Efficient and Robust From-Point Visibility

Voicu Popescu, *Member, IEEE*, Elisha Sacks, Jian Cui, and Rohan Ashok

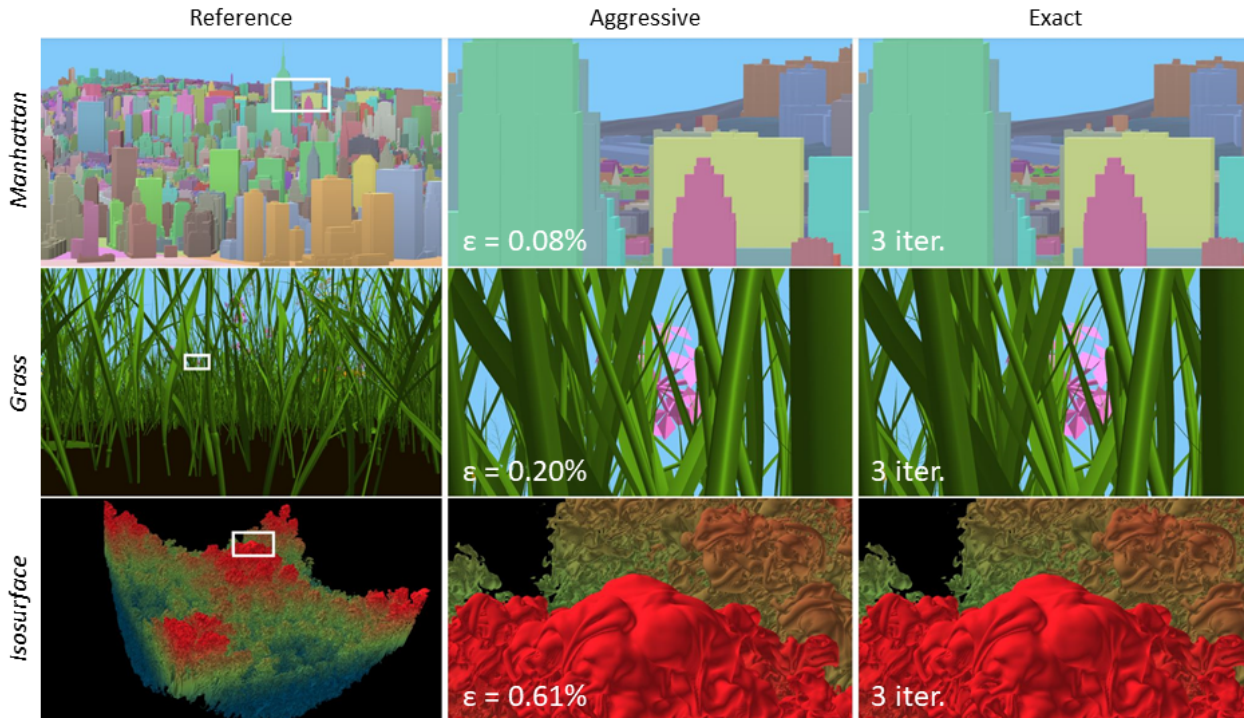

Fig. 1. Results of our from-point visibility algorithms on three datasets with 4, 55, and 500 million triangles (from top to bottom). The left column shows the reference images on which the algorithms were run. The middle and the right columns show frames rendered from the aggressive and exact visible sets. Despite the large zoom factors of 7x, 17x, and 10x, the frames from the aggressive set have only a small percentage ε of incorrect pixels. The frames from the exact set are identical to the ones one would obtain from the entire dataset. Despite the complex occlusion patterns in these examples, the exact algorithm converges in three iterations.

Abstract—This paper presents two from-point visibility algorithms: one aggressive and one exact. The aggressive algorithm efficiently computes a nearly complete visible set, with the *guarantee* of finding *all* triangles of a front surface, no matter how small their image footprint. The exact algorithm starts from the aggressive visible set and finds the remaining visible triangles efficiently and robustly. The algorithms are based on the idea of generalizing the set of sampling locations defined by the pixels of an image. Starting from a conventional image with one sampling location at each pixel center, the aggressive algorithm adds sampling locations to make sure that a triangle is sampled at all the pixels it touches. Thereby, the aggressive algorithm finds all triangles that are completely visible at a pixel regardless of geometric level of detail, distance from viewpoint, or view direction. The exact algorithm builds an initial visibility subdivision from the aggressive visible set, which it then uses to find most of the hidden triangles. The triangles whose visibility status is yet to be determined are processed iteratively, with the help of additional sampling locations. Since the initial visible set is almost complete, and since each additional sampling location finds a new visible triangle, the algorithm converges in a few iterations.

Index Terms—from-point visibility, aggressive visibility, exact visibility, triangle visibility, particle visibility, image generalization.

