Adaptive Global Visibility Sampling

Jiří Bittner* Oliver Mattausch† Peter Wonka‡ Vlastimil Havran* Michael Wimmer†

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

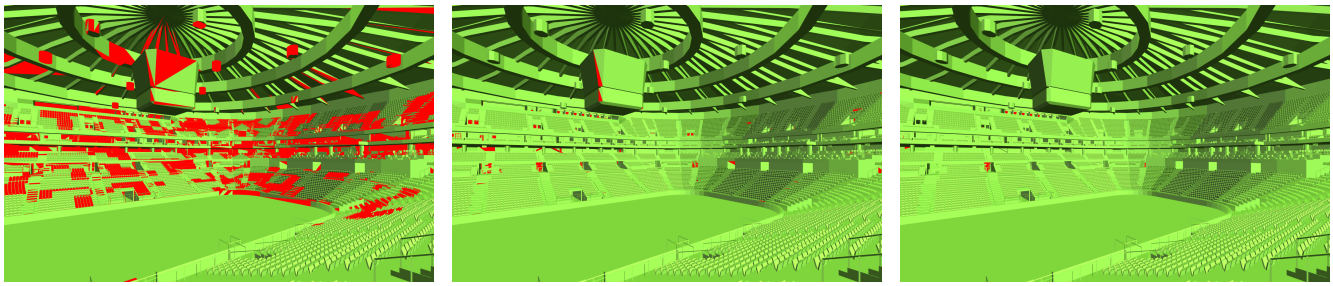

Figure 1: *Results of visibility computations after 1 minute of sampling. Visibility errors are marked in red. Left: Traditional per-view cell sampling. Middle: Adaptive Global Visibility Sampling. Right: Adaptive Global Visibility Sampling with visibility filter. Observe the severe underestimation of visibility in the left image. The visibility computed by our method in the middle produces significantly less visible artifacts. To the right, our method with a visibility filter applied is practically artifact-free. Note that during this minute, the potentially visible sets for* all *8,192 view cells in this example model have been generated.*

Abstract

In this paper we propose a global visibility algorithm which computes from-region visibility for all view cells simultaneously in a progressive manner. We cast rays to sample visibility interactions and use the information carried by a ray for all view cells it intersects. The main contribution of the paper is a set of adaptive sampling strategies based on ray mutations that exploit the spatial coherence of visibility. Our method achieves more than an order of magnitude speedup compared to per-view cell sampling. This provides a practical solution to visibility preprocessing and also enables a new type of interactive visibility analysis application, where it is possible to quickly inspect and modify a coarse global visibility solution that is constantly refined.

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

Keywords: Visibility preprocessing, from-region visibility

1 Introduction

This paper addresses the problem of computing Potentially Visible Sets (PVS) for a set of view cells. PVS computation is typically carried out as a preprocess in applications that need to know visibility information in advance and where online visibility cannot be used. These applications include polygon budget computations, level design for computer games, line-of-sight analysis, planning geometry and texture transmissions for networked virtual environments, acquisition planning for object scanning, or path planning for artificial intelligence in computer games.

Recently it has been shown that sampling is a robust solution to PVS computation. In particular the Guided Visibility Sampling (GVS) algorithm introduced by Wonka et al. [Wonka et al. 2006] efficiently computes a PVS for a view cell using ray casting. While GVS is very efficient at sampling from a single view cell, it does not exploit the coherence among different view cells and does not work in progressive fashion with respect to all view cells.

In this paper we propose a method which computes from-region visibility for all view cells simultaneously in a progressive manner. We build on global sampling strategies [Mattausch et al. 2006] and use visibility information from one ray for all view cells it intersects. The main contributions of the paper are: (1) We introduce a set of sampling strategies for global visibility computation which adapt to the scene geometry and explore the visibility coherence. (2) We show the first progressively refined global visibility solution, in which PVS estimates for all view cells are obtained within seconds or minutes (see Figure 1).

The method achieves more than an order of magnitude speedup compared to per-view cell sampling. This provides a practical solution to visibility preprocessing and also enables a new type of interactive visibility analysis applications (see Figure 2), where it is possible to quickly inspect and modify a global visibility solution. The proposed algorithm consumes little memory, is easy to implement, and works on arbitrary input scenes consisting of view cells and objects intersectable with a ray tracer.

2 Related work

Visibility and occlusion is essential to a wide area of graphics problems and we refer the reader to recent surveys for a broader discussion of visibility [Cohen-Or et al. 2003; Bittner 2003]. In the following we will focus on visibility algorithms that compute visi-

*e-mail:{bittner|havran}@fel.cvut.cz
†e-mail:{matt|wimmer}@cg.tuwien.ac.at
‡e-mail:wonka@asu.edu
§Faculty of Electrical Engineering

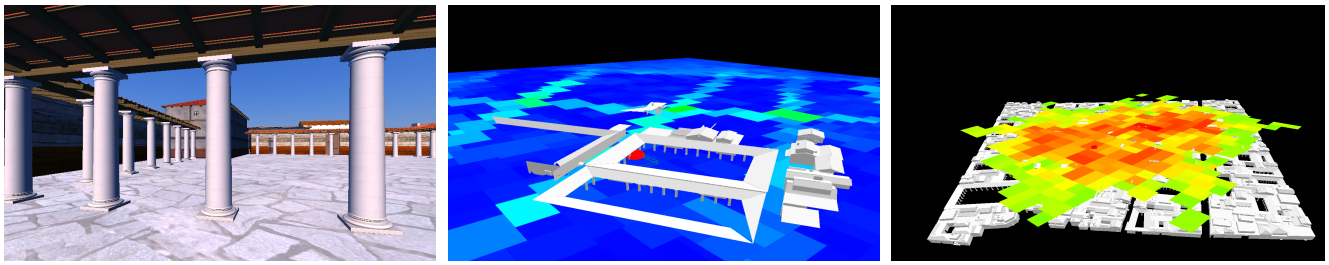

Figure 2: *Interactive visibility analysis. Left: A view of the Pompeii model. Middle: The PVS of the corresponding view cell depicted together with PVS costs of the view cells at the same horizontal level (warmer colors correspond to higher costs). Right: Interactive exploration of visibility hot spots. Note that in this scene the hot spots correspond to view points above the roofs in the central part of the scene.*

bility from one or multiple regions in space as opposed to calculating visibility from a single point.

Geometric solutions. First, it is important to study exact solutions to the visibility problem [Duguet and Drettakis 2002; Nirenstein et al. 2002; Bittner 2003; Haumont et al. 2005]. These algorithms are an important inspiration, but it is currently unclear if exact algorithms can be extended to handle large scenes robustly. Therefore, many authors set out with simplifying assumptions to make the problem more tractable. Interesting simplifications are 2.5D visibility [Wonka et al. 2000; Bittner et al. 2001; Koltun et al. 2001], architectural scenes [Airey et al. 1990; Teller and Séquin 1991], the restriction to volumetric occluders [Schaufler et al. 2000], or the restriction to larger occluders close to the view cell [Durand et al. 2000; Leyvand et al. 2003].

