Eurographics Symposium on Rendering (2006)
Tomas Akenine-Möller and Wolfgang Heidrich (Editors)

Adaptive Visibility-Driven View Cell Construction

Oliver Mattausch[1], Jiří Bittner[1,2], Michael Wimmer[1]

[1]Vienna University of Technology, Austria
[2]Czech Technical University in Prague, Czech Republic

Abstract

We present a new method for the automatic partitioning of view space into a multi-level view cell hierarchy. We use a cost-based model in order to minimize the average rendering time. Unlike previous methods, our model takes into account the actual visibility in the scene, and the partition is not restricted to planes given by the scene geometry. We show that the resulting view cell hierarchy works for different types of scenes and gives lower average rendering time than previously used methods.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling

1. Introduction

Visibility preprocessing is an important tool in many application areas, for example for achieving interactive walkthroughs of large scale virtual environments. Traditional visibility preprocessing algorithms assume a *view space* that is partitioned into a set of *view cells*. 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.

While there is a huge body of literature on how to calculate a PVS for a given view cell, the problem of how to actually find the view cells has received only marginal attention so far. This is surprising, since the selection of view cells is crucial for several reasons: (1) view cells that take the visibility structure of the scene into account allow achieving smaller PVSs, and therefore faster rendering speed at runtime, (2) a bad view cell definition can severely impact the time required for the preprocessing step, and (3) the amount of memory required to store the PVS data depends strongly on the quality of the view cell distribution.

Consider, for example, a model of a city with buildings of different heights. A naive subdivision method that concentrates on the 2D layout of the city might end up with most view cells seeing the whole city, because the upper parts of all view cells extend over the roof tops. Clearly we would like to separate these regions from those which see only the nearby streets. This shows that even for relatively simple types of scenes, important visibility changes can occur due to the height structure of the model.

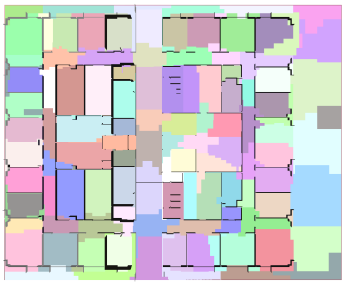

Figure 1: *A 2D cut through a set of view cells constructed with our method for the Soda hall building. Note how the shape of view cells adapts to loci of visibility events.*

The only published methods dealing with this problem are either based on very simple regular top-down subdivision schemes [SVNB99, vdPS99, NB04], or treat only special types of scenes [ARB90, TS91, HDS03, LCOC03], while in most practical applications (e.g., computer games), the difficulty of view-cell construction is one of the factors that

hinders the wide-spread use of visibility preprocessing methods for general scenes. The finest possible subdivision is given by the aspect graph, which partitions the view space into cells from which the qualitative aspect of the scene does not change [PD90]. However, the full aspect graph requires as much as $O(n^9)$ nodes for perspective views (n is number of polygonal edges) and is therefore prohibitively expensive to compute and to store.

In this paper we provide an adaptive view cell construction technique for general scenes which takes the actual visibility structure of the scene into account. The technique relies on four novel contributions: (1) the visibility of the scene can be approximated by an inexpensive stochastic sampling step, (2) the view cell construction is driven by the average rendering time of the resulting partition, which is ultimately the most important factor in visibility processing, (3) an optimized set of view cells is found by combining top-down subdivision and bottom-up merging steps, and (4) our method allows easy control over the subdivision by using intuitive global termination criteria such as a threshold on rendering time reduction or a maximal memory budget. Another novelty is that we do not construct a fixed set of view cells, but provide a hierarchy of view cells, which makes it easy to extract an optimized set of view cells for a given set of constraints. As a result, our view cell subdivision provides fast rendering times with a minimal set of view cells, thus saving both preprocessing time and PVS storage space.

2. Related Work

The first visibility preprocessing methods were designed for accelerating walkthroughs of indoor architectural environments [ARB90, TS91]. These methods 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. [ARB90] defined a set of rules which the view space subdivision algorithm should satisfy. They construct a kD-tree hierarchy where the subdivision planes are aligned with scene polygons. For each candidate plane they compute its priority as a weighted sum of its occlusion properties and the estimated balance and size of the tree. A similar technique was used by Teller and Séquin [TS91], and was later extended to an auto-partition BSP tree [Tel92].

As noticed by Teller [Tel92] in general 3D scenes with non-axial polygons, the subdivision may result in cell fragmentation. This problem was addressed by Meneveaux et al. [MBMD98] who focused on building interiors. In the first step they extract the floors of the building and in the second step they use a 2D method to partition each floor separately. The 2D method clusters candidate splitting planes in dual space to find those planes which provide the best fit to the walls of the building.

Despite the research on constructing cell and portal graphs the manual construction of cells and portals during the modeling phase is still considered a valuable option especially for indoor maze-like scenes [LG95, Ail00].

Recently, Lerner et al. designed an algorithm which aims to create short portals [LCOC03]. The algorithm is suitable for 2D scenes and 2.5D scenes with buildings of comparable height. The authors also present a cost model for evaluating the efficiency of the resulting partition. Using this cost model, it is shown that the method delivers superior partitions compared to previous BSP tree based algorithms.

Haumont et al. [HDS03] used a significantly different strategy for constructing a cell/portal graph. They use a voxelization of the scene and the watershed algorithm computed on a distance field. The method grows cells from local minima of the distance field and introduces portals when two cells meet. Similarly to the method of Lerner et al. [LCOC03], this approach uses a top-down subdivision (voxelization) as well as a localized bottom-up cell construction (watershed).

