Guided Visibility Sampling++

THOMAS KOCH and MICHAEL WIMMER, TU Wien, Austria

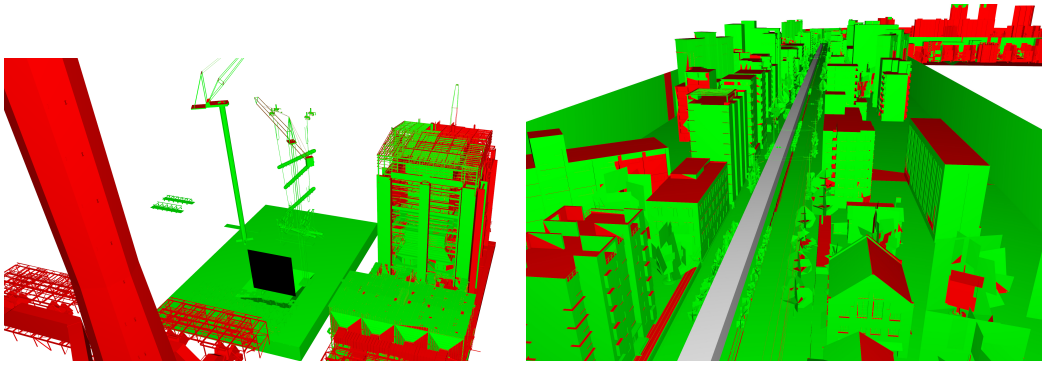


Fig. 1. Triangles that are found to be visible by Guided Visibility Sampling++ (GVS++) are shaded green. GVS++ uses hardware-accelerated ray tracing, achieves low error rates, and is over four orders of magnitude faster than the CPU-based Guided Visibility Sampling implementation. Left: A single rectangular view cell (black region) on the UNC power plant model [UNC 2001]. Right: Multiple rectangular view cells combined to a box-shaped view cell on a city model ©VIRES Simulationstechnologie GmbH.

Visibility computation is a common problem in the field of computer graphics. Examples include occlusion culling, where parts of the scene are culled away, or global illumination simulations, which are based on the mutual visibility of pairs of points to calculate lighting. In this paper, an aggressive from-region visibility technique called Guided Visibility Sampling++ (GVS++) is presented. The proposed technique improves the Guided Visibility Sampling algorithm through improved sampling strategies, thus achieving low error rates on various scenes, and being over four orders of magnitude faster than the original CPU-based Guided Visibility Sampling implementation. We present sampling strategies that adaptively compute sample locations and use ray casting to determine a set of triangles visible from a flat or volumetric rectangular region in space. This set is called a potentially visible set (PVS). Based on initial random sampling, subsequent exploration phases progressively grow an intermediate solution. A termination criterion is used to terminate the PVS search. A modern implementation using the Vulkan graphics API and RTX ray tracing is discussed. Furthermore, we show optimizations that allow for an implementation that is over 20 times faster than a naive implementation.

CCS Concepts: • **Computing methodologies** → **Visibility**.

Additional Key Words and Phrases: visibility, occlusion culling, potentially visible set, visibility sampling

ACM Reference Format:
Thomas Koch and Michael Wimmer. 2021. Guided Visibility Sampling++. *Proc. ACM Comput. Graph. Interact. Tech.* 4, 1 (May 2021), 16 pages. https://doi.org/10.1145/3451266

Authors' address: Thomas Koch, thomas-koch@kabelplus.at; Michael Wimmer, wimmer@cg.tuwien.ac.at, TU Wien, Vienna, Austria.

1 INTRODUCTION

Visibility is a fundamental problem, not just in computer graphics but also in robotics, computer vision, and architecture, to name a few. There are various approaches for applications that rely on visibility calculations. Examples in computer graphics include global illumination simulations, which are based on the mutual visibility of pairs of points in the scene to calculate lighting and shadow generation, where the visibility of a point to a light source has to be calculated. In real-time rendering applications, visibility from a viewpoint is commonly determined using the z-buffer. Visibility may also be solved using ray casting, where rays are shot from a viewpoint through each pixel, finding the closest primitives along each ray.

Another common application for visibility algorithms is occlusion culling. Occlusion culling is an acceleration technique, where parts of the scene that are hidden by other parts in the scene are culled. This way, hidden geometry does not have to go through the graphics pipeline. In general, occlusion culling techniques can be categorized into methods that calculate visibility from a point or from a region [Bittner and Wonka 2003]. From-region visibility is more expensive to compute since it is necessary to determine a point's visibility from any position of a given region. Therefore, from-region visibility is often precomputed. Some approaches utilize well-known real-time rendering techniques, such as the z-buffer [Greene et al. 1993]. At the same time, other methods use techniques such as ray casting to determine the visibility of objects and primitives, such as the aggressive from-region sampling-based visibility approach *Guided Visibility Sampling* (GVS) by Wonka et al. [Wonka et al. 2006].

Due to the recent developments in hardware-accelerated ray tracing, it is worth revisiting ray casting-based approaches such as GVS. In late 2018, Nvidia introduced the first GPU architecture, Turing [NVIDIA 2018], that supports hardware-accelerated ray tracing. Two years later, Nvidia released their second generation of GPUs [NVIDIA 2020] that support real-time ray tracing, which also allows compute workloads, such as hardware-accelerated ray tracing, as well as graphics workloads, to be executed concurrently. Both architectures employ dedicated hardware units to accelerate ray tracing tasks such as bounding volume hierarchy traversal and ray/triangle intersection testing. Recent graphics APIs, such as Vulkan and DirectX, offer ray tracing APIs that allow developers to utilize hardware-accelerated ray tracing.

In this work, we present an aggressive from-region visibility algorithm called *Guided Visibility Sampling*++ (GVS++) that uses ray casting and intelligent sampling schemes for visibility determination and works on general 3D scenes. Our algorithm builds upon the work of Wonka et al. [Wonka et al. 2006]. We provide a publicly available Vulkan implementation that uses Vulkan's ray tracing API. We analyze the efficiency of the algorithm on various test scenes and provide a comparison to similar approaches. Our contributions can be summarized as follows:

- Guided Visibility Sampling++, an aggressive from-region sampling-based visibility approach based on *Guided Visibility Sampling* (GVS) [Wonka et al. 2006]. GVS++ is more accurate and offers more flexibility than GVS. Low error rates are achieved by intelligent sampling schemes that find new triangles and parallelize well.
- A publicly available Vulkan implementation of our algorithm that uses hardware-accelerated ray tracing. Using RTX ray tracing, our algorithm is over four orders of magnitude faster than the original CPU-based GPU implementation.
- An in-depth analysis of GVS++ on multiple scenes and a comparison to a brute-force random sampling approach, a rasterization-based from-region visibility technique, and the GVS algorithm by Wonka et al. [Wonka et al. 2006].