Visibility Sampling. A popular concept in visibility computation is to sample visibility interactions on a regular grid using graphics hardware, giving an approximate solution. Conservative algorithms attempt to interpret a pixel as a small subset of ray space [Durand et al. 2000; Wonka et al. 2000; Koltun et al. 2001; Leyvand et al. 2003] and record occlusion information only if all rays corresponding to the ray space subset are blocked. These algorithms are not able to handle complex visibility configurations within the subset of ray space defined by a pixel and basically require a single occluder to block all rays [Cohen-Or et al. 1998]. As computing power increased, it became feasible to avoid conservative algorithms and just sample ray space very densely. A conceptually elegant idea is to shoot either random rays from a view cell [Airey et al. 1990], or to first sample the boundary of the view cell with points and then sample visibility from each of these points [Levoy and Hanrahan 1996]. Recent algorithms try to address the question on how to best position new samples based on visibility information from previous samples [Gotsman et al. 1999; Pito 1999; Wilson and Manocha 2003; Nirenstein and Blake 2004; Wonka et al. 2006].

Sampling in other fields. Sampling has been widely used in global illumination to calculate the illumination integral [Dutré et al. 2003]. In contrast to global illumination, it is not an integral that is evaluated in visibility computations, but the *maximum set* of different values of the visibility function. Related to our technique are the VEGAS algorithm [Lepage 1980], which adapts the sampling distribution based on an estimation of the integrand from previous samples, and Metropolis sampling [Veach and Guibas 1997], where a new sample is generated by displacing the previous one randomly. A similar strategy is known as *mutation* in genetic algorithms [Goldberg 1989]. Analogous to genetic algorithms, we start with a random sampling step and then generate new samples by mutation. In sampling theory, this strategy is also called *adaptive cluster sampling* [Thompson and Seber 1996], and it is used to sample rare populations.

3 Overview

3.1 Problem Statement

We consider visibility problems of the following form: given a view space as a set V of view cells $v \in V$, and a set O of objects $o \in O$, we are interested in which objects can be seen from which view cell. More formally, let a ray $r = (x_r, d_r)$ be defined by a ray origin x_r and a ray direction d_r. The ray casting function $h(r) \in O$ assigns each ray the first object hit by the ray. Then the desired visibility solution is the *exact visible set* EVS_v for each view cell. It is defined as the actual set of objects that can be seen along some ray from v: $EVS_v = \{h(r) | x_r \in v\}$.

The potentially visible set PVS_v of a view cell is an approximation to EVS_v determined by a particular visibility algorithm. Conservative algorithms overestimate the visible set ($PVS_v \supseteq EVS_v$), while aggressive algorithms underestimate it ($PVS_v \subseteq EVS_v$). Sampling algorithms such as the one described in this paper provide an aggressive solution. We also describe a visibility filter to extend PVSs, which makes the solution approximate ($PVS_v \sim EVS_v$).

A *global visibility solution* is simply the set $\{PVS_v | v \in V\}$. A global visibility algorithm is *progressive* if at each step i it computes a solution $\{PVS_v^i | v \in V\}$ with $PVS_v^{i-1} \subseteq PVS_v^i \subseteq EVS_v$. Loosely speaking, we only call a solution progressive if it is so globally, i.e., the ratio $|PVS_v^i|/|EVS_v|$ increases for all view cells v simultaneously during the runtime of the algorithm (in particular, sequential view cell evaluation is not a progressive solution).

3.2 Algorithm Overview

The Adaptive Global Visibility Sampling (AGVS) algorithm uses *sampling* to determine visibility. We assume the availability of a ray tracing algorithm [Shirley et al. 2006] that can evaluate $h(r)$, and in addition $r \cap v$ for any view cell v.

At the core of our algorithm is a *global visibility sampler* (Section 4.1), which casts bidirectional rays to determine maximal free line segments in the scene, and then determines their contribution to all view cells. Thus a single visibility sample can contribute to many view cells.

The second part of the algorithm generates the samples. The 5D sampling domain is too large to quickly capture all important rays by regular sampling. Therefore we use different heuristical distributions which are combined in an *adaptive mixture distribution* (Section 4.2) that takes into account their success in discovering new PVS entries. The sampling uses several ray distributions that are

suitable for a global visibility algorithm. *Stationary distributions* (Section 4.3) sweep visibility globally and seed the *mutation-based distributions* (Section 4.4), which focus the sampling at places of visibility changes and allow adapting the sampling rate to the distance of visible objects.

We present *visibility filtering* (Section 5), which counteracts errors due to undersampling in early stages of the algorithm by including in the PVS also objects that are likely to be visible based on the sampling density. We also describe a quick algorithm to discover areas affected by scene edits, thus allowing *dynamic edits* to the scene without having to recompute the whole visibility solution (Section 6).

4 Adaptive Global Visibility Sampling

In this section we describe novel sampling strategies that are well suited to the global computation of visibility.

4.1 Global Visibility Sampling

Visibility sampling relies on *ray tracing* to obtain visibility information. One main reason for the efficiency of our approach is the use of spatial coherence in visibility. In contrast to a per-view cell algorithm, where each ray can only contribute to one single view cell, we determine all view cells the ray encounters, as was proposed in the context of view-cell optimization [Mattausch et al. 2006], and calculate the contribution to those view cells. For this, we use *visibility samples* created from rays.

Visibility sample definition. In visibility sampling algorithms, a sample ray r is usually defined to start in a view cell ($x_r \in v$) and with a direction d_r. A visibility contribution is then determined by shooting the ray using a standard ray tracer and determining the closest object $h(r)$. We define the contribution of a ray as

$$\begin{aligned} H(r) &= \{h(r)\} \quad \text{if } h(r) \in O \text{ (object hit)} \\ &= \{\} \quad \text{otherwise (no object hit)} \end{aligned} \tag{1}$$

In global visibility, however, we are not only interested in the visibility contribution of the hit object to the view cell at the ray origin, but to all view cells pierced by the ray along an unobstructed path from the hitpoint. This path is equivalent to the *maximal free line segment* defined by r. To obtain this line segment, we not only shoot the original ray, but also the ray in the opposite direction, i.e., $-r = (x_r, -d_r)$, and obtain a second hit $h(-r)$. Our sample s is thus a line segment associated with zero, one or two visible objects at its endpoints, with contribution $H(s) = H(r) \cup H(-r)$. A sample with one or two visible objects ($|H(s)| > 0$) is a *valid sample*.

Updating view cells. The view cells affected by a valid sample are determined by intersecting the line segment with the data structure containing the view cells. The objects associated with the sample are then added to *all* view cells pierced by the line segment.

Sample contribution. During this process, the *contribution* of the sample is evaluated in order to distinguish between samples which give us valuable information about visibility and samples which either hit no object or hit objects already discovered as visible. This measure will be used to drive the sampling process (Section 4.2).

The local contribution of the sample to a particular view cell equals the number of objects associated with the sample that are added to the PVS of the view cell (and have not already been included in the PVS in previous steps of the algorithm): $\bar{c}(s,v) = |H(s) \backslash PVS(v)|$.