All the methods mentioned above deal with the construction of cell and portal graphs. However the PVS concept has been used by numerous methods which do not need portals for PVS computation [COCSD02]. Most of these methods focus only on the PVS determination step, i.e., computing from-region visibility. They assume that the view cells are either defined by the user or use a simple view space subdivision without further considerations.

However, there are several methods which indirectly deal with the problem of finding good set of view cells. Additionally, unlike the above mentioned cell and portal methods, they make use of the actual visibility information [GSF99, SVNB99, vdPS99, NB04]. Gotsman et al. [GSF99] 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. [SVNB99] 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 [vdPS99] designed a compression scheme for PVSs computed for a set of view cells. As a side-product of the compression, some cells get merged. Similarly to Saona-Vázquez et al. [SVNB99], Nirenstein and Blake [NB04] 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 aggressive visibility sampling. The view cell determination in all these methods is driven by rather simple models which consider only PVS set differences. Additionally the methods of Gotsman et al., Saona-Vázquez et al. and Nirenstein and Blake perform only top-down view space subdivision, which need not adapt well to local visibility changes. On the other hand the method of van de Panne and Stewart performs only bottom-up con-

struction, which does not permit using a larger number of initial view cells.

Our paper aims to give a deeper analysis of finding a good set of view cells based on actual visibility. We provide a new cost model for evaluating the efficiency of the constructed view cells. The model is based on the estimated rendering cost for a given view space partition. Our method does not provide only a fixed set of view cells. Instead it constructs a novel form of view cell hierarchy from which we can extract an optimized set of view cells for a given memory budget.

3. Overview

3.1. What is a good view cell partition?

It is instructive to think about what criteria determine a "good" view cell partition. Since the ultimate goal of visibility preprocessing is to accelerate rendering, the runtime *rendering costs* will play an important role. The intuitive answer is that the partition should *minimize* the rendering costs at runtime for each possible viewpoint. A view cell subdivision corresponding to this criterion actually exists and is given by the aspect graph. This structure is fully determined by the visibility structure of the scene, namely by the so-called visual events (boundaries at which changes in visibility occur). However, this subdivision would be prohibitively expensive to compute and to store.

This leads to *PVS storage space* and *precomputation time* as further important criteria. Both of these are determined by the total number of view cells in the subdivision, the actual visibility, the visibility algorithm, and the PVS storage method used: output-sensitive visibility algorithms can make computation time sublinear in the number of view cells, and PVS compression schemes can significantly reduce the required storage space.

An alternative to minimizing runtime rendering costs is to specify a rendering budget, i.e., a *maximum rendering cost* for a view cell. However, a model can contain an arbitrary number of viewpoints for which this rendering budget cannot be met. This can easily lead to excessive subdivision in areas with unrestricted visibility, and therefore again to high storage costs and precomputation time. Furthermore, such a view cell partition will be strongly tied to a particular runtime system, which is undesirable for a preprocessing algorithm.

We therefore propose the *average rendering cost* of the whole view space as the criterion to drive the view cell construction, because it can be well combined with a constraint on the number of view cells in order to limit PVS storage costs and precomputation time.

Another question is the shape of the candidate view cells that will be subjected to the above criteria. One option is a regular subdivision (e.g., a kD-tree of the scene). However, the boundaries of such view cells will not coincide with actual changes in visibility, i.e., visual events. On the other hand, finding these visual events (e.g., using the aspect graph) is expensive. We therefore propose a two-tier approach: an initial subdivision will follow (but not be restricted by) the geometry in the scene. Cells of this initial subdivision will subsequently be merged according to the visibility information in the scene. This causes the final view cells to approximate the important visual events in the scene.

3.2. Algorithm outline

Our view cell construction method consists of three main steps:

1. Visibility sampling
2. View space subdivision
3. View space merging

The first step estimates visibility in the scene, which allows basing the view cell construction on scene visibility without incurring the overhead of having to calculate complete visibility. We use stochastic sampling by casting rays distributed in the whole view space. As a result we obtain a set of maximal free line segments which we call *visibility segments*. The visibility segments provide information about scene visibility for the subsequent steps of the view cells construction.

The second step performs an adaptive hierarchical subdivision of view space. The subdivision is driven by heuristics which aim to minimize the estimated rendering cost of the resulting subdivision. The result is a set of elementary view cells which satisfy certain global termination criteria (maximal memory budget or minimal render cost reduction).

The third step merges the elementary view cells to larger ones while minimizing the increase of the estimated rendering cost for each merging step. The merging progress is recorded in a *merge history tree*. The initial subdivision and the merge history define a *view cell hierarchy*, which allows retrieving an optimal set of view cells for a specified granularity of the view space subdivision. Additionally, this hierarchy can be used to compress the PVSs in a simple and efficient way. The merging step will implicitly approximate important visibility events in the scene. The three steps of our algorithm are illustrated in Figure 2.

3.3. Cost Model

Our view cell construction is driven by a cost model which estimates the average rendering time based on the sampled visibility information.