2 PREVIOUS WORK

There are extensive surveys [Bittner and Wonka 2003; Cohen-Or et al. 2003; Durand 1999] covering visibility algorithms. Typically, authors distinguish between from-region and from-point algorithms, which either operate in object or image space. Visibility algorithms are commonly classified as *aggressive*, *conservative*, *approximative* or *exact* [Nirenstein et al. 2002] algorithms, depending on the PVS that is produced. In the following, the focus is on from-region visibility algorithms.

Conservative Techniques. Durand et al. [Durand et al. 2000] presented a conservative visibility technique for general scenes. Objects are projected onto the image plane to determine whether an object is occluded. This idea is based on point-based visibility approaches. Schaufler et al. [Schaufler et al. 2000] *fuse* occluders and also extend occluders into empty space to classify regions in space as occluded. This approach also allows checking whether moving objects or objects that were not part of the original scene are visible. Leyvand et al. [Leyvand et al. 2003] note that from-region visibility is inherently 4D. This is because a ray effectively leaves and enters the view cell and a target region through 2D surfaces. Therefore, the authors presented a factorization of the 4D visibility problem into 2D vertical and horizontal components. The visibility calculation is based on merging the umbrae of objects intersected by vertical planes, comparable to the occluder fusion by Schaufler et al. [Schaufler et al. 2000]. In a more recent approach, Hladky et al. [Hladky et al. 2019] introduce the *camera offset space* to calculate under which camera offsets within a view cell a stored triangle is visible.

Aggressive Techniques. Nirenstein et al. [Nirenstein and Blake 2004] presented an aggressive visibility algorithm that operates in image space. In this approach, the PVS of a given view cell is determined by recording the triangles that are visible when rendering the scene from multiple viewpoints on the view cell. In contrast, Wonka et al. [Wonka et al. 2006] use ray casting and a combination of intelligent sampling strategies to efficiently find triangles, comparable to a flood fill algorithm. Bittner et al. [Bittner et al. 2009] also use ray casting. In this approach, the PVSs of multiple view cells are calculated simultaneously. The idea is to let a ray contribute to each view cell it intersects. In a recent approach, Ho et al. [Ho et al. 2012] developed an image space sampling algorithm that builds upon the work of Nirenstein et al. [Nirenstein and Blake 2004]. The authors present an importance sampling scheme in image space that places hemicubes on the view cell such that new primitives are more likely to be found.

Approximative Techniques. Early on, Airey et al. [Airey et al. 1990] proposed a from-region visibility algorithm for indoor architectural scenes. A scene is automatically subdivided into view cells, e.g., such that each room is covered by one view cell. The idea is that the set of visible primitives is mostly the same for most viewpoints within a room but changes more rapidly near *portals*, e.g., doors or windows. Later, Gotsman et al. [Gotsman et al. 1999] presented an approximative visibility algorithm for general scenes. Similar to previously discussed techniques, ray casting is used for visibility computation. The difference is that the visibility of objects instead of individual triangles is determined. Objects are sampled at uniformly distributed random locations and are only regarded as visible if they are visible from a non-negligible part of the view cell.

Exact Techniques. Teller and Séquin [Teller and Séquin 1991] presented a visibility algorithm based on the idea of using cells and portals by Airey et al. [Airey et al. 1990]. In this approach, an adjacency graph of neighboring cells that are reachable through portals is built in a preprocessing stage. By finding unobstructed sightlines through sequences of portals, visible cells from a viewpoint can be determined. Another, more recent cells and portals technique [Bittner et al. 2005] employs *line space*, which is a dual space in which each line in primal space corresponds to a point, to

determine visibility. The visibility computation is based on the idea that rays that intersect an occluder form a blocker polygon in line space. Bittner [Bittner 2002] uses line space and Plücker coordinates to calculate exact-from region visibility.

3 OVERVIEW

Our algorithm, called *Guided Visibility Sampling++* (GVS++), determines the *potentially visible set* (PVS) of triangles visible from a region in space, called *view cell*. The algorithm is *aggressive* [Nirenstein et al. 2002], meaning that the resulting PVS is a subset of the exact visibility solution. GVS++ does not rely on additional triangle connectivity information.

The algorithm uses ray shooting to sample the scene and is based on the GVS algorithm by Wonka et al. [Wonka et al. 2006]. The idea is to use intelligent sampling strategies guided by previously found triangles to sample new triangles efficiently. The general procedure of GVS++ is similar to that of GVS: An initial set of triangles is found by randomly sampling the scene. Found triangles act as seed points during the subsequent exploration phase, which employs two intelligent sampling strategies to find further triangles: *Adaptive border sampling* (ABS) and *reverse sampling* (RS). Adaptive border sampling samples a triangle's neighborhood for new triangles and can be compared to a flood fill algorithm. The reverse sampling algorithm is used to sample discontinuities.

Compared to the approach by Wonka et al. [Wonka et al. 2006], GVS++ employs an improved adaptive border sampling and reverse sampling algorithm. Various changes and improvements were made to increase the overall accuracy of the algorithm. Changes are motivated by shortcomings of the original algorithm and the fact that current hardware is more capable and offers different features such as hardware-accelerated ray tracing than hardware during the development of the GVS algorithm. Our sampling schemes take the highly parallel nature of modern GPUs into account and addresses shortcomings of GVS. The sample location computation is independent of the sampling result of other samples. This allows a modern GPU implementation to rapidly sample in parallel. The main advantage of GVS++ over GVS is that the resulting PVS is a more accurate estimation of the exact visible set resulting in lower average and maximum pixel errors in the final image. Also, GVS++ produces a highly converged PVS in under one second for all tested scenes.

Our method can be used in various applications. Many current real-time rendering applications such as game engines already use modern APIs for real-time ray tracing, therefore requiring only little extra work to include our solution. Our technique is a viable solution for modeling and level-creation tools to generate fast previews of scene visibility. It is also accurate enough to generate a final PVS-based visibility solution. Another application scenario includes networked visibility [Wonka et al. 2001], where a server calculates the visibility and continuously streams the found PVS to its clients. Further applications are visibility in urban planning and line-of-sight analysis.

4 APPROACH

The approach starts by randomly sampling the scene (see Section 4.1). Triangles that are found are added to a queue. In the subsequent exploration phase, triangles in the queue are processed by the adaptive border sampling algorithm to find neighboring triangles (see Section 4.2). A mechanism is used to find discontinuities during the adaptive border sampling, which are then handled by the reverse sampling algorithm (see Section 4.3). Once the queue is empty, the whole algorithm starts anew by randomly sampling the scene again. A termination criterion is used to terminate the PVS search (see Section 4.4).