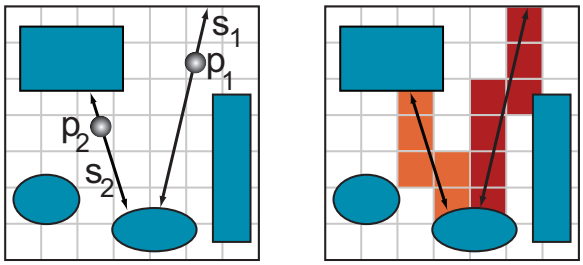

Figure 3: *Left: This figure shows the creation of visibility samples. The scene objects are shown in blue. A sample point (see p_1 and p_2) is generated together with a direction vector. A ray is cast from the sample point in the generated direction as well as into the opposite direction. Right: A ray associates all its points with visibility of object(s) on its endpoints. These objects are added to all PVSs of view cells pierced by the ray.*

Note that $\bar{c}(s,v)$ yields a value of 0, 1 or 2. The total contribution $c(s)$ of the sample is simply the sum of contributions to all view cells $V(s)$ which are pierced by the sample:

$$c(s) = \sum_{v \in V(s)} \bar{c}(s,v) \tag{2}$$

4.2 Adaptive Mixture Distribution

In addition to the explicit use of spatial visibility coherence, the most important novel contribution of our global visibility algorithm is the ability to adapt to the visibility structure in the scene. This is achieved in two ways: first, by choosing sample distributions according to previous visibility contributions, and second by using a ray mutation strategy which places samples near visibility events and thus adapts to the required sampling density in ray space (described in Section 4.4).

We have found that no single sampling strategy is efficient for all types of scenes, as the efficiency of a sampling strategy depends on the scene properties and its visibility characteristics. We therefore employ a probabilistic approach based on a distribution mixture: we allow the use of several different distributions to generate visibility samples. The success of previous samples generated by each distribution is used to drive its selection probability, thus automatically adapting to the scene visibility properties. This strategy is reminiscent of VEGAS importance sampling [Lepage 1980], although we do not try to compute an integral here.

More specifically, the contribution $C(D)$ of a sample distribution D at a specific point in time is defined as the sum of contributions $c(s)$ of all samples s in a reference set $S(D)$. The reference set is typically the set of rays generated by the distribution D in a certain time window:

$$C(D) = \sum_{s \in S(D)} c(s). \tag{3}$$

Next we compute the average contribution per sample from D as $c_s(D) = C(D)/|S(D)|$. To account for the fact that samples from different distributions have different processing costs (e.g. a distribution need not be very succesful per ray, but the rays are generated and cast very quickly) we compute the *weight* $w(D)$ of the

distribution D as its average contribution per time unit: $w(D) = c_s(D)/t_s(D)$, where $t_s(D)$ is the average time for processing a sample from the distribution D. $t_s(D)$ is measured for each distribution in a *calibration pass* by generating and processing a sufficiently large number of samples (e.g. 100k) from each distribution. Note that $w(D)$ can slightly change over time, so we repeat the calibration passes during the computation (e.g. after each 100M samples).

The probability for drawing a sample from distribution D_i in the mixture distribution is then set to:

$$p(D_i) = \frac{w(D_i)}{\sum_D w(D)}. \quad (4)$$

The following two subsections describe the distributions which we use in the algorithm. Three of these distributions are stationary, while two mutation-based strategies adapt to the actual visibility contributions.

4.3 Stationary Ray Distributions

A distribution is stationary if each sample is independent of previously drawn samples. These distributions are used to provide an initial efficient covering of ray space and to seed the more adaptive mutation-based strategies described below. The stationary distributions explore the whole ray space and therefore guarantee the progressivity of the algorithm.

In contrast to per-view cell visibility, there is no obvious "uniform" sample distribution. Instead, the best strategy to discover new visible objects depends on the visibility configuration of the scene. We list three distributions which we have found to work well. All rely on low-discrepancy series like the Halton sequence to generate samples. We use independent random variables $\psi_1, \psi_2, \ldots$ in the range $[0,1)$.

View space-direction distribution. This distribution makes sure that all view cells are sampled in all directions. We generate a random point x in view space using a uniform distribution and a random direction d using a uniform distribution in the directional space with spherical coordinates $\phi = 2\pi\psi_1, \quad \theta = \arccos(1-2\psi_2)$.

Object-direction distribution. This distribution makes sure that all objects are sampled in all directions (note that this can be inefficient if the view space does not fully include the object space). We generate a random point x on the surface of a randomly chosen object, and a random direction d using a cosine-weighted uniform distribution on the hemisphere erected over the tangent plane of x. The spherical coordinates of d are: $\phi = 2\pi\psi_1, \quad \theta = \arccos\sqrt{\psi_2}$.

Two-point distribution. This distribution focuses on the most probable visibility interactions between objects and view cells. It can be seen as a combination of the previous two strategies which considers only the points generated by them and not the directions. So from a point o on a random object and a point v in view space, the ray generating the visibility sample is $r = (v, o-v)$. The most important feature of the two-point distribution is that it adapts to the shape of the scene. For example, typical urban scenes are much wider than they are high. This fact is taken into account in the two-point distribution so that most samples are cast in a roughly horizontal direction. For this reason, the two-point distribution is typically the most successful stationary sampling strategy.

4.4 Mutation-Based Distributions

Even using an optimal mix of stationary sample distributions, the size of ray space that needs to be sampled is prohibitively high. The required sampling density is very non-uniform: for a particular view cell, far away regions need to be sampled more densely than near regions. Furthermore, Wonka et al. [2006] have shown that the efficiency of visibility sampling can be vastly improved by trying to sample near possible changes in visibility. We exploit these two observations using mutation-based sampling distributions: a *two-point mutation* distribution and a *silhouette mutation* distribution.

Mutation candidate maintenance. Both strategies are based on the following principle: During the sampling process, each sample with non-zero contribution $c(s)$ (regardless of which distribution generated it) is stored as a candidate for mutation. In case the visibility sample has two objects associated with it ($|H(s)| = 2$), and both objects actually generated a contribution in at least one cell, we generate two mutation candidates from it, as we distinguish between the segment termination point (where the object under consideration is hit) and the segment origin (either the intersection with an object or a view cell). These two points and the object id of the object are stored with the sample.

All mutation candidates for a strategy are collected in a buffer. When a new sample is requested from a mutation-based strategy, a candidate is chosen from the buffer. However, instead of choosing randomly from the buffer, we observe that the history of mutations conveys important information about the possible importance of a candidate: If a candidate has received a large number of mutations already, it is likely that its neighborhood is already well explored. On the other hand, new mutation candidates have not received any mutations so far, and especially after sampling has been running for some time, new mutation candidates represent more "difficult" cases of visibility. We therefore count the number of mutations generated from each candidate. This mutation count is used to sort the buffer and we always select the candidate with the lowest mutation count value. If the buffer is full, the candidate with the highest value is dropped.

Two-point mutation. The aim of the two-point mutation is to adapt the sampling rate of visibility to the complexity of ray space. More specifically, given a segment $s = (s_o, s_t)$, the segment termination point s_t is mutated so as to discover nearby *objects*, while the segment origin s_o is mutated so as to discover nearby *view cells* (note that it is entirely due to the global nature of the algorithm that a strategy for discovering new view cells is possible at all).