1 Introduction

Visibility is a fundamental problem in visualization that remains open despite decades of research. Given a 3D dataset and a set of viewpoints, the visibility problem asks which of the geometric primitives in the dataset are visible from at least one of the viewpoints. Visibility algorithms usually work with triangles, which are building blocks of more complex geometric primitives. A triangle t is visible from a viewpoint v if there is a point p in t to which there is line of sight from v, i.e., the line segment vp does not intersect any other triangle in the dataset. As the number of triangles in the visible set is typically a small fraction of the total number of triangles in the dataset, visibility is a powerful tool for reducing dataset complexity.

One approach for solving visibility is to probe for visible triangles along rays that originate from the given set of viewpoints. Such sample-based visibility algorithms, called *aggressive* algorithms, find *some* but not *all* visible triangles. A sample-based visibility algorithm cannot verify that a triangle is hidden, as that would require an infinite number of rays to verify that there is no line of sight to any of the triangle points. Therefore, a sample-based visibility algorithm cannot know whether the set it finds is complete. A potential advantage of sample-based algorithms is efficiency, as they can find all the visible triangles using

- *Voicu Popescu is with Purdue University. E-mail: popescu@purdue.edu.*
- *Elisha Sacks is with Purdue University. E-mail: eps@purdue.edu.*
- *Jian Cui is with Alphabet Inc. E-mail: cuijian2005686@gmail.com.*
- *Rohan Ashok is with Purdue University. E-mail: ashokr@purdue.edu.*

one ray per triangle, but deciding which rays to use is challenging. Aggressive visibility algorithms strive to minimize the number of rays while maximizing the number of visible triangles found.

Another approach to visibility is to analyze the continuous space of rays originating from the given viewpoints, subdividing it into regions where a single triangle is visible. Such continuous visibility algorithms can provide the *exact* visible set, containing *all and only* visible triangles. Continuous visibility algorithms are computationally expensive, as the dataset complexity is compounded by the dimension of the space of rays. They are also prone to robustness problems, meaning that even tiny rounding errors can cause large errors in the visibility set that they output.

This paper addresses from-point visibility: the problem of finding all triangles that are visible from a *single* viewpoint. From-point visibility is a foundational problem with important applications in computer graphics and visualization. One application is the computation of *accurate* visualizations of complex datasets. For such datasets, the color at a pixel depends on the contribution of the many triangles that partially cover the pixel. Conventional antialiasing schemes, such as 4x4 supersampling, are only palliative as they haphazardly select a subset of the triangles covering the pixel. From-point visibility can find all triangles visible at a pixel, which results in a correct output color computed by blending the colors of all triangles covering the pixel, with weights commensurate to their pixel areas. A second application is to use from-point visibility as a building block for solving higher-order visibility problems, such as from-region visibility. The view region is sampled with viewpoints, from-point visibility is solved for each viewpoint, and the from-point visible sets are unioned to obtain an approximation of the from-region visible set. A third application is to use exact from-point visibility as an instrument for measuring the performance of approximate visibility algorithms. Knowing the exact visible set enables one to find the false positives and false negatives in an approximate visible set. Furthermore, the exact set allows one to measure the area of an output image where the approximate visible set provides the correct visible triangle, which is a more discerning quality metric than the number of visible triangles found.

In addition to these three applications, which we demonstrate in our paper, from-point visibility can also reduce bandwidth and latency in remote visualization. The client sends the desired viewpoint to the server; the server computes the visible set and sends it to the client; the visible set is a fraction of the entire dataset, which saves bandwidth; the client renders the visible set locally, with low latency, to support user view rotations and zoom changes at interactive rates; the completeness of the from-point visible set supports zooming in with good output image quality, even for large magnification factors. Another application of from-point visibility is accurate hard shadow computation. Finding all triangles visible from a point light source avoids the light leaks that plague methods that rely on an approximate from light-point visibility solution, such as shadow mapping. From-point visibility is also useful for the high-fidelity simulation of sound propagation.

From-point visibility does not have a prior algorithm that is efficient and robust. A promising aggressive approach is to render the dataset from the given viewpoint into an image that records one visible triangle per pixel. The advantage is efficiency, as visibility is probed along a large number of rays at a small computational cost in feed forward fashion, by projection followed by rasterization. The challenge is that, in the case of complex datasets, many visible triangles have a small image footprint and hence are not found by any of the centers of the image pixels. Improving the aggressive visible set by uniformly increasing the resolution of the image is inefficient. Continuous from-point visibility algorithms have to build a subdivision of the 2D space of rays originating at the viewpoint, which is computationally expensive if they have to process all the triangles of a complex dataset.

We present two from-point visibility algorithms, one aggressive and one exact. The algorithms are based on the idea of generalizing the set of sampling locations defined by the pixels of an image. The aggressive algorithm starts from a conventional image with sampling locations at pixel centers, and adds sampling locations to make sure that each triangle is sampled at all the pixels that it touches. If a triangle t is completely visible at a pixel p, t is guaranteed to be found by its sampling location in p. Hence, the algorithm finds all triangles of a front surface, regardless of geometric level of detail, distance from viewpoint, or view direction.