The cost of a given set of view cells $\mathcal{S}$ is given by the expected value of the rendering time as follows:

$$c_r(\mathcal{S}) = \sum_{i \in \mathcal{S}} p(i) r(PVS_i), \tag{1}$$

Figure 2: *(left) Visibility segments determined by the visibility sampling step; 5000 of 1M line segments are depicted for sake of clarity. (center) Initial view space subdivision consisting of 5000 view cells. (right) 200 view cells retrieved from the merge history tree.*

where $r(PVS_i)$ is a rendering time estimator [WW03] for the approximate PVS of cell i, and $p(i)$ is the probability of the viewpoint being located in view cell i. Assuming that viewpoints will be distributed uniformly in the whole view space, $p(i)$ can be chosen as the ratio of the volume V_i of the given view cell and the total volume V of the view space:

$$p(i) = \frac{V_i}{V}.$$

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. $p(i)$ is then given by:

$$p(i) = \frac{\int_{x \in i} d(x)}{d(x)}.$$

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

$$r(PVS) = \sum_{o \in PVS} r(o).$$

The rendering time estimation function $r(o)$ is difficult to establish exactly since it depends not only on the particular set of objects and their attributes, 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 to determine the absolute value of the rendering time. Therefore we propose to loosely calibrate an analytic rendering time estimation function [WW03] to a small number of target machines. Since current graphics hardware is CPU limited for small batches, the following function provides good results:

$$r(o) = \max(a, bt_o, cp_o),$$

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.

3.4. Representation of View Cells

We maintain the view space partition as a binary space partition tree which is constructed top-down. The leaves of this tree are convex polyhedra which form a set of elementary view cells. The final view cell partition is constructed from these elementary view cells using a bottom-up merging procedure which is recorded in the merge history tree. Note that merging is not tied to the original subdivision and therefore usually results in a different, more optimal tree. Both steps of the view cell hierarchy construction will be detailed in the next section. The representation of view cells using the two hierarchies is shown in Figure 3.

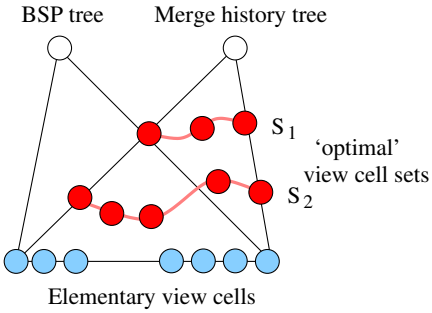

Figure 3: *The view cell hierarchy is represented using a BSP tree and the merge history tree. The BSP tree provides a geometrical description for elementary view cells. The merge history tree provides a logical grouping of the view cells, which allows extracting a set of view cells with a specified granularity of the subdivision. The example shows two sets $\mathcal{S}_1$ and $\mathcal{S}_2$, where the desired number of view cells for $\mathcal{S}_2$ is larger than for $\mathcal{S}_1$.*

Note that we partition the view space only in the spatial domain, since the observer can quickly move through the whole directional space within a few frames by changing the viewing direction. Fortunately, culling in directional space is efficiently handled by view-frustum culling.

4. Adaptive View Cell Construction

4.1. Visibility Sampling

We gain information about global visibility in the scene by sampling the whole view space. A view space sample is a 5D entity corresponding to a ray in primal space. For simplicity, let's assume that the view space is defined by a 3D spatial box of possible ray origins (*view space box*) and contains all possible ray directions. The view space is then sampled using the following strategy:

1. Select a point p inside the view space box and a direction d using uniform distributions [CLF98].
2. Cast a 'forward' ray from p in direction d and a 'backward' ray from p in the opposite direction $-d$.
3. Construct a line segment formed by the calculated termination points of the forward and backward rays. If at least one of the two rays hits an object, we call the resulting line segment a visibility segment and store it for later use.

The determination of visibility segments is illustrated in Figure 4.

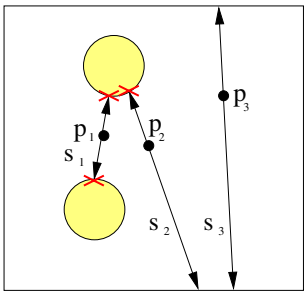

Figure 4: *Illustration of determination of visibility segments. Three visibility samples are generated from points p_1, p_2 and p_3, resulting in two valid visibility segments. s_1 carries information about visibility of two objects, s_2 about one object. s_3 is not a valid visibility segment since it does not hit any object.*

Note that although the ray casting is performed using all scene polygons, we use the objects (i.e., logical groups of polygons) for representing visibility. The PVS for a view cell is computed as a union of objects associated with the visibility segments intersecting the cell.

There is an interesting subtlety involved in polygon orientations. For a general scene, the above procedure will create visibility segments in the interiors of objects if those regions are not explicitly exempted from view space. If, however, the input model is guaranteed to be watertight and the polygons have a consistent orientation, the algorithm can detect *empty view space* at no additional cost: if a ray hits the back side of a polygon, the ray starting point is simply shifted to the intersection point and the ray is re-cast (see Figure 5). In this way, no visibility segments will be generated in empty space. We have found that empty view space detection can improve the final view cell hierarchy, because visibility regions are more clearly separated.

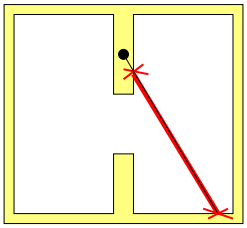

Figure 5: *Example of empty view space detection. The origin of the ray lies inside the wall of the building and so the ray first hits a back-facing polygon. We shift the ray origin to the intersection point and the ray is re-cast. The resulting visibility segment is shown as a thick line.*

4.2. View Space Subdivision

The view space subdivision uses a top-down approach to create a set of elementary view cells. In particular we use *binary space partitioning* (BSP) maintained by a BSP tree. Starting with a single view cell corresponding to the whole view space, we recursively subdivide the current view cell using either axis-aligned planes or planes aligned with scene geometry. Each cell of the subdivision also references all visibility segments that intersect it.