Let $o(x)$ be the object or view cell (in case the reverse ray generating the segment didn't hit an object) associated with point $x \in \{s_o, s_t\}$, and $r(o)$ the radius of the bounding volume of object or view cell o. Then we construct a plane perpendicular to the segment, and mutate x by drawing a new point x' from a two-dimensional Gaussian distribution on this plane centered at x with standard deviation $\sigma = r(o(x))$ (see Figure 4, left).

Note that for both mutation-based strategies, once we have obtained the new segment (s'_o, s'_t), a visibility sample is generated from this segment by creating a forward ray $r' = (x_{r'}, d_{r'})$ together with the reverse ray $-r'$ with $x_{r'} = (s'_o + s'_t)/2$ and $d_{r'} = s'_t - s'_o$. The new ray origin is chosen at the center of the new segment so as to avoid local occlusion at the start or termination points of the segment.

Silhouette mutation. The silhouette mutation adapts the deterministic adaptive border sampling strategy proposed by Wonka et al. [2006] for a progressive setting. This strategy places samples

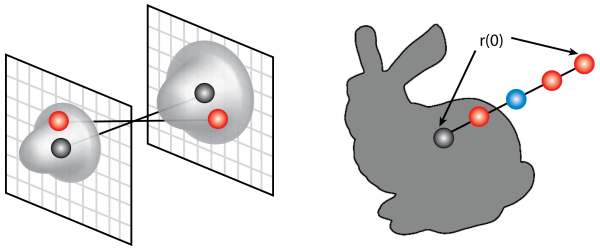

Figure 4: *Left: The two-point mutation moves both start and endpoints according to Gaussian distributions. Right: Silhouette search.*

near the silhouettes of newly discovered objects, where changes in visibility are most likely to occur. However, deterministically sampling the whole silhouette of an object is not only computationally expensive, but would also quickly saturate the list of mutation candidates.

Therefore we chose a probabilistic approach that randomly selects *one* silhouette point. As in the two-point mutation, a plane perpendicular to the segment is placed at s_t. On this plane, we choose a random direction d. Then we shoot "discovery rays" with endpoints s'_t along the segment $(s_t, r(o)\cdot d)$. The closest discovery ray that does not intersect the object is chosen as a silhouette ray and the mutation point s'_t is used to construct a new visibility sample (see Figure 4, right).

Note that for most ray tracers, these discovery rays can be evaluated much faster than ordinary rays. First, the region of interest can be restricted to the bounding volume of the object so that the ray does not need to be intersected with the whole scene. Second, the segment origin remains fixed for all rays, so that ray packet optimizations can be exploited. For example, shooting packets of 4 rays, a quaternary search of depth 3 provides a ray very close to the object silhouette.

Important visibility events also appear at possible *depth discontinuities* discovered by the silhouette sample. For this, we adapt the reverse sampling strategy described by Wonka et al. [2006]. When the silhouette ray $r = (s_o, s'_t - s_o)$ is cast, a depth discontinuity is reported if the distance between mutated segment endpoint s'_t and the actual hitpoint $h(r)$ is larger than k times the original object radius: $s'_t - h(r) > k \cdot r(o)$, where k is a user supplied constant (we used $k = 3$). This margin is intended to avoid situations where a closer hitpoint on the same object could be interpreted as a depth discontinuity. Now instead of looking for the exact depth discontinuity as in reverse sampling, we do a new silhouette mutation as described above using the segment $(s'_t, h(r))$, and store the resulting ray along with the original silhouette ray.

4.5 Termination of the computation

An important problem of previous visibility preprocessing techniques has been choosing the parameters of the method in order to achieve a good solution. For conservative methods, the parameters need to be set so that the PVS is not too overestimated, whereas for aggressive methods the final pixel error needs to be controlled.

The fact that our method is based on sampling allows us to devise a very elegant way to estimate an average pixel error on the fly. The estimated pixel error serves as a measure of the quality of the current solution and also as an intuitive termination criterion.

We define the average pixel error as the average number of rays from a randomly positioned and oriented camera with a given resolution (number of rays) which hit a different object when using the PVS related to the camera position compared to the situation when using all scene objects.

The distribution of rays covered by randomly positioned cameras corresponds exactly to the view space-direction distribution of rays described above. Thus we evaluate the ratio of contributing rays generated by the view space-direction distribution. As a contributing ray we count a ray which generated a new PVS entry with respect to the view cell contaning the origin of the ray. The average pixel error ε is given as $\varepsilon = resolution \cdot N_c/N$, where N is the number of rays generated by view space-direction distribution and N_c is the number of contributing rays.

In order to obtain a sufficiently large N we use a time window consisting of rays recently generated using the view space-direction distribution. We adapt the size of the window following the ratio of N_c to N: in the beginning of the computation a small window is sufficient, whereas in the nearly converged state, when the pixel error is small (1 pixel / 1M pixel image), we need to collect many rays in order to estimate the pixel error with reasonable precision. By following the weak law of large numbers for binomial distributions we get:

$$N \geq \frac{1-\varepsilon}{k^2\varepsilon(1-P)}, \tag{5}$$

where P and k are constants such that P represents the desired probability with which the absolute difference between the estimated error and the real error is smaller than $k\varepsilon$. We observed that a good tradeoff between the size of the window and the accuracy of the estimation is achieved using $P = 0.9$ and $k = 0.5$.

4.6 Putting It All Together

The complete Adaptive Global Visibility Sampling algorithm works in a loop as follows:

```
while (!terminate)
{
  select distribution
  draw a sample from selected distribution
  cast forward and reverse rays
  if (hit)
  {
    update view cells
    if (contribution > 0)
      store sample as mutation candidate
  }
  update distribution probabilities
}
```

Note that since the solution is progressive, it can be inspected at any time, and if it is not satisfactory, the algorithm can easily be resumed.

Loop optimization. In practice the loop is not carried out for individual rays, but for larger batches of rays (e.g., 1M). This allows reordering the samples so as to achieve better coherence for the ray tracer. In addition, if the window used for calculating ray distributions is chosen to be exactly one batch, the update of the distribution probabilities $C(D)$ (see Section 4.2) needs to be carried out only once per batch.

5 Visibility Filtering

While Adaptive Global Visibility Sampling is an aggressive algorithm (i.e., each PVS only contains a subset of the exact visible set), the sampling strategies described in the previous section aim to approach the exact solution as quickly as possible. However, especially in the initial phases of the algorithm, visible errors will appear.

To cope with this problem, we introduce the so-called *visibility filter*, which extends the computed PVSs by additional objects which are likely to be visible. The main novelty is that we take into account the visibility error expected from the sampling density. This means that PVSs for regions which have been sampled densely will not be extended significantly, whereas undersampled regions will have more objects added. The visibility filter fills gaps in "visible fronts" that appear when the sampling rate in a region is lower than the object density. It cannot discover objects that are visible in an isolated manner (i.e., objects smaller than the sampling density that have no already discovered visible neighbors).

The visibility filter can be applied as a postprocess to all PVSs after running the main algorithm, or it can be evaluated lazily during walkthrough for the current view cell. We present an *object space filter*, which adds objects in proximity of already visible objects, and a *view space filter*, which merges visible objects from neighboring view cells.