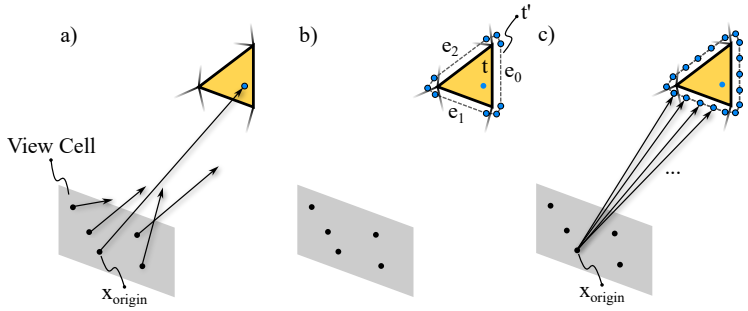


Fig. 2. The adaptive border sampling procedure is shown. Random sampling is used to find initial triangles (seed points) (a). Adaptive border sampling enlarges the boundary of the current triangle t to get a 9-vertices polygon t' (b). Additional samples are placed along the edges e_0, e_1 and e_2 of t'. Starting from x_{origin}, rays are traced to the nine vertices that form t' and to the samples along the edges e_i (c).

4.1 Random Sampling

The algorithm starts by sampling the scene using a pseudo-random sampling approach. Rays emanating from uniformly distributed pseudo-random locations on a 2D quad in space are intersected with the scene. Such a 2D quad is called *view cell*. A position on the view cell is determined by mapping randomly shifted standard Halton [Bhat 2003] points onto the view cell:

$$x_{\text{origin}} = v_p + h_{a,i} * v_{\text{sx}} * v_x + h_{b,i} * v_{\text{sy}} * v_y, \tag{1}$$

where v_p is the position of the view cell, $h_{a,i}$ is the i-th point of the Halton sequence h_a with base a, v_s is the size of the view cell and v_x and v_y form the two-dimensional coordinate frame of the view cell. Similar to the random sampling approach of GVS, Halton points are also used to calculate a pseudo-random direction:

$$\begin{aligned} \phi &= 2\,\pi\,h_{c,i} \\ r &= \sqrt{\max(1 - h_{d,i} * h_{d,i},\ 0)} \\ x_{\text{dir}} &= (r * \cos\phi,\ r * \sin\phi,\ h_{d,i}) \end{aligned} \tag{2}$$

4.2 Adaptive Border Sampling

The goal of the adaptive border sampling algorithm is to quickly find new triangles by thoroughly exploring the neighborhood of a triangle. The algorithm takes a newly found triangle t and constructs an enlarged polygon t'. The neighborhood of t is explored by sampling at the vertices and along the edges of t'. Wonka et al. [Wonka et al. 2006] used a sampling strategy that obtains sample points by recursively subdividing the edges of t': An edge is subdivided if sampling adjacent sample points finds different triangles. However, this can lead to undersampled edges as illustrated in Figure 3. This problem is alleviated by replacing the conditional recursive subdivision with a fixed number of samples placed along the edges of the border polygon. This way, the edges of the border polygon are always sampled, independently of the sampling result of other border polygon samples. The placement of the nine vertices that form the border polygon follows the idea of Wonka et al. [Wonka et al. 2006]. This change makes an efficient GPU implementation possible, allowing us to sample the fixed positions along the border polygon's edges in parallel. The whole ABS procedure is shown in Figure 2.

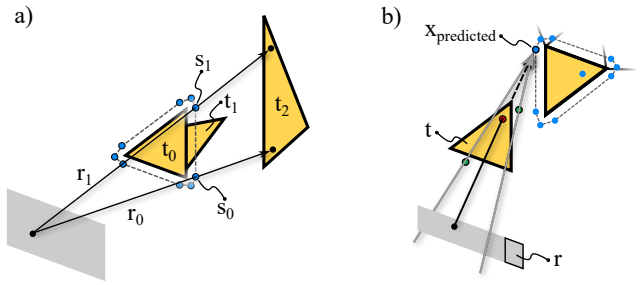


Fig. 3. Shortcomings of the adaptive border sampling (a) and reverse sampling (b) algorithm of GVS that are addressed by GVS++. During adaptive border sampling (a), the edge (s_0, s_1) is not further recursively sampled if the same triangle t_2 is found when sampling the border polygon of a triangle t_0 at position s_0 and s_1. This leads to undersampled edges and therefore triangles (t_1) can be missed. The reverse sampling algorithm (b) of GVS mutates rays such that an occluding triangle t is passed. However, with narrow view cells or large occluding triangles, mutated ray origins may lie outside the view cell. Region r on the view cell would contain valid positions to sample $x_{\text{predicted}}$.

4.3 Reverse Sampling

The reverse sampling algorithm is used to sample discontinuities and regions of space that have not been discovered. The idea is that once a discontinuity is detected during the adaptive border sampling, the origin of the respective ray is mutated such that the ray penetrates the discontinuity.

A discontinuity is detected as follows. Let t be a triangle that is currently processed by the adaptive border sampling algorithm. A discontinuity occurs, if a ray through a vertex of the border polygon t' of t intersects a triangle t_o that is closer than the target point x_{target}:

$$|x_{\text{target}} - x_{\text{origin}}| - |x_{\text{hit}} - x_{\text{origin}}| > \Delta$$

The reverse sampling algorithm also identifies discontinuities where a ray cast during adaptive border sampling does not intersect any triangle at all. Once a discontinuity is detected, the ray's origin is mutated. The mutation strategy presented by Wonka et al. [Wonka et al. 2006] may generate mutated rays that do not intersect the view cell, leading to detected discontinuities not being sampled as illustrated in Figure 3.

Our approach mutates a ray's starting point based on sample locations distributed on the view cell. Sample locations are distributed along the edges and on the corners of the view cell. Additionally, points are uniformly placed on the surface of the view cell based on Halton sequences. When processing discontinuities caused by a closer triangle t_0, the occluding triangle t_0 is projected onto the view cell to avoid intersecting it again. If a sample point lies within the projected triangle p_0, a ray from such a point to x_{target} would intersect t_0. Therefore, rays are only traced from points that are not within the boundaries of p_0 (see Figure 4).

The occluding triangle t_0 is projected onto the view cell by constructing lines originating at x_{target} through the vertices of t_0. The lines are intersected with the plane of the view cell to get the projected triangle p_0 with coordinates (A, B, C). Baryzentric coordinates are used to determine whether a sample point on the view cell is within p_0.