The BSP construction uses greedy optimization for the next-best split. Note that in contrast to most previous work on BSP or kD-tree construction, we employ a priority queue for selecting the splitting plane candidates. An entry in the priority queue consists of a reference to a leaf node, the best splitting plane candidate inside this leaf, and a cost reduction which would be achieved when splitting the leaf by the plane. For each step of the subdivision, we select the node for which its best splitting plane provides the highest render cost reduction. The subdivision is thus refined progressively and regions with the highest potential render cost decrease are subdivided first.

The best splitting plane for a node is established as follows: we generate axis-aligned splitting plane candidates aligned with the endpoints of rays intersecting the node. Additionally, we generate planes aligned with the geometry contained in that node (if any). For each candidate, we calculate the reduction of the expected rendering cost $c_r(\mathcal{S})$

that would result from subdividing the node by that plane. This is done by partitioning the current set of visibility segments according to the plane (note that a segment can be assigned to both sets), computing the PVSs (i.e., the union of the objects hit by the visibility segments) for the front and back segment sets, and using those to evaluate the reduction of the cost. Finally, we choose the candidate plane that provides the most reduction in expected rendering cost and put a corresponding entry in the priority queue.

The subdivision is terminated when one of the following termination criteria is met:

- A specified maximum number of elementary view cells have been generated. This ensures that the algorithm stays within reasonable memory bounds.
- The cost reduction for the best splitting plane is below a specified threshold. As the cost reduction can temporarily stagnate, we only terminate when the reduction was below the threshold in several successive subdivision steps.

Due to the priority-driven subdivision the view space will be evenly subdivided regardless of the termination point. This is not the case for depth-first approaches, where for example a termination on low memory would leave whole view space regions unsubdivided. Additionally, the local termination criteria used in the depth-first approach are hard to tune.

We experienced that in some rare cases the described greedy optimization can lead into a local minimum. Consider the situation that the render cost of a node cannot be reduced by the current split, but only by subsequent splits of the child nodes. However, the node is never chosen for subdivision because it provides no render cost decrease. We have addressed this problem by computing a weighted sum of the render cost reduction and the absolute render cost of the node in the evaluation of its priority. A very small weight of the absolute render cost (1%) has been sufficient to ensure that the split is taken at some point during subdivision.

Note that the time required for the subdivision is dominated by the cost evaluation for the candidate planes. To accelerate this process, we limit the number of axis-aligned as well as geometry-aligned candidates. If the number of visibility segments or geometry planes is above these limits we select their random subsets. This speeds up the selection especially for nodes near the root of the subdivision tree.

4.3. View Space Merging

View space merging is a bottom-up process which aims to reduce the number of view cells while minimizing the cost of the merged view cell set. We use a greedy algorithm that always merges the pair of view cells resulting in the minimal cost increase. This is done by maintaining a priority queue of view cell merge candidates. Each pair of neighboring view cells forms a merge candidate. The cost increase due to the merge candidate consisting of view cells x and y is given as:

$$\Delta_{c_r}(x,y) = c_r(\{x \oplus y\}) - c_r(\{x,y\}), \tag{2}$$

where $x \oplus y$ is the view cell resulting from merging x and y. The priority of the merge candidate is then given by $-\Delta_{c_r}(x,y)$.

In the beginning, the queue is initialized with all pairs of neighboring view cells. At every step, we select the merge candidate with the highest priority (smallest relative cost increase) and merge the associated view cells. The PVS and the estimated rendering cost of the new view cell is calculated. After the merge, the priority queue is updated by removing entries corresponding to the merged view cells and inserting new entries corresponding to the created view cell and its neighbors. Note that the set of neighbors for a view cell is determined using the BSP tree.

The merging process provides a sequence of view cell sets: at every merge step we obtain a new set of view cells with exactly one view cell less than in the previous set. We record the whole merging process in a merge history tree. The leaves of this tree correspond to elementary view cells. Every internal node corresponds to a merged view cell. With each internal node we associate the current cost of the subdivision resulting from the corresponding merge.

Once the merge history tree is built, there are several ways how to create a view cell partition from the tree. The easiest way is to specify a target number n of view cells and extract these from the tree (for more detail see Section 5). These view cells can then be used as input for a visibility preprocessing algorithm.

5. Exploiting the View Cell Hierarchy

This section discusses different possibilities for using the constructed view cell hierarchy.

5.1. Extracting View Cells

The view cell hierarchy allows extracting the set of view cells most suitable for the target application. Below we discuss three possibilities of view cell extraction.

Getting a specified number of view cells. Often, it is most convenient to specify a desired number of view cells to use for visibility calculation. This limits both the preprocessing time and the storage required for PVS data, as well as restricting the frequency with which new PVS data has to be fetched due to crossing into a new view cell at runtime. To obtain a given number of view cells, we perform a priority traversal of the merge history tree. The priority of a node is given by the cost associated with the node. When reaching a leaf node, we add it to the list of resulting view cells. When the sum of traversed leaves and the nodes in the priority queue becomes equal to the desired number of view

cells, we terminate the traversal and add the contents of the priority queue to the resulting view cells. The collected view cells form a cut of the merge history tree at optimal depths with respect to the specified granularity.