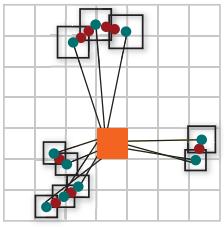

Figure 5: *Illustration of the object space visibility filter. Objects originally in the PVS are shown in blue. The extended bounding volumes are shown in black. Objects added to the PVS by the filter are shown in red.*

Object space filter. The object space filter works on a view cell v and its potentially visible set PVS_v. For each object $o \in PVS_v$, the size of *extension* $e(o)$ is calculated as the estimated distance to a visibility sample in the vicinity of the object. The object space filter is then applied by extending the bounding volume of each object o by $e(o)$ (e.g., the radius of a bounding volume or the axes of a bounding box), and adding to PVS_v all scene objects that intersect one of the extended bounding volumes (see Figure 5). The extension $e(o)$ can be calculated either from the density of all visibility samples intersecting the view cell (global extension $e_g(o)$), or from the density of samples that hit the object (local extension $e_l(o)$).

To estimate the *global extension* $e_g(o)$, we assume that n_v visibility samples are uniformly distributed on a sphere of radius $d(o)$, which is the distance of the object from the view cell. As extension we use half the approximated distance between two neighboring samples on this sphere: $e_g(o) = 2d(o)/\sqrt{n_v}$.

To estimate the *local extension* $e_l(o)$, we assume that the $n_v(o)$ visibility samples which hit the object are uniformly distributed on a disk of radius $r(o)$, which is the radius of the object bounding volume. As extension we use half the approximated distance between two neighboring samples on this disk: $e_l(o) = r(o)/\sqrt{n_v(o)}$.

Note that both estimations are not accurate: the global estimation ignores that rays due to the mutation-based strategy are not distributed uniformly, while the local estimation ignores that parts of the object may be occluded from the view cell, leading to an overestimation of e_l. In practice we therefore choose the minimum of the two estimations, and allow the user to increase or decrease the filter size with a constant k for more conservative or more aggressive results: $e(o) = k \cdot min(e_g(o), e_l(o))$.

View space filter. The view space filter is useful if the size of the view cells is relatively small (i.e., comparable to size of the objects). This filter is very simple: it merges the PVS of the given view cell with the PVSs of neighboring view cells, for example those with the most similar PVSs.

6 Dynamic Updates

Previous PVS-based methods were not applicable in interactive scenarios in which the scene gets manipulated. Our method can work in interactive sessions and so we also propose a method for updating the global visibility solution after scene edits.

Scene edits are implemented as follows: (1) When deleting an object, we remove it from the PVSs of all view cells. (2) When inserting an object O, we need to remove all objects from the current visibility solution that might be hidden by O: For each view cell v we construct a penumbra shadow volume of O and remove all objects of PVS_v which intersect the shadow volume (see Figure 6). (3) When editing an object (e.g. scaling, translation, mesh modification), we perform subsequent deletion and insertion of the object.

Note that before we continue with the visibility computation, we also need to update the ray casting data structures in order to reflect the scene edits.

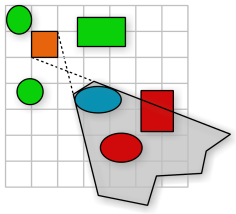

Figure 6: *Dynamic visibility updates: One object has been inserted (in blue). For a given view cell (in orange) the PVS objects intersecting the penumbra of the inserted object are removed from the PVS (in red). The remaining PVS objects (in green) can not be affected by the insertion operation.*

AGVS is suitable for dynamic edits because the visibility solution gets constantly refined, even without explicit treatment of the edits in the sampling process. The removed entries are quickly reinserted by our visibility sampling strategies (if they are still visible); in particular the mutation-based strategy will automatically focus on places in which the entries have been invalidated.

Dynamic updates are an important tool in global visibility analysis: a user can tentatively insert or edit an object (e.g., a wall or other blocker) and observe the effect this object has on the visibility solution and on render cost. This can be used to remove visibility hotspots in large scenes.

7 Results

We have evaluated the proposed method on five different scenes, which are depicted in Figure 8 (top row). Statistics including the number of triangles, objects, and view cells are shown in Table 1. The view cells are represented as kd-trees, the objects as bounding volume hierarchies (BVH). The view space kd-tree is generated according to the optimized construction described in [Mattausch et al. 2006]. The object BVH is built using the surface-area heuristics. The batch size for the computation was set to 1M samples, the buffer size for the mutation candidates was 2M samples.

The results were measured on a server with two Intel Xeon E5440 2.83GHz quad-core CPUs with 32GB of RAM. We used a custom BVH-based ray tracer which provides between 0.1M and 1M rays/s on the tested scenes. The ray tracer is about 30% slower than the state-of-the-art [Reshetov et al. 2005], but has very fast setup times (a few seconds for building a kd-tree for Arena, about 1-2 minutes for Powerplant), which is important for visibility analysis applications. Note that in the plots in this section, one sample always corresponds to two rays being cast, the forward and the reverse ray.

Scene	triangles	objects	view cells
Vienna	3,609,675	6,156	8,192
Arena	4,528,160	7,804	8,192
Pompeii	5,646,041	12,288	8,192
PowerPlant	12,748,510	10,150	8,192
Boeing 777	337,000,000	130,000	8,192

Table 1: *Statistics for all scenes.*

We have implemented the presented method in a multi-threaded application. Visibility is computed in the background by the visibility computation thread, while the GUI thread allows interactive manipulation/walkthrough using the current visibility solution. Figure 7 shows a small subset of the samples generated by the AGVS algorithm and demonstrates how the samples adapt to the visibility structure of the scene. Figure 2 shows an example how AGVS could be used for interactive visibility analysis. The accompanying video shows a real-time capture of such an analysis using our tool.

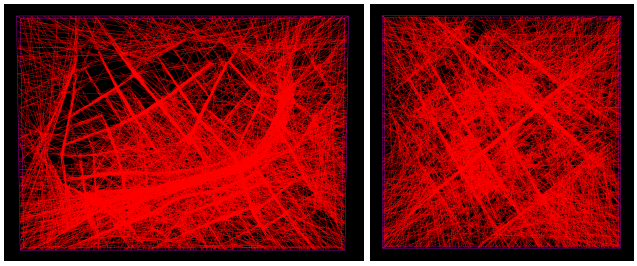

Figure 7: *Subset of rays generated by the AGVS algorithm after about 20M samples have been cast. Left: Vienna, Right: Pompeii.*

Progressive sampling efficiency. We are not aware of any published method that is able to compute global visibility progressively. Therefore as a basic reference technique (REF) we use a method which randomly selects a view cell and then casts a ray from a random point inside the view cell into a random direction. If an object is hit, we add this object to the PVS of the view cell. Similar to our new method, REF is fully progressive; as we cast more and more rays the PVSs become more accurate.