4.4 Termination Criterion

A termination criterion is used to decide when the visibility sampling should be terminated, and the PVS is considered to be converged. Different termination criteria may be used that take the PVS size, the total number of traced rays, or the rate at which the PVS grows into account. In our

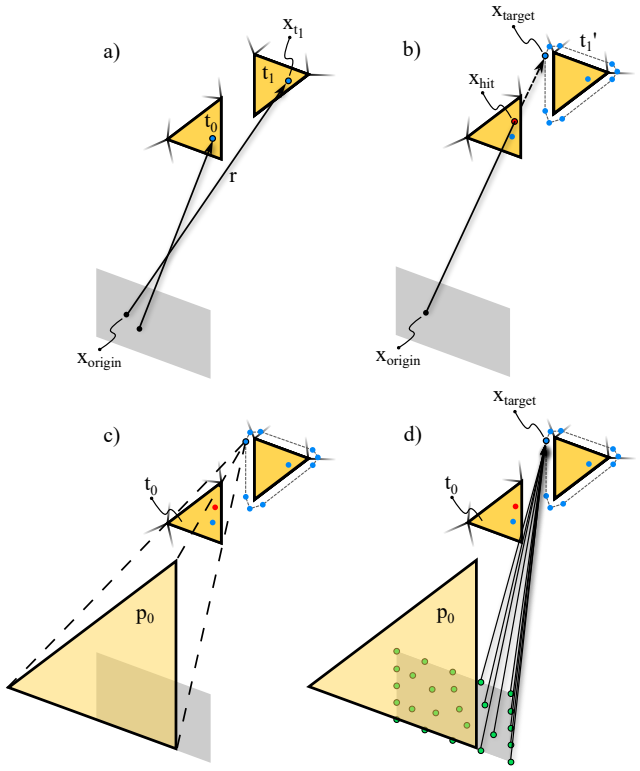


Fig. 4. The reverse sampling procedure is shown for the case where a discontinuity is detected caused by a closer triangle. Triangle t_0 and t_1 are sampled and added to the queue (a). Adaptive border sampling is executed for t_1, sampling x_{target} of the border polygon t_1'. The ray intersects the closer triangle t_0 at position x_{hit}. A discontinuity is detected, since the intersection point is closer to the ray origin than the sample point x_{target} (b). The occluding triangle t_0 is projected onto the view cell from the viewpoint of x_{target} to get the projected triangle p_0 (c). Samples are distributed uniformly on the view cell and along its edges. For each sample on the view cell that is not within the bounds of p_0, a ray is traced to x_{target} (d).

implementation, the termination criterion is checked after the exploration phase. We test whether the number of found triangles is below a threshold T, e.g., 10 or 50. If the number of found triangles is below T, the algorithm is terminated, and the PVS is considered converged. Otherwise, the algorithm starts anew by, again, randomly sampling the scene followed by an exploration phase. A different termination criterion where the number of newly found triangles per n rays is checked may be used instead. The termination criterion may also be checked more frequently, for instance, after each execution of adaptive border sampling and reverse sampling.

4.5 Data Structures and Optimizations

Our algorithms allow an efficient parallelization of the sampling procedure, so that the main workload, i.e., the sampling of the scene, can be executed on the GPU. However, a naive implementation significantly impedes performance as it requires regular GPU-CPU communication and frequent costly set operations on the CPU. Whenever a triangle is intersected on the GPU, its ID and accompanying intersection data is stored in a buffer on the device. Once the shader finishes execution,

the buffer is transferred to the host. Note that this buffer may contain duplicate elements. On the host-side, a *set* data structure is used to store the PVS. The intersected triangles are inserted into the PVS. Newly intersected triangles are also inserted into a queue for further processing in an exploration phase. This naive approach is over 20 times slower than our optimized implementation. This is due to the expensive set operations on the host side during which the GPU is idle, since further exploration phases depend on the result of the set operations. Therefore, to maximize performance, the goal should be to keep PVS maintenance completely on the GPU.

In our optimized implementation, we achieve minimal GPU idle times by keeping the frequency and amount of data transferred between the CPU and the GPU to a minimum. Furthermore, we avoid expensive set operations on the CPU side by ensuring uniqueness of stored elements on the GPU.

Instead of storing the PVS on the CPU, we store it in VRAM on the GPU to allow for fast read and write access by the shaders. Furthermore, the buffer storing the intersected triangle IDs and intersection data is allocated from memory on the host system, which also allows access from the GPU. This way, less data has to be transferred between the device and the host, resulting in a 3× speedup compared to keeping this buffer in VRAM and frequently transferring it between GPU and CPU. Whenever a triangle is intersected, our implementation checks whether it is already stored in the PVS via an atomic compare-and-swap operation. If the triangle is sampled for the first time, it is inserted into the PVS and into the buffer that is later accessed by the host. This buffer's content is inserted into a queue on the CPU-side. The queue is realized as a dynamic size array to allow efficient insertion of the whole buffer at once. No set is required since the content of the buffer does not contain any duplicate elements. We further reduce CPU-GPU communication times by submitting data transfer workloads to a transfer-only queue to ensure maximum transfer speeds.

5 IMPLEMENTATION

The implementation of our algorithm is based on the Vulkan graphics API, using the `VK_NV_ray_tracing` Vulkan extension for hardware-accelerated ray tracing. The main GVS++ algorithm is implemented as two shaders. One shader contains the random sampling logic, while another shader implements the adaptive border sampling and the reverse sampling algorithm. The PVS is stored on the device, while the CPU keeps a queue that stores triangle data processed during the exploration phase.

A high-level overview of the whole system is given in Figure 5. The random-sampling shader uses Halton points that are computed beforehand to compute ray origins and directions on the view cell. The adaptive border sampling and reverse sampling shader, i.e., the exploration phase, are then executed until the queue of triangles that have not been processed by the exploration phase is empty. The termination criterion is then checked before proceeding with randomly sampling the scene or terminating the GVS++ algorithm.

Parameters. Various parameters can be changed to change the sampling behavior of the GVS++ algorithm. This allows an application to fine-tune our algorithm, if necessary. However, we found that a standard set of parameters gave good results on various scenes and view cell placements. Our implementation exposes T, the termination threshold, and n_{rand}, the number of samples used for one execution of the random sampling shader. For the adaptive border sampling, the parameters Δ_{ABS}, the distance of an enlarged border polygon to the original triangle, and n_{ABSEdge}, the number of samples along an edge of the border polygon are exposed. The sampling behavior of the reverse sampling algorithm is defined by n_{RSEdge}, the number of samples along each edge of the view cell, and n_{RSArea}, the number of samples on the view cell.