Fulfilling a given memory budget. The procedure described above can be extended to allow specifying an approximate memory budget for the complete PVS representation (it is only approximate because it relies on the approximate PVS from the sampling step). At every step of the tree traversal, we can easily calculate the memory requirements for the current set of view cells and their approximate PVSs. We can terminate the traversal when the budget is reached and collect the resulting view cells as described above. Note that this step can also be applied after computing the final visibility classification. In this case the memory budget would be the real budget for storing the PVS representation.

Extracting important view cells. An alternative to the methods described above is view cell extraction based on their estimated rendering cost. In particular we can use the maximal tolerance of the increase of the estimated rendering cost over the minimal cost, i.e., the cost provided by the densest view space partition (elementary view cells). The view cells are extracted again by a priority traversal of the merge history tree. At each step we evaluate the ratio of the current cost (stored with the processed node) and the minimal cost. If the ratio falls below a threshold, we terminate the traversal and collect the resulting view cells as described above.

Interactive specification. In practice, the user can combine these methods in an interactive setup. The selection process can easily be accomplished with a real-time visualization of the view cells and a depiction of the associated cost and memory budgets, as well as the just described cost ratio. Typically, the user would start by setting an initial cost ratio and refining the result interactively. This allows the user interactive control over the process, which is a feature often desired by practitioners.

5.2. PVS Compression

Although the view cell hierarchy is constructed in order to minimize the average rendering cost, it also provides a powerful tool for PVS compression. As a result of the rendering cost minimization, the siblings in the hierarchy will have mostly coherent PVSs. Thus, once we have calculated the actual PVSs using a visibility preprocessing algorithm, we can propagate the PVS information as high in the hierarchy as possible [GSF99]. The intersection of the PVSs of the children is propagated to the parent and deleted from the children, which reduces the number of references stored at the leaves.

6. Results

We have evaluated our method on three test scenes. The first scene (soda) represents a building interior, the second scene (atlanta) represents 30km^2 of Atlanta, and the third scene (vienna) represents the city of Vienna.

The soda scene consists of 9129 objects formed directly by the scene polygons, the atlanta scene scene of 3495 objects (100k polygons), and the vienna scene of 12668 objects (8M polygons).

6.1. Evaluation Framework

In our experiments we have observed that evaluating the view space partition by visual inspection is difficult. Often such an inspection can even be misleading: nicely looking view cells like those corresponding to corridors in a building, are bad in terms of the render cost. As a basic tool for evaluating the quality of the partition we use the dependence of the expected render cost on the number of view cells. For selected tests, we also use histograms which show the distribution of render cost among view cells for a specified granularity.

In order to compare different view cell construction strategies, we cast additional *evaluation* samples (rays) after the view cells have been constructed. The only purpose of the evaluation samples is to obtain comparable render cost estimations. These samples determine PVSs for all view cells of the view cell hierarchy, which are then used for evaluating equation 1. Note that although the number of evaluation samples we used is larger than the number of initial samples, the computed PVSs are still only approximations to the exact ones. For methods which don't use view space merging, the view cell hierarchy is just defined by the initial subdivision step.

6.2. Visibility Sampling

The first test aims to verify our assumption that a relatively coarse visibility sampling is sufficient to establish a stable render cost estimate for driving the view space partition. In order to evaluate this, we have constructed view space subdivisions using different numbers of visibility samples (50k, 200k, 1M). The results are summarized in Figure 6. As expected, lower numbers of visibility samples generate lower render cost estimates due to visibility undersampling. However, by casting the same number of evaluation samples, we can observe that the resulting subdivisions provide comparable render costs. For example there is only a minor improvement by moving from 200k to 1M samples.

6.3. Comparison with other methods

In Figure 7, we show a comparison with other proposed view cell construction methods. We compared 7 different

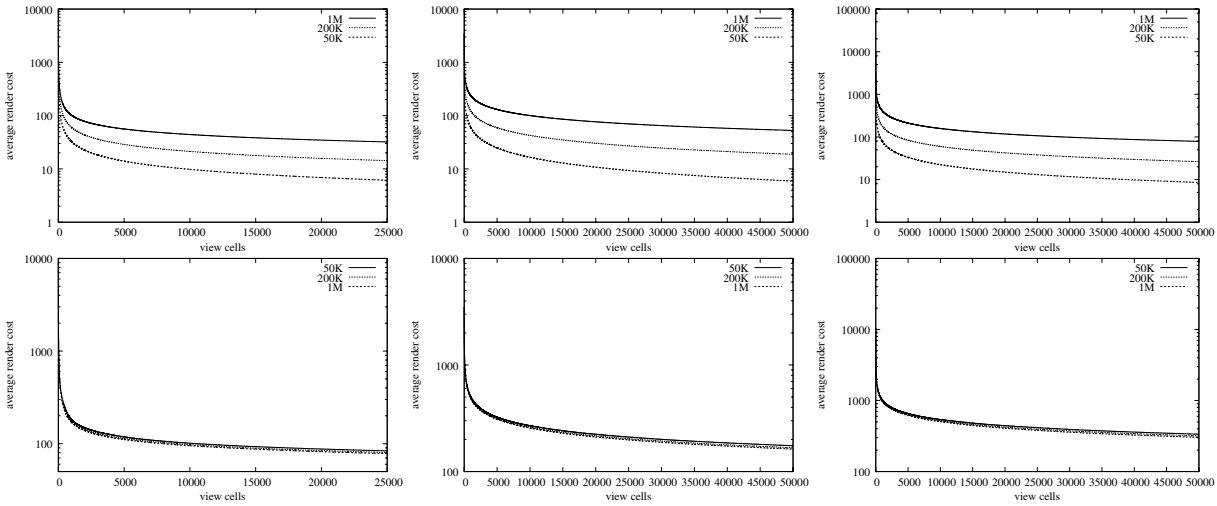

Figure 6: *Evaluation of visibility sampling. The top row of plots shows the actual render cost estimates for the soda, atlanta and vienna scenes for subdivisions created from 50k, 200k, and 1M visibility samples. The bottom row shows a verification of the resulting subdivisions by using 8M evaluation samples.*