As a second reference we implemented a GPU-based method (GPU-REF), which is similar to the techniques used in industry [Hastings 2007]. This progressive method selects a random view point, renders the scene into a z-buffer of $1{,}024\times1{,}024$ pixels for six directions of the surrounding cube, and uses occlusion queries to discover visibility of the objects. NVIDIA's depth clamping functionality was used to avoid problems with the near clip plane. We evaluated the GPU-REF method on all but the Boeing 777 scene, as our GPU renderer is in-core and we can only run it on desktops, which do not have sufficient memory to store the Boeing 777 scene.

As a third reference, we use a global method that uses only the view space-direction distribution for sampling (SGVS for stationary global visibility sampling), similar to [Mattausch et al. 2006]. SGVS shows how much of the benefit of our method is due to the global view cell evaluation (when comparing to REF), and how much additional benefit is gained by the adaptive mixture distribution and the mutation-based sampling strategies (when comparing to AGVS).

Convergence. Figure 8 compares the convergence behavior of the four methods on our test scenes. We show the progressive evolution of the PVS size (measured as the average number of triangles in a PVS) with respect to the running time. Using the PVS size to compare the methods is a compromise: ideally, one would like to compare to the exact solution, and plot $|PVS|/|EVS|$. However, we are not aware of any algorithm that can provide an exact solution for the scenes we tested in reasonable time, and as shown by Wonka et al. [Wonka et al. 2006], the exact solution suffers from numerical errors similarly to the ray tracing solution. Therefore we believe that PVS sizes are a good measure for comparing the relative efficiencies of algorithms. As an alternative evaluation, we provide a comparison of pixel error later on. Note that we do not use visibility filtering in this test, so the AGVS curve does not overestimate the PVS.

The most important observation from these plots is that our proposed AGVS method is much more efficient in determining the PVS than the other methods. The AGVS method provides a PVS significantly larger than SGVS and GPU-REF, and 2-4 times larger than the one provided by REF. Note however that the factor between the PVSs provided by the method is *not* an indicator of the speedup or benefit of AGVS. Instead, one has to compare how much time it takes the methods to achieve the *same PVS size*. The curves clearly show that SGVS consistently takes more than one order of magnitude longer than AGVS to obtain the same average PVS size. When compared to REF and GPU-REF, AGVS provides up to two orders of magnitude speedup. Note that the positions marked by dashed lines in the plots correspond to a state where AGVS is still in a phase of steep increase, so if we extrapolated the curves to show the same comparison for a later phase, the benefit of AGVS over REF and GPU-REF would probably exceed two orders of magnitude. The figure shows clearly that convergence of the reference method compared to AGVS is so slow that it is almost imperceptible in the duration we have run the tests.

An interesting observation follows from the comparison to GPU-REF. Although in the same time the GPU-REF method is able to process more than one order of magnitude more samples than AGVS, the overall convergence of GPU-REF is significantly slower. The samples produced by GPU-REF are highly correlated (6M samples always intersect at a common view point) and thus they do not easily discover some difficult visibility interactions such as objects occluded from a view cell by nearby occluders. This can also be observed in the pixel error analysis shown later.

Influence of distributions. Figure 9, left, shows the mixture of distributions actually selected by the adaptive mixture distribution algorithm (Section 4.2) according to previous success rates (Vienna

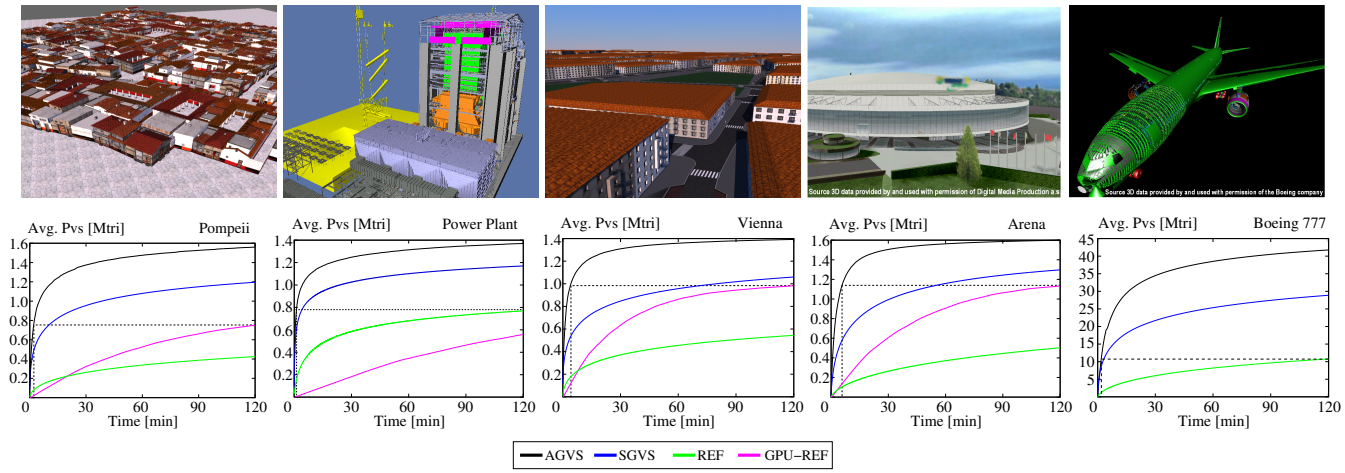

Figure 8: *The tested scenes (top row) and the analysis of the convergenece of the tested methods (bottom row). The dashed lines show reference points for comparing the benefit of AGVS over the better reference method (REF or GPU-REF), i.e., the factor of time needed to achieve the same PVS. Visibility filtering is* not *used in this test.*

model). The mutation-based strategies (shown as one curve) are the most important, closely followed by the stationary two-point distribution, which works well because it adapts to the shape of the scene.

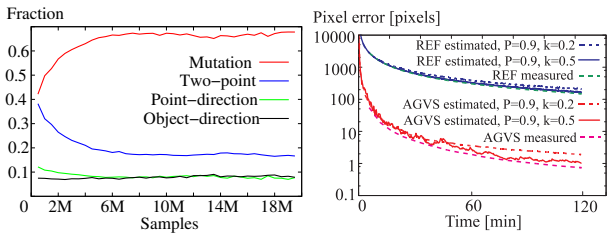

Figure 9: *Left: Ratio of distributions actually selected by the adaptive mixture distribution. Right: Pixel error estimation. The plot shows the estimated pixel error in a city scene in comparison with pixel error measured using 20,000 random views (1,024×1,024 pixels each).*

Pixel error estimator. A typical curve corresponding to the pixel error estimator described in Section 4.5 is depicted in Figure 9, right. For comparison we also show the reference error curve measured using offline pixel error evaluation using 20,000 cameras. Please note that the proposed estimator comes practically for free already during the computation, whereas the camera-based evaluation is a costly computation applied as a postprocess.

The pixel error estimator for the view point-direction distribution closely follows the measured behavior. The pixel error estimator for AGVS tends to be conservative as larger time windows must be used (the number of uniform samples cast is low as these samples are not very succesful in discovering new PVS entries). Note that using smaller k smooths the estimator, but makes it even more conservative as larger time windows extending to the past are used.

Practical behavior. For demonstrating the practical influence of the average PVS sizes shown above, we have measured the average and maximum pixel errors (i.e., number of incorrect pixels) for a subset of viewpoints (more specifically, an actual walkthrough). Note that there is as yet no feasible way to generate the "actual" PVS as a reference: exact visibility solvers do not work on scenes of this complexity, while the convergence of a brute-force reference solution is simply too slow to be feasible.