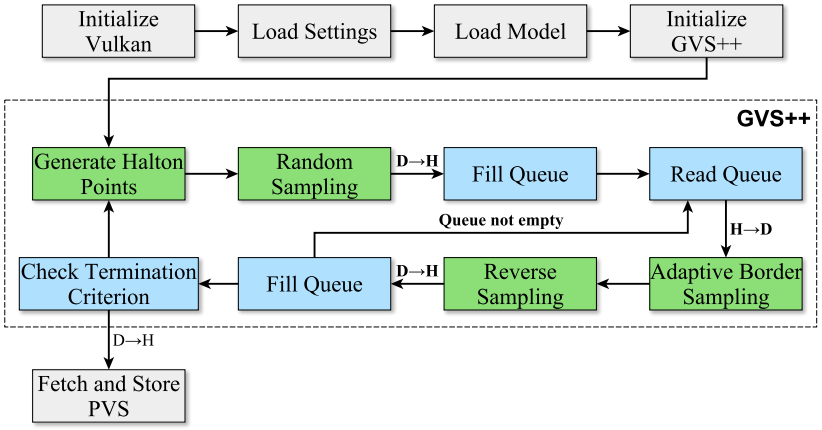


Fig. 5. An overview of the implementation of GVS++. Green boxes represent components that are implemented on the device. Blue boxes are components that are implemented on the host side. The Halton point generation, as well as random sampling, are implemented as separate shaders. The adaptive border and reverse sampling are realized as a single common shader. D⟶H and H⟶D show that data is transferred from device to host and vice versa.

6 RESULTS

To quantify the quality of the resulting PVS, the *pixel error* [Nirenstein and Blake 2004] is measured. The pixel error is determined by calculating the number of wrong pixels from 2000 different viewpoints on the view cell rendered in a resolution of 1000×1000. The viewpoints on the view cell are pseudo-randomly distributed, using randomly shifted standard Halton sequences [Bhat 2003]. The maximum pixel error over all viewpoints and the average pixel error of each view cell is calculated. A connected region of wrong pixels may be more noticeable than individual wrong pixels that cause the same pixel error. Therefore, the average and maximum number and size of connected pixel error regions are calculated as well.

Four different scenes have been selected for the evaluation: CANYON (2 242 504 triangles), a model of the Grand Canyon, PPLANT (12 748 210 triangles), the UNC powerplant model [UNC 2001], GERMANY (3 667 284 triangles), a city model ©VIRES Simulationstechnologie GmbH, and BISTRO [Lumberyard 2017] (5 657 097 triangles), which has been modified to include the hairball model by Laine et al. [Laine and Karras 2010]. The PPLANT scene consists of the most triangles of the four scenes and includes heavily occluded regions. GERMANY is a typical city scene with long streets. This scene is used to cover the case of far away triangles, i.e., triangles at the other end of a street. BISTRO is a small but detailed excerpt from a city. Due to the hairball model's inclusion, this scene also contains highly occluded regions of small structures.

The following results were produced using an AMD Ryzen 9 3900X CPU, Nvidia GeForce RTX 3080 GPU and 32GB of RAM.

6.1 Overview

In this section, GVS++ is compared to other from-region visibility sampling algorithms on four different scenes. We use two versions of GVS as baseline: the original CPU-based implementation, which uses MLRTA [Reshetov et al. 2005] for ray tracing (*CPU-GVS*), and a GPU-based implementation that uses hardware-accelerated ray tracing via the Vulkan API for the visibility evaluation

(*GPU-GVS*). Furthermore, we compare our approach to a brute-force random sampling algorithm, RAND, and a rasterization-based visibility sampling algorithm, RASTER. RAND is based on the implementation of the GVS++ algorithm but does not use any intelligent sampling mechanisms. RASTER is based on the visibility sampling algorithm by Nirenstein et al. [Nirenstein and Blake 2004]. In this approach, the scene is rendered from hemicubes that are intelligently distributed on the view cell. The IDs of the rendered primitives are gathered to populate the PVS. One notable difference between RASTER and GVS++ is that the runtime of the former approach inherently depends on the render resolution of the framebuffer that is used. On our four test scenes, however, we found that distributing hemicubes uniformly based on Halton points gives results that are on-par or better than when the adaptive placement is used in the original paper. Therefore, RASTER places hemicubes uniformly on the view cell, according to Halton points. There are numerous techniques to render multiple views efficiently [Unterguggenberger et al. 2020]. In our RASTER implementation, multi-view rendering is used, which is supported by Vulkan and allows rendering to multiple viewports simultaneously.

Pixel error and runtime measurements of the four algorithms on our scenes can be seen in Table 1. GVS++ produces the most accurate PVS and achieves the lowest average and maximum pixel errors on every scene among all of the tested algorithms. As seen in Table 1, the largest connected error region is, on average, 46% smaller than the maximum reported pixel error. This shows that in a given rendering, there are, on average, multiple smaller error regions instead of a single error region that contains all of the wrong pixels. We argue that a single large region is more noticeable than multiple smaller erroneous regions. Comparing GVS++ to GVS shows that GVS++ achieves significantly lower average and maximum pixel errors.

Figures 6(a) and 6(b) show that GVS++ converges more rapidly than GPU-GVS, RAND, and RASTER, and delivers a well-converged PVS in under one second. Overall, GVS++ finds more triangles per ray than GPU-GVS. This is mostly due to our improved RS algorithm that finds significantly more triangles than GVS RS. The behavior for different parameter configurations of our RS algorithm can be seen in Figure 6(d). We found that a higher number of samples for RS should be preferred, especially for scenes with highly occluded regions such as PPLANT and BISTRO. Comparing our ABS algorithm to GVS ABS shows that overall a comparable number of triangles are found but GVS ABS intersects more new triangles per ray. However, our ABS algorithm offers better parallelizability (Figure 7) which allows tracing more ABS rays and therefore finding more triangles within the same time frame. The overall efficiency improvement of our technique over GVS can also be seen when comparing the growth rate of the PVS during the random sampling phases of GVS++ and GPU-GVS (Figure 6(c)). Our algorithm relies less on repeatedly random sampling the scene since most triangles of the final PVS are found during the exploration phase.

During the same time frame, the exploration phase of GPU-GVS contributes 60% of the triangles to the PVS, while the exploration phase of GVS++ finds 84% of the triangles. For fixed ray budgets such as 1×10^{10} or 0.5×10^{9}, similar ratios can be observed. Discovering most visible triangles during the exploration phase is favorable to avoid inefficient repeated random sampling of the scene. Furthermore, GVS++ offers better parallelizability than GVS, tracing 57% more rays than GPU-GVS during the same time (Figure 6(c)).