view cell construction methods: our method with view space merging (AV-M), our method without view space merging (AV-S), breadth-first kD-tree subdivision using spatial median split and along cycling axis (KD-CASM), breadth-first kD-tree subdivision using spatial median split and along the longest axis (KD-LASM), depth-first kD-tree subdivision along the longest axis with additional visibility-based termination (KD-VT*), a breadth-first version of the previous method (KD-VT) and BSP subdivision aligned with the scene geometry (BSP). For our method we casted 1.5M rays to generate visibility samples. Note that BSP corresponds to the method for cell and portal construction described by Teller [Tel92] and KD-VT* corresponds to subdivisions performed by Saona-Vázquez et al. [SVNB99] and Nirenstein et al. [NB04] for visibility preprocessing in general scenes.

The results for three different test scenes are summarized in Figure 7. A number of observations can be made from the measured results:

- Our method performs better than the reference ones in all tests. The gain over the best reference method (KD-CASM) in terms of the render cost is about 20-30%.
- The gain of view space merging (AV-M) applied on our subdivision (AV-S) is very limited. The only benefit appears in the soda scene for a lower number of view cells.
- KD-CASM performs significantly better than KD-LASM in outdoor city scenes. In these scenes, horizontal subdivision of view cells is very important, as it separates regions with complex visibility which see above the roofs of the buildings from those which see only the nearby streets. When using KD-LASM, the horizontal splits occur only for a very high density of the subdivision.
- The plots show that BSP performs worst of the tested methods as it cannot provide a sufficient reduction of the render cost. This follows from the inability of BSP to subdivide regions with high render cost which however contain no geometry. This observation is not surprising for the atlanta scene and the vienna scene, as the method is not designed for outdoor scenes. However, even in the soda scene the BSP method leaves large view cells corresponding to the corridors with complex visibility. On the other hand as shown in Figure 8, our method successfully subdivides these high PVS regions.
- The depth-first subdivision (KD-VT*) arrives at a termination point with a cost of 50-80% higher than that of our method for the same number of view cells. However, as seen from the plot, it does not provide a scalable solution with respect to the number of view cells. This problem can be addressed by a breadth-first modification of the method (KD-VT). Alternatively, as we show in Section 6.4, scalable solution can be obtained by applying our merging step on the resulting subdivision.
- Similarly to KD-LASM, KD-VT also splits at the spatial median of the longest axis. As we used a relatively low PVS termination criterion, the results for these methods are practically the same.
- Another criterion for evaluating the methods is the number of view cells needed to achieve the same render cost. We can see that the reference methods need significantly larger numbers of view cells in order to reach the same render cost as the AV-S or AV-M methods. Also note that for some methods, a particular render cost cannot be reached within a given view cell budget.

We have also compared the KD-VT and AV-M methods using a histogram showing the distribution of render cost

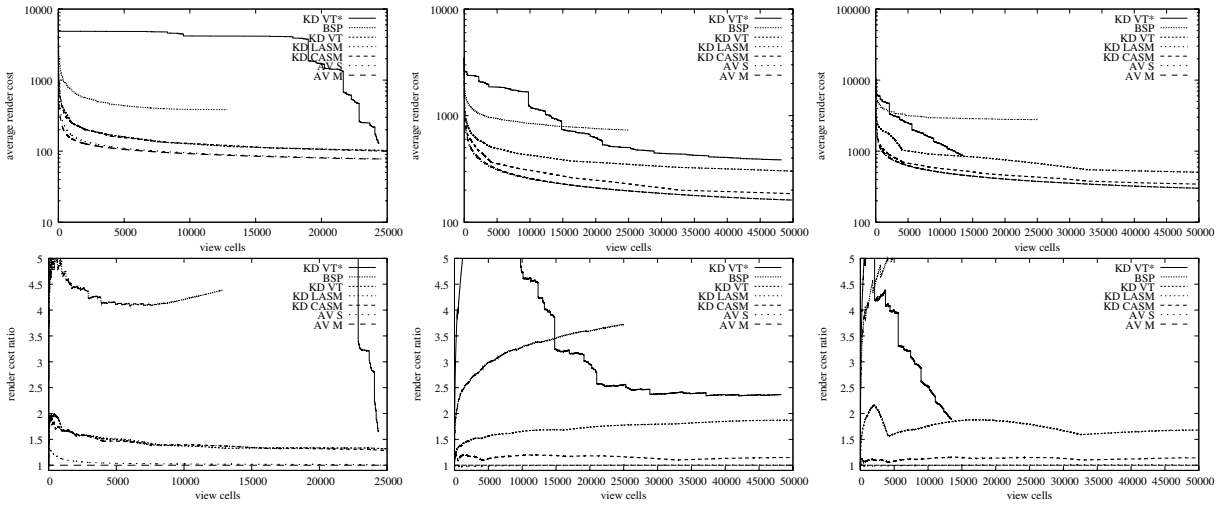

Figure 7: *Comparison of the different view cell construction methods for three different scenes (from left to right: soda, atlanta, vienna). The top row shows the expected render cost depending on the number of view cells. The bottom row shows the render cost ratio with respect to our AV-M method.*