Figure 10 shows this for the Vienna scene, with pixel error evaluated using a walkthrough consisting of 1,282 view points. The first part of the walkthrough corresponds to walking on a street, the second part is a flyover sequence. After two hours of sampling, the average pixel error among all tested view points drops to 10 pixels on a 1,024×1,024 screen, while the corresponding maximum pixel error for the whole scene drops to 501 pixels. Note that these values were reached even without the visibility filter. The application of the visibility filter would typically reduce the error by an order of magnitude at the cost of increasing the PVS size by 50 to 150%. The increase of PVS size depends on the scene and the object representation. Although the visibility filter adapts to the sampling density (it uses a smaller kernel when more samples have already been cast), it typically does not converge to the unfiltered solution: if the objects overlap in space, filtering even with no extension at all adds overlapping objects, including those which need not be visible. Since the PVS increase can reduce the performance of the target application, we suggest to use the visibility filter only in the early stages of the computation to compensate for larger visibility errors. The decision for using/not-using the visibility filter can also be based on the estimated pixel error.

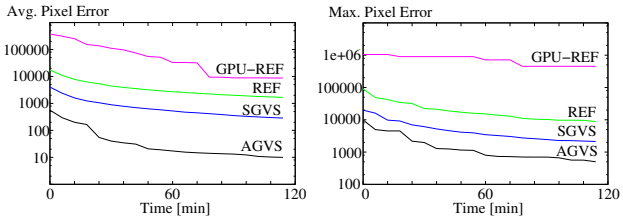

Figure 10: *Pixel error measurement for Vienna scene on a resolution of 1,024×1,024.*

Comparison with Guided Visibility Sampling. Guided Visibility Sampling (GVS) [Wonka et al. 2006] is an aggressive visibility algorithm that also uses ray tracing to sample visibility. A comparison between AGVS and GVS is difficult because GVS does not

work on object granularity and AGVS does not work on triangle granularity.

We applied GVS on the view cells corresponding to the walkthrough sequence used for the AGVS pixel error evaluation and converted the resulting triangle-based PVSs to the object-based representation used by AGVS. Note that the conversion of GVS results to object-based PVSs inflates the number of triangles in the PVS by about 50% to 100%. There were 33 view cells visited by the walkthrough. GVS took 17 minutes to calculate these 33 view cells using about 500M samples. The termination criterion in GVS was set to a threshold of 50 triangles per 1M rays [Wonka et al. 2006]. The error evaluation of the GVS solution lead to an average pixel error of 5.4 and a maximum pixel error of 424. Extrapolated to 8,192 view cells, GVS would have taken 62 hours to calculate visibility for the whole scene.

For pure AGVS, we have higher pixel errors after 17 minutes of computation (average 123 and maximum 4,519 – see Figure 10), but at this time the PVSs for all 8,192 view cells are already available. After two hours of computation the average pixel error of AGVS drops down to 10 and the maximum pixel error to 501. This shows that AGVS can also compete with GVS for applications where a saturated PVS is required.

Comparison with online occlusion culling. We implemented support for PVSs in a rendering engine. For each view point we tag objects in the corresponding PVS and use hierarchical view frustum culling to remove objects outside of the view frustum. As a reference for the comparison we used the CHC++ algorithm [Mattausch et al. 2008] – a state of the art online occlusion culling algorithm based on hardware occlusion queries. A render time comparison between View Frustum Culling (VFC), View Frustum Culling + PVSs (VFC+PVS), and CHC++ can be seen in Figure 11, for a walkthrough in Vienna and a walkthrough in Powerplant.

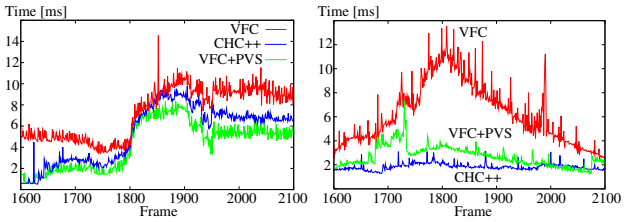

Figure 11: *Comparison of View Frustum Culling (VFC), CHC++, and View Frustum Culling + PVSs (VFC + PVS) in a walkthrough in Vienna (left) and Powerplant (right).*

We tested the algorithms on an Intel Core 2 2.66GHZ quad-core CPU (using only one core) and an NVIDIA GeForce 280 GTX GPU. For all tested scenarios both VFC+PVS and CHC++ are generally faster than plain VFC, while there are passages where either CHC++ or VFC+PVS performs better. Note that CHC++ usually renders fewer triangles since it computes visibility for a set of image samples from a given view point, whereas a PVS is computed for all view points in a view cell, thus overestimating each individual view point. It is an interesting topic for future work to analyze whether splitting view cells on demand during visibility sampling could improve this overestimation. On the other hand there is an additional cost associated with CHC++, which for some view points becomes more important than visibility overestimation associated with the view cell. For a reasonable number of view cells, we observed that rendering using precomputed PVSs is competitive to state-of-the-art online culling algorithms if a slight pixel error is tolerable. Visibility preprocessing is usually the best choice when visibility has to be known beforehand or a bound on visibility has to be guaranteed. We are also interested in combining precomputed visibility with occlusion queries in future work.

8 Discussion

Memory consumption. One advantage of our algorithm is that it maintains only a small and easily controllable state. In contrast to conservative and exact methods, no ray space or scene data structure is created to represent visibility. The memory consumption therefore depends on two main factors: 1) The total memory required for the complete visibility solution (i.e., all PVS entries for all view cells). This memory is influenced by the visibility structure of the scene (which cannot be changed) and by the view cell subdivision and object clustering (which can be changed). For example, in a scene with 4,000 view cells and 1,000 objects visible per view cell on average, the data structure would require 16MB of main memory at convergence (using 32 bit object identifiers). There are also algorithms for compressing PVS data if this becomes an issue [van de Panne and Stewart 1999]. 2) The memory required by the geometry of the scene and the ray tracer acceleration data structure. This is the only factor that limits the type of scene our algorithm can be applied to, and depends on the ray tracer being used. The Boeing 777 model, for example, requires 28GB of main memory including the BVH. For larger models, out-of-core ray tracing is an option that needs to be evaluated further.

Visibility coherence. Some recent algorithms have attempted to exploit the spatial coherence between objects [Nirenstein and Blake 2004; Laine 2005]. The fundamental difference is that these algorithms propagate occlusion, whereas our algorithm propagates visibility. However, neither hierarchical [Nirenstein and Blake 2004] nor sequential [Laine 2005] propagation of occlusion information lends itself to progressive computation, as this requires a complete visibility solution for a particular region to establish occlusion, whereas a single ray suffices to establish visibility. Note that algorithms that construct the PVS top-down proceed in a depth-first manner and are thus not progressive in our sense, while breadth-first traversal is most likely infeasible due to memory requirements.