GVS++ also scales down well, making it suitable to generate from-region visibility previews in real-time applications. GVS++ achieves an average and maximum pixel error of 190 and 1530 with a very limited time budget of 30ms for the representative view cell in Figure 6. In comparison, GPU-GVS generates a PVS with 1130 and 6650 average and maximum pixel error with the same time budget. With a larger time budget of 100ms, GVS++ reduces the average and maximum pixel error to 18 and 200 (760 and 5308 for GPU-GVS). Similarly, with a limited ray budget of 3×10^{8}

Table 1. Statistics of different from-region visibility algorithms. The error measurements and the PVS size are averaged over ten view cells per scene. Errors are measured on 1000×1000 pixel renderings. For each scene, each algorithm had the same time budget. GVS++ and GVS used the same parameters for all view cells and scenes: $\Delta_{\mathrm{ABS}} = 0.001$, and $n_{\mathrm{rand}} = 10\,000\,000$. Furthermore, $n_{\mathrm{ABSEdge}} = n_{\mathrm{RSEdge}} = n_{\mathrm{RSArea}} = 20$ was set for GVS++. Also, GVS recursively subdivided an edge of a border polygon up to three times. The column showing the calculation time shows times for GVS, GVS++, and RAND without the time to generate Halton sequences. Calculation times for RASTER solely show render times of the hemicubes. GVS++ outperforms the other algorithms, giving a low average and maximum pixel error across all scenes.

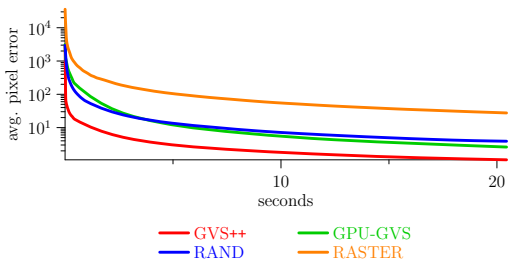
Scene	Algorithm	Avg. Error	Max. Error	Avg. Num. Err. Regions	Max. Error Region Size	Calc. Time	PVS Size
CANYON	**GVS++**	**0.09**	**14**	**0.08**	**8**	**100ms**	**8.60%**
	GPU-GVS	29.15	507	11.08	64	100ms	8.39%
	RAND	234.79	5143	87.26	910	100ms	7.84%
	RASTER	2219.51	29052	249.24	1192	100ms	7.27%
PPLANT	**GVS++**	**2.85**	**77**	**2.37**	**55**	**1000ms**	**1.64%**
	GPU-GVS	21.98	296	13.58	100	1000ms	1.51%
	RAND	34.49	879	27.72	95	1000ms	0.93%
	RASTER	442.28	5110	243.43	3681	1000ms	0.54%
GERMANY	**GVS++**	**0.50**	**16**	**0.45**	**9**	**500ms**	**4.35%**
	GPU-GVS	2.87	150	2.51	23	500ms	4.03%
	RAND	19.56	667	15.11	146	500ms	2.84%
	RASTER	84.74	3856	44.08	714	500ms	2.24%
BISTRO	**GVS++**	**5.39**	**84**	**4.89**	**27**	**2000ms**	**6.82%**
	GPU-GVS	36.61	643	21.54	53	2000ms	6.94%
	RAND	84.91	1924	67.86	169	2000ms	5.23%
	RASTER	41.32	1803	34.33	652	2000ms	5.56%

rays, GVS++ computes a PVS with 15 and 190 (128ms) and GPU-GVS 210 and 2310 (414ms) average and maximum pixel errors.

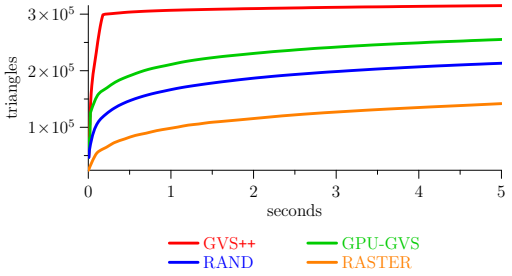
Our approach also outperforms the rasterization-based technique RASTER across all scenes. It should be noted that the times in Table 1 reported for RASTER only show the render times of the hemicubes to avoid distorting the result. The calculation time of RASTER inherently depends on the resolution of the framebuffer that is used and requires a dense placement of hemicubes on the view cell to avoid missing subpixel triangles. The latter is especially a problem in large scenes, where far away triangles are likely to cover less than a pixel.

Comparing GVS++ to the brute-force ray-tracing approach RAND shows the effectiveness of our intelligent sampling schemes, since RAND is based on the same implementation as GVS++ but does not use any intelligent sample placement. The time complexity of both algorithms as well as the convergence of the PVS they produce mostly depends on the complexity of the scene itself. Therefore, scenes with a high primitive count and highly occluded regions will lead to longer computation times than simpler scenes.

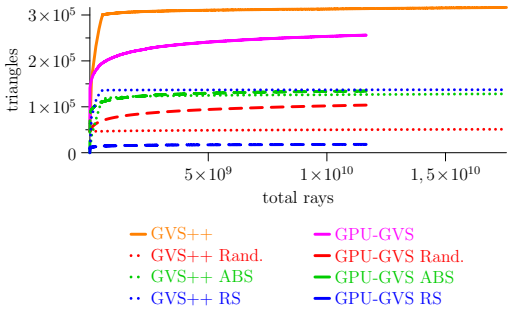
The performance uplift through hardware-accelerated ray tracing can be observed when comparing our GPU implementation of GVS *GPU-GVS* to the original CPU implementation *CPU-GVS* that uses MLRTA [Reshetov et al. 2005] for ray tracing. GPU-GVS uses hardware-accelerated ray tracing



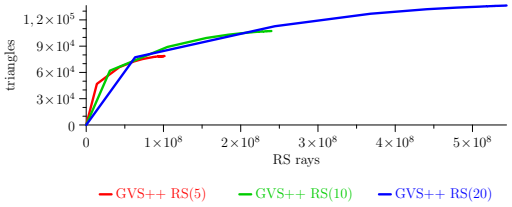
(a) Average pixel error over time on the PPLANT scene.



(b) Zoom-in of the PVS size over time for the first five seconds of each algorithm on the PPLANT scene.



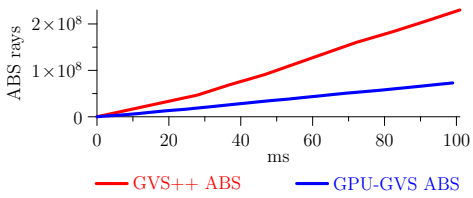
(c) Zoom-in of the first five seconds of the visibility sampling of GPU-GVS and GVS++ on the PPLANT scene. The solid lines show the number of triangles found by GPU-GVS and GVS++ over the total number of traced rays. The dashed (GVS++) and dotted (GPU-GVS) lines show to which sampling strategy (random, reverse sampling, ABS) these triangles can be attributed. Overall, GVS++ finds more triangles per ray and the improved ABS algorithm offers better parallelizability, allowing GVS++ to trace 57% more rays than GPU-GVS during the same time frame (evident by the longer graph of GVS++).