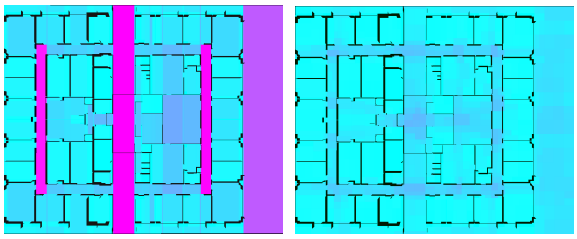

Figure 8: *BSP based view cells have the problem that large regions with high PVS (shown in dark/magenta) are not subdivided any further (subdivided down to 2734 leaves for both methods).*

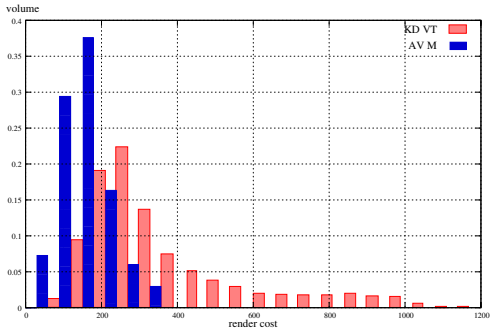

Figure 9: *Render cost / volume histogram for the AV-M and KD-VT methods computed for the vienna scene with 15000 view cells.*

over the view space volume (see Figure 9). For the AV-M method, most of the volume is covered by lower render cost. In contrast, the KD-VT method still contains numerous regions with high render cost: there are regions with up to 3 times higher render cost compared to the maximum render cost for the AV-M method. This observation indicates that even if AV-M does not provide an impressive reduction of the average render cost, it results in a significantly better render cost distribution over the view space volume.

6.4. Using view space merging on existing solutions

View space merging can be applied as a standalone technique for obtaining a scalable view cell representation from an existing set of view cells. An example of this process is shown in Figure 10. We can see how the initial depth-first subdivision has been smoothed by the application of view space merging. Additionally the render cost curve resulting from the merging process gives us information about the required granularity of the subdivision. If many initial merging steps result only in a minor increase of the render cost, then we can safely reduce the number of view cells for the final visibility representation.

6.5. Influence of geometry-aligned splits

We have evaluated the influence of the geometry-aligned splits on the expected render cost in non-axis aligned environments. In particular, we measured the render cost curves for our method when using only axis-aligned splitting planes and when using also geometry-aligned planes. We limited the number of tested geometry-aligned split planes to 150. The result is summarized in Figure 11. We can see

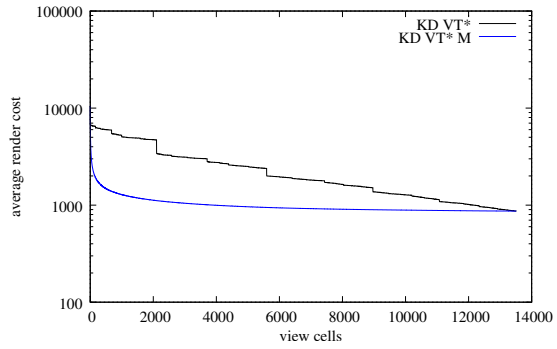

Figure 10: *View space merging applied on the subdivision using the KD-VT* method.*

that for this test the geometry-aligned planes give a benefit from 4 to 15%. This suggests that when taking into account the increased computational complexity connected with geometry-aligned splits, the benefit they provide is only marginal.

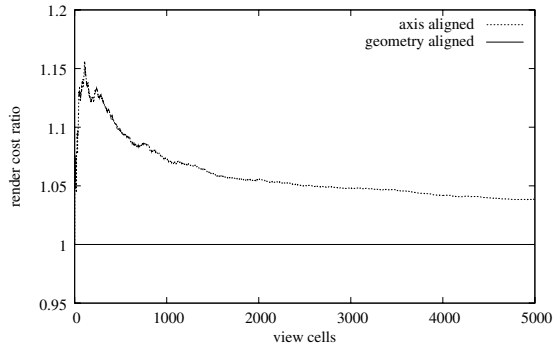

Figure 11: *Influence of geometry-aligned splits in a soda building rotated by 30 degrees. The plots shows the render cost ratio of using only axis-aligned splits with respect to a method using also geometry aligned splits (render cost ratio=1).*

6.6. Timings

We have measured the running times of our view cell construction for several scenarios. The timings measured on an 3.4GHz Intel Pentium 4 with 2GB of RAM are summarized in Table 1. We see that for more complex scenes, the time overhead due to geometry-aligned candidates as well as the time required for the merging increases. Since the benefit of these two techniques is relatively small in practice it should be sufficient to use our priority-driven subdivision using the cost heuristics and axis-aligned splits.

scene	subdivision AA + GA [s]	subdivision AA [s]	merge [s]
soda	430.8	287	103.3
atlanta	1488	338.1	1912
vienna	1565	354.1	1569

Table 1: *Timings for generating 50000 view cells using our view cell construction method. For all tests we cast 1.5M rays to generate visibility samples. The table contains timings for the subdivision when using axis-aligned as well as geometry-aligned planes (AA + GA) and when using only the axis-aligned planes (AA). We limited the number of geometry-aligned candidate split planes to 150. The last column shows timings for the view space merging step.*

7. Discussion

7.1. Visibility sampling

The idea which drives our view cell construction algorithm and which allows analyzing the quality of different view space subdivisions is visibility sampling and the usage of the visibility segments. As we have shown in the results, it is not necessary to calculate an accurate visibility solution for each step of the subdivision process in order to determine the following steps. Instead, even a relatively coarse sampling of visibility already gives stable estimates of the render cost. Our algorithm is therefore not limited by a slow visibility solver. It can quickly perform a deep top-down subdivision, which would not be possible if each subdivision step were to depend on the outcome of a complete visibility processing step.