Object-level visibility. Since AGVS calculates all view cells simultaneously, it is sensitive to the total number of objects in the scene, both in memory consumption for all PVSs, and in convergence speed. Therefore, a triangle-level solution as provided by Guided Visibility Sampling is typically not feasible. However, any application that requires a global visibility solution will need to compress triangle-level visibility into object-level visibility because of the high memory overhead of triangle-level PVSs. For real-time rendering applications, object-level visibility is also required because current graphics hardware works best on batches of triangles and not on individual triangles.

Accuracy. Wonka et al. [2006] discussed the issue of accuracy in the context of visibility processing and claim that accuracy is limited by numerical precision even for exact algorithms. Specific limitations concern the tendency of current ray tracers to shoot "through" an object if a ray pierces an edge that does not lie on the silhouette. We have found that in scenes consisting of closed objects, this artifact can be reduced by discarding rays that hit a backfacing triangle, but the development and evaluation of robust ray casters remains a topic of future work.

9 Conclusion

We described a new visibility algorithm that computes global visibility in the scene by calculating PVSs for all view cells simultaneously in a progressive fashion. The main contribution of the paper is a set of adaptive sampling strategies based on ray mutations that exploit the spatial coherence of visibility. The mutation based distributions are mixed with other heuristic distributions using the adaptive mixture distribution technique. We have shown that our algorithm achieves more than an order of magnitude speedup compared to sequential per-view cell visibility computation. We believe that the Adaptive Global Visibility Sampling algorithm breaks new grounds in the applicability of visibility algorithms. In addition to making preprocessed visibility feasible for everyday use, it also enables new applications like visibility analysis for level design.

Acknowledgments

This work has been supported by the Austrian Science Fund (FWF) contract P21130-N13, Czech MŠMT programs no. LC-06008 and MSM 6840770014, the Aktion Kontakt 2009/6, and the NSF.

References

AIREY, J. M., ROHLF, J. H., AND BROOKS, JR., F. P. 1990. Towards image realism with interactive update rates in complex virtual building environments. In *Computer Graphics (1990 Symposium on Interactive 3D Graphics) 24*, 2, 41–50.

BITTNER, J., WONKA, P., AND WIMMER, M. 2001. Visibility preprocessing for urban scenes using line space subdivision. In *Proc. of Pacific Graphics '01*, 276–284.

BITTNER, J. 2003. *Hierarchical Techniques for Visibility Computations*. PhD thesis, Czech Technical University in Prague.

COHEN-OR, D., FIBICH, G., HALPERIN, D., AND ZADICARIO, E. 1998. Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. *Computer Graphics Forum (Eurographics '98) 17*, 3, 243–254.

COHEN-OR, D., CHRYSANTHOU, Y. L., SILVA, C. T., AND DURAND, F. 2003. A survey of visibility for walkthrough applications. *IEEE Trans. on Visualization and Computer Graphics 9*, 3, 412–431.

DUGUET, F., AND DRETTAKIS, G. 2002. Robust epsilon visibility. *ACM Transactions on Graphics 21*, 3, 567–575.

DURAND, F., DRETTAKIS, G., THOLLOT, J., AND PUECH, C. 2000. Conservative visibility preprocessing using extended projections. In *Proc. of SIGGRAPH '00*, 239–248.

DUTRÉ, P., BALA, K., AND BEKAERT, P. 2003. *Advanced Global Illumination*. AK Peters.

GOLDBERG, D. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

GOTSMAN, C., SUDARSKY, O., AND FAYMAN, J. A. 1999. Optimized occlusion culling using five-dimensional subdivision. *Computers and Graphics 23*, 5, 645–654.

HASTINGS, A., 2007. Occlusion systems. Insomniac Games Tech Presentation. http://www.insomniacgames.com/tech/articles/1107/occlusion.php.

HAUMONT, D., MÄKINEN, O., AND NIRENSTEIN, S. 2005. A low dimensional framework for exact polygon-to-polygon occlusion queries. In *Rendering Techniques '05*, 211–222.

KOLTUN, V., CHRYSANTHOU, Y., AND COHEN-OR, C.-O. 2001. Hardware-accelerated from-region visibility using a dual ray space. In *Rendering Techniques '01*, 205–216.

LAINE, S. 2005. A general algorithm for output-sensitive visibility preprocessing. In *Proc. of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 31–40.

LEPAGE, G. 1980. Vegas: An adaptive multidimensional integration program. Tech. Rep. CLNS-80/447, Cornell University.

LEVOY, M., AND HANRAHAN, P. 1996. Light field rendering. In *Proc. of SIGGRAPH '96*, 31–42.

LEYVAND, T., SORKINE, O., AND COHEN-OR, D. 2003. Ray space factorization for from-region visibility. *ACM Transactions on Graphics 22*, 3, 595–604.

MATTAUSCH, O., BITTNER, J., AND WIMMER, M. 2006. Adaptive visibility-driven view cell construction. In *Rendering Techniques '06*, 195–206.

MATTAUSCH, O., BITTNER, J., AND WIMMER, M. 2008. CHC++: Coherent hierarchical culling revisited. *Computer Graphics Forum (Eurographics '08) 27*, 3 (Apr.), 221–230.

NIRENSTEIN, S., AND BLAKE, E. 2004. Hardware accelerated visibility preprocessing using adaptive sampling. In *Rendering Techniques '04*, 207–216.

NIRENSTEIN, S., BLAKE, E., AND GAIN, J. 2002. Exact from-region visibility culling. In *Rendering Techniques '02*, 191–202.

PITO, R. 1999. A solution to the next best view problem for automated surface acquisition. *IEEE Trans. Pattern Anal. Mach. Intell. 21*, 10, 1016–1030.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. *ACM Transactions on Graphics 24*, 3, 1176–1185.

SCHAUFLER, G., DORSEY, J., DECORET, X., AND SILLION, F. 2000. Conservative volumetric visibility with occluder fusion. In *Proc. of SIGGRAPH '00*, 229–238.

SHIRLEY, P., SLUSALLEK, P., WALD, I., MARK, B., STOLL, G., AND MANOCHA, D. 2006. SIGGRAPH 2006 course 4, State of the art in interactive ray tracing.

TELLER, S. J., AND SÉQUIN, C. H. 1991. Visibility preprocessing for interactive walkthroughs. In *Computer Graphics (Proc. of SIGGRAPH '91)*, 61–69.

THOMPSON, S. K., AND SEBER, G. A. F. 1996. *Adaptive Sampling*. Wiley.

VAN DE PANNE, M., AND STEWART, A. J. 1999. Effective compression techniques for precomputed visibility. In *Rendering Techniques '99*, 305–316.

VEACH, E., AND GUIBAS, L. J. 1997. Metropolis light transport. In *Proc. of SIGGRAPH '97*, 65–76.

WILSON, A., AND MANOCHA, D. 2003. Simplifying complex environments using incremental textured depth meshes. *ACM Transactions on Graphics 22*, 3, 678–688.

WONKA, P., WIMMER, M., AND SCHMALSTIEG, D. 2000. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Rendering Techniques '00*, 71–82.

WONKA, P., WIMMER, M., ZHOU, K., MAIERHOFER, S., HESINA, G., AND RESHETOV, A. 2006. Guided visibility sampling. *ACM Transactions on Graphics 25*, 3, 494–502.