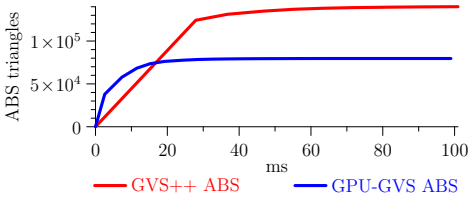
(d) Zoom-in of the first five seconds of GVS++ on the PPLANT scene using three different RS parameter configurations (GVS++ RS($n_{\text{RSEdge}} = n_{\text{RSArea}}$)). The number of triangles found per RS ray largely overlap. With RS(20), GVS++ found 6.4% more triangles than with RS(5) within the same time frame.

Fig. 6. Comparison of GVS++, GPU-GVS, RAND, and RASTER on the PPLANT scene. A representative view cell is used. The same parameters as in Table 1 are used.

via the Vulkan API. Figure 8 shows a significant speedup of the GVS algorithm when hardware-accelerated ray tracing is used (GPU-GVS). When comparing a single iteration of GVS, i.e., random sampling followed by an exploration phase, GPU-GVS is 63 times faster than CPU-GVS. Also, GVS++ is over four orders of magnitude faster than CPU-GVS to compute a PVS on the CANYON scene with comparable pixel errors. It is important to note that a direct runtime comparison is difficult since the CPU-GVS implementation is not watertight and therefore also finds triangles that are not visible. In contrast, our GPU-GVS implementation uses watertight RTX ray tracing. We found that in the example of the CANYON scene, the PVS of CPU-GVS after the first iteration is already 4-6% larger than the PVS of GPU-GVS after 1000 seconds (i.e., practically converged).



(a) The number of traced ABS rays are plotted over the measured ABS execution time.



(b) The number of intersected triangles plotted over the measured ABS execution time.

Fig. 7. GVS++ ABS traces more rays and therefore finds more triangles within the same time frame compared to GVS ABS. This is due to the better parallelizability. Both algorithms were executed with deactivated RS and use a similar number of samples per triangle.

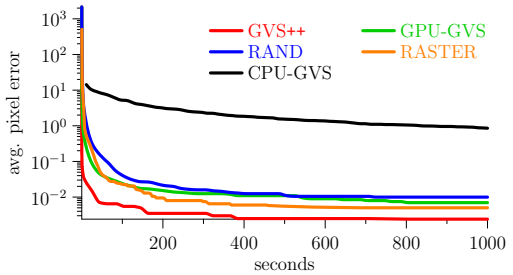


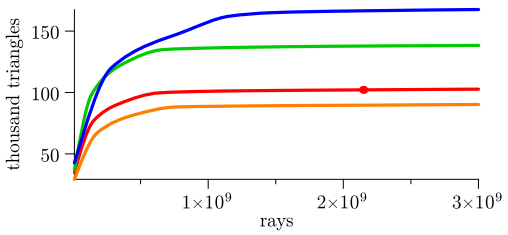
Fig. 8. Comparison of GVS++, GPU-GVS, CPU-GVS, RAND, and RASTER on the CANYON scene, showing average pixel error over time. A representative view cell is used. GVS++, GPU-GVS, and CPU-GVS use the same parameters as in Table 1.

6.2 Asymptotic Behavior

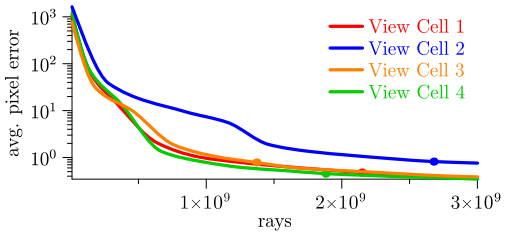
In the following, the asymptotic behavior of the GVS++ algorithm is analyzed in terms of PVS size and pixel error. The same parameters for GVS++ were used for all view cells across all scenes: $n_{rand} = 10\,000\,000$, $n_{ABSEdge} = 40$, $n_{RSEdge} = 40$, $n_{RSArea} = 40$, $\Delta_{ABS} = 0.001$ and $T = 10$.

The PVS size and average pixel error for PPLANT are shown in Figure 9. It can be seen that the PVS shows very similar logarithmic growth for all view cells. The termination threshold $T = 10$ is met in a rather converged state for all view cells. Also, it can be seen that the average pixel error correlates to the PVS size.

Examining the behavior of the algorithm in more detail (Figure 10), it can be seen that most triangles are found early on—that is, during the initial random sampling and the subsequent exploration phase. 99.5% of the triangles of the resulting PVS are found during the initial random sampling and the subsequent exploration phase. After that, the PVS is already highly converged for most scenes, and could thus be used in many practical applications. Examining Figure 10, it can be seen that after the first execution of the exploration phase, the number of triangles found by the ABS and RS shader drop sharply, which also shows the high convergence of the PVS. After the first exploration phase, most (60.2%) of the new triangles are found by random sampling, followed by ABS (35.6%) and RS (4.2%). This indicates that mostly smaller, disconnected regions of triangles or regions that are only visible from a fraction of the view cell are missed during the first random sampling and exploration phase and are found afterward by repeated random sampling.



(a) Average PVS size over time on the PPLANT scene over four view cells.



(b) Average pixel error over time on the PPLANT scene over four view cells.

Fig. 9. PVS size (a), and average pixel error (b) over the number of traced rays for four view cells of PPLANT. Each line represents a view cell. A dot represents the used termination criterion of 10.

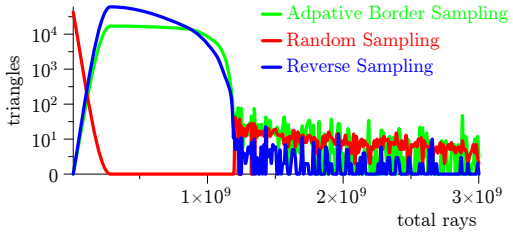


Fig. 10. Detailed plot of the triangles found in one view cell (PPLANT scene, blue in Figure 9) by random sampling (red), adaptive border sampling (blue), and reverse sampling (green) over the number of total traced rays.

7 LIMITATIONS AND FUTURE WORK

One of the main limitations of the proposed algorithm stems from the use of random sampling to find new seed points. The initial random sampling finds a good set of seed points. Once the first execution of the exploration phase does not find any more triangles, the PVS is typically already highly converged, and the algorithm proceeds with randomly sampling the scene. In such a highly converged stage, further triangles are mostly found through random sampling. Such triangles often are highly occluded and are only visible from a fraction of the view cell. Therefore, finding such regions may require a significant number of rays to be traced. This problem could be alleviated by employing an importance sampling approach similar to Ho et al. [Ho et al. 2012]. In the case of GVS++, initially, random sampling could still be used. In later, highly converged stages, a reliability function on the view cell boundary based on previously traced samples could be calculated. This function could then encode locations on the view cell from which highly occluded regions can be seen. The scene could then be sampled from these locations to improve the likelihood of finding missed triangles in later stages.