7.2. Handling empty space

One of the important differences of our method compared to techniques designed for cell and portal graph construction is the handling of 'empty space', i.e., regions which contain no geometry. The results show that even in indoor environments it is very important to further refine the subdivision in order to adapt to visibility (rendering) complexity. A purely geometry-based subdivision might result in big view cells which look 'natural', but their PVS is way too large, even though a significant reduction is possible (as shown in Figure 8).

7.3. Handling difficult regions

The existing visibility preprocessing methods which perform adaptive view space partitioning [SVNB99, NB04] handle difficult regions (i.e., regions with large irreducible PVS) by terminating subdivision at a specified maximal depth. This can be costly for preprocessing as well as storage. In our method this problem is addressed already at the subdivision stage and gets further refined in the merging

step. The subdivision uses the cost-based global termination criterion which prevents further subdivision when all regions already have irreducible render cost. However there might be regions where the local benefit of the subdivision is just above the specified threshold. These regions will be merged soon in the merging stage, and thus the corresponding fragmented view cells will reside at the bottom of the merge history tree.

8. Conclusions

We have described a new method for automatic view space partitioning. The algorithm uses a cost model in order to minimize the average rendering time of the resulting set of view cells. The model is based on a global visibility estimation determined by sampling. We have shown that the method provides efficient sets of view cells for both indoor and outdoor environments. Since we use the actual visibility of the scene for driving the view cell construction, our view cells adapt to scene visibility changes. This is important for example for regions with no geometry, where other cell construction methods can fail.

As a result of the view cell construction we obtain more than a fixed set of view cells. Depending on the properties of the visibility preprocessing algorithm or a runtime storage budget, we can extract a specified number of view cells which provide an optimized partition for a given granularity.

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).

References

[Ail00] AILA T.: *SurRender Umbra: A Visibility Determination Framework for Dynamic Environments*. Master's thesis, Helsinki University of Technology, 2000.

[ARB90] AIREY J. M., ROHLF J. H., BROOKS, JR. F. P.: Towards image realism with interactive update rates in complex virtual building environments. In *1990 Symposium on Interactive 3D Graphics* (Mar. 1990), ACM SIGGRAPH, pp. 41–50.

[CLF98] CAMAHORT E., LERIOS A., FUSSELL D. S.: Uniformly sampled light fields. In *Rendering Techniques* (1998), pp. 117–130.

[COCSD02] COHEN-OR D., CHRYSANTHOU Y., SILVA C., DURAND F.: A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics.* (2002).

[GSF99] GOTSMAN C., SUDARSKY O., FAYMAN J. A.: Optimized occlusion culling using five-dimensional subdivision. *Computers and Graphics 23*, 5 (Oct. 1999), 645–654.

[HDS03] HAUMONT D., DEBEIR O., SILLION F.: Volumetric cell-and-portal generation. *Computer Graphics Forum 22*, 3 (September 2003), 303–312.

[LCOC03] LERNER A., COHEN-OR D., CHRYSANTHOU Y.: Breaking the walls: Scene partitioning and portal creation. In *Pacific Graphics* (2003).

[LG95] LUEBKE D., GEORGES C.: Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *1995 Symposium on Interactive 3D Graphics* (Apr. 1995), Hanrahan P., Winget J., (Eds.), ACM SIGGRAPH, pp. 105–106.

[MBMD98] MENEVEAUX D., BOUATOUCH K., MAISEL E., DELMONT R.: A new partitioning method for architectural environments. *Journal of Visualization and Computer Animation 9*, 4 (1998), 195–213.

[NB04] NIRENSTEIN S., BLAKE E.: Hardware accelerated aggressive visibility preprocessing using adaptive sampling. In *Rendering Technqiues 2004* (2004), pp. 207–216.

[PD90] PLANTINGA H., DYER C.: Visibility, occlusion, and the aspect graph. *International Journal of Computer Vision 5*, 2 (1990), 137–160.

[SVNB99] SAONA-VÁZQUEZ C., NAVAZO I., BRUNET P.: The visibility octree: a data structure for 3D navigation. *Computers and Graphics 23*, 5 (Oct. 1999), 635–643.

[Tel92] TELLER S. J.: *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.

[TS91] TELLER S. J., SÉQUIN C. H.: Visibility preprocessing for interactive walkthroughs. In *Proceedings of SIGGRAPH '91* (July 1991), pp. 61–69.

[vdPS99] VAN DE PANNE M., STEWART A. J.: Effective compression techniques for precomputed visibility. In *Rendering Techniques* (1999), pp. 305–316.

[WW03] WIMMER M., WONKA P.: Rendering time estimation for real-time rendering. In *Rendering Techniques* (2003), pp. 118–129.