curves for Vienna. (left) Original curves, 6M samples. (center) 3M samples, measured using 160M evaluation samples. (right) 6M samples, measured using 160M evaluation samples.





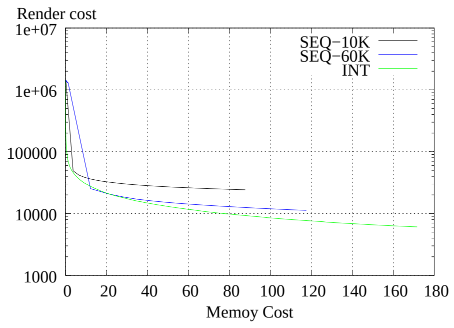
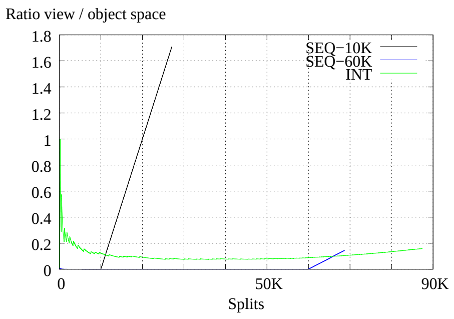


Figure 7: Render cost vs. memory cost curves for Arena. (left) Original curves, 6M samples. (center) 3M samples, measured using 160M evaluation samples. (right) 6M samples, measured using 160M evaluation samples.



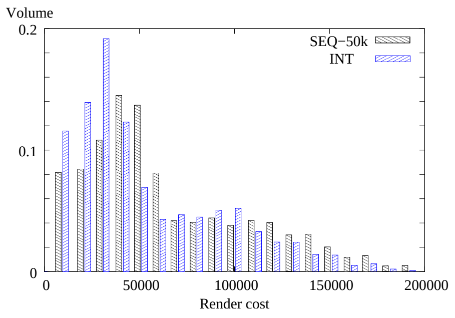


Figure 8: (left) Evaluation of the ratio of the total number of view cells and objects during the subdivision of Arena. (right) Histogram showing amount of view space volume in a particular range of render cost for the subdivision of Vienna.

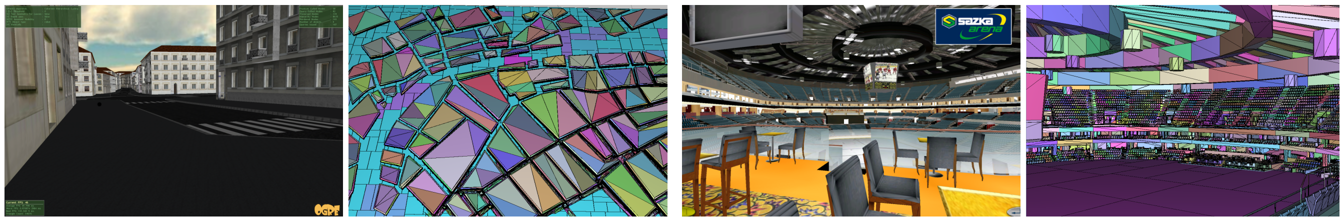


Figure 9: Snapshot of Vienna scene (leftmost) and visualization of the view / object space partition (second from left). Snapshot of Arena scene (second from right) and visualization of the view / object space partition (rightmost). Each colored patch represents one object from the PVS. We show a cut through the 3d view space subdivision. The view cells are colored from blue (low render cost) to magenta (high render cost).