Another area of improvement is the adaptive border sampling algorithm. Currently, the edges of a triangle's border polygon are sampled using a fixed application-defined number of samples. This could be suboptimal, especially in scenes that consist mostly of very large triangles. To avoid the need for fine-tuning GVS++ for such scenes, the adaptive border sampling algorithm could be changed to sample border polygon edges every x meters. This would alleviate the problem of possible large distances between samples in some cases.

Similar to the technique by Bittner at al. [Bittner et al. 2009], our technique could be extended to compute the PVS of multiple view cells at once. Whenever a ray intersects a triangle, the triangle would then be inserted into the PVS of each view cell that is intersected by the ray.

8 CONCLUSION

We have presented an aggressive from-region visibility algorithm to compute the potentially visible set from a view cell in a general 3D scene. GVS++ calculates a more accurate solution in a shorter timespan compared to brute-force random sampling, GVS, and a comparable rasterization-based visibility sampling technique. The focus was on improving the intelligent sampling strategies to miss less triangles in edge cases. Our algorithms take the highly parallel nature and new features of modern GPUs, such as hardware-accelerated ray tracing, into account and allow for an efficient parallelization of the sampling procedure. The adaptive border sampling algorithm was improved to miss fewer triangles by sampling the neighborhood more thoroughly, and the reverse sampling algorithm was replaced by a different strategy that increases the likelihood of sampling discontinuities and unexplored regions.

Various scenes were used to evaluate the proposed algorithm. Results show that GVS++ converges faster than comparable algorithms. This entails that the algorithm needs less time to compute a potentially visible set with lower errors than comparable algorithms. This can be attributed to our new sampling algorithms that promote efficient parallel execution on the GPU. The improved intelligent sampling strategies inherently provide more flexibility than GVS, allowing the algorithm to be fine-tuned for difficult scenes.

A publicly available implementation of our algorithm is available. Our implementation is based on the Vulkan graphics API and uses hardware-accelerated ray tracing.

REFERENCES

John M Airey, John H Rohlf, and Frederick P Brooks Jr. 1990. Towards image realism with interactive update rates in complex virtual building environments. *ACM SIGGRAPH computer graphics* 24, 2 (1990), 41–50.

Chandra R Bhat. 2003. Simulation estimation of mixed discrete choice models using randomized and scrambled Halton sequences. *Transportation Research Part B: Methodological* 37, 9 (2003), 837–855.

Jirı Bittner. 2002. *Hierarchical techniques for visibility computations.* Ph.D. Dissertation. Czech Technical University.

Jiří Bittner, Oliver Mattausch, Peter Wonka, Vlastimil Havran, and Michael Wimmer. 2009. Adaptive global visibility sampling. *ACM Transactions on Graphics (TOG)* 28, 3 (2009), 1–10.

Jiří Bittner and Peter Wonka. 2003. Visibility in computer graphics. *Environment and Planning B: Planning and Design* 30, 5 (2003), 729–755.

Jirí Bittner, Peter Wonka, and Michael Wimmer. 2005. Fast exact from-region visibility in urban scenes. In *Rendering Techniques*. 223–230.

Daniel Cohen-Or, Yiorgos L Chrysanthou, Claudio T. Silva, and Frédo Durand. 2003. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics* 9, 3 (2003), 412–431.

Frédo Durand. 1999. *3D Visibility: analytical study and applications.* Ph.D. Dissertation.

Frédo Durand, George Drettakis, Joëlle Thollot, and Claude Puech. 2000. Conservative visibility preprocessing using extended projections. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 239–248.

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

Ned Greene, Michael Kass, and Gavin Miller. 1993. Hierarchical Z-buffer visibility. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. 231–238.

Jozef Hladky, Hans-Peter Seidel, and Markus Steinberger. 2019. The camera offset space: real-time potentially visible set computations for streaming rendering. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–14.

Tan-Chi Ho, Ying-I Chiu, Jung-Hong Chuang, and Wen-Chieh Lin. 2012. Aggressive region-based visibility computation using importance sampling. In *Proceedings of the 11th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and its Applications in Industry*. 119–126.

Samuli Laine and Tero Karras. 2010. Two Methods for Fast Ray-Cast Ambient Occlusion. In *Computer Graphics Forum*, Vol. 29. Wiley Online Library, 1325–1333.

Tommer Leyvand, Olga Sorkine, and Daniel Cohen-Or. 2003. Ray space factorization for from-region visibility. In *ACM SIGGRAPH 2003 Papers*. 595–604.

Amazon Lumberyard. 2017. *Amazon Lumberyard Bistro, Open Research Content Archive (ORCA).* Retrieved November 25, 2020 from http://developer.nvidia.com/orca/amazon-lumberyard-bistro

Shaun Nirenstein and Edwin Blake. 2004. Hardware accelerated visibility preprocessing using adaptive sampling. (2004).

Shaun Nirenstein, Edwin Blake, and James Gain. 2002. Exact from-region visibility culling. Eurographics.

NVIDIA. 2018. *NVIDIA Turing GPU Architecture.* Retrieved November 25, 2020 from https://www.nvidia.com/en-us/geforce/news/geforce-rtx-20-series-turing-architecture-whitepaper/

NVIDIA. 2020. *NVIDIA Ampere GPU Architecture.* Retrieved November 25, 2020 from https://www.nvidia.com/en-us/geforce/news/rtx-30-series-ampere-architecture-whitepaper-download/

Alexander Reshetov, Alexei Soupikov, and Jim Hurley. 2005. Multi-level ray tracing algorithm. *ACM Transactions on Graphics (TOG)* 24, 3 (2005), 1176–1185.

Gernot Schaufler, Julie Dorsey, Xavier Decoret, and François X Sillion. 2000. Conservative volumetric visibility with occluder fusion. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 229–238.

Seth J Teller and Carlo H Séquin. 1991. Visibility preprocessing for interactive walkthroughs. *ACM SIGGRAPH Computer Graphics* 25, 4 (1991), 61–70.

UNC. 2001. *Power Plant Model.* Retrieved November 25, 2020 from http://gamma.cs.unc.edu/POWERPLANT/

Johannes Unterguggenberger, Bernhard Kerbl, Markus Steinberger, Dieter Schmalstieg, and Michael Wimmer. 2020. Fast Multi-View Rendering for Real-Time Applications. (2020).

Peter Wonka, Michael Wimmer, and François X Sillion. 2001. Instant visibility. In *Computer Graphics Forum*, Vol. 20. Wiley Online Library, 411–421.

Peter Wonka, Michael Wimmer, Kaichi Zhou, Stefan Maierhofer, Gerd Hesina, and Alexander Reshetov. 2006. Guided visibility sampling. *ACM Transactions on Graphics (TOG)* 25, 3 (2006), 494–502.