The exact algorithm starts from the aggressive visible set and finds the remaining visible triangles efficiently and robustly. It builds an initial visibility subdivision from the aggressive visible set, which it then uses to find most of the hidden triangles. The triangles whose visibility status is yet to be determined are processed iteratively, by generating additional sampling locations where these triangles are not hidden by the current visibility subdivision. The additional sampling locations find new visible triangles, which are added to the visibility subdivision. The algorithm converges quickly because the initial visible set is almost complete and each additional sampling location finds a new visible triangle. Only visible triangles are added to the visibility subdivision, which makes our exact algorithm output sensitive.

The robustness requirement is that the visibility subdivision be correct. Correctness is challenging because the subdivision is computed by evaluating millions of predicates: numerical expressions on whose sign the algorithm execution branches. A single incorrect predicate, due to floating point rounding error, can corrupt the entire subdivision because of global dependencies. We achieve robustness using the Exact Geometric Computation (EGC) [43] strategy of evaluating predicates correctly despite numerical error.

Figure 1 illustrates our from-point visibility algorithms. The aggressive visible set is nearly complete, as confirmed by the small errors obtained in frames with a large zoom factor (middle column). The few incorrect pixels are at surface boundaries, so the frames are comparable to the truth frames obtained from the exact set. Another measure of the completeness of the aggressive set is the percentage of the reference image where visibility is solved correctly, which is 99.98%, 99.82%, and 99.66% for the three datasets. In all our experiments, the exact visibility algorithm extended the aggressive set to the exact set in at most four iterations. We also refer the reader to the accompanying video.

In summary, our paper makes three major contributions:

- the first aggressive visibility algorithm with a quality guarantee;
- the first exact from-point visibility algorithm with an output-sensitive visibility subdivision construction;
- the first robust implementation of exact from-point visibility.

2 Prior Work

Visibility algorithms are classified based on the visible sets that they compute. Conservative algorithms overestimate visibility, so no visible triangle is omitted. The benefit is an accurate image, but the number of hidden triangles in the output can be substantial [10, 14]. Aggressive algorithms underestimate the set of visible triangles, which leads to image errors. The goal of aggressive visibility research is to reduce and control the error [35, 42]. Exact algorithms find only and all visible triangles, which avoids the cost of rendering unnecessary triangles as well as any image error.

Aggressive Visibility. We distinguish between probing visibility by casting individual rays and by rendering entire images. Algorithms in the first category use heuristics to shoot rays that are likely to find visible triangles, and subsequent sampling is guided by what the initial rays find [4, 29, 42]. The advantage is the flexibility to cast precisely the rays deemed necessary, which limits sampling redundancy, and allows supporting effects such as lens distortion, foveation, or depth of field [25]. However, it is difficult to place error bounds on the results, and, as we show in the results section, our aggressive visible set has a quality advantage over guided visibility sampling [4, 29].

Algorithms in the second category leverage the fact that the amortized cost of rays in an image is lower than that of individual rays. Our algorithms fall in this second category. An image only captures samples visible from its viewpoint. One option is to use images from additional viewpoints [31], which are highly redundant, or to eliminate redundancy as a pre-process [30, 39]. The challenge of these approaches

is to decide which images are needed for a sufficient sampling of the visibility parameter space. The usual strategy is to sample uniformly as densely as possible, and thus the visibility error is not bounded. Multiperspective images capture in a single shot more than what is visible from a single viewpoint through innovation at the camera model level [11, 44], but there is no visible set quality guarantee.

Specialized visibility algorithms have been developed for many contexts. The algorithms are typically aggressive, focusing on finding the visible triangles of highest relevance in the particular context. The semi-analytical visibility algorithm [21], developed for motion blur, samples the image with lines as opposed to points, an idea borrowed from temporal antialiasing [27]. Visibility is analyzed continuously over time for each line sample. The algorithm is aggressive because the analysis is restricted to a uniform grid of image lines. Line samples are a brute force approach for improving uniform point sampling. The line parameter adds an expensive second dimension to the 1D motion blur visibility problem. The uniform line sample pattern is heuristic, so even after solving the higher-dimension visibility problem, there is still no guarantee for the quality of the solution. We propose deterministic point sampling that guarantees a quality visible set without increasing the visibility problem dimensionality.

Recent work analyzes visibility in the camera offset space defined by viewpoint translations [24]; the visible set is exact at pixel centers under camera translations, which means that visible triangles that are not visible at a pixel center are missed; in order to capture additional visible triangles, such as those visible under camera rotations, additional sampling locations are added heuristically. Our aggressive algorithm guarantees quality by generalizing the set of sampling locations on the image plane, beyond the predefined set of pixel centers.

Exact Visibility. Finding all and only visible triangles brings the greatest possible reduction of the dataset without the errors implied by false negatives. Theoretical computational geometry has explored the high-dimensional space of the visibility rays for from-region visibility. In from-rectangle visibility, one strategy is to decompose the 4D non-Euclidean space of rays according to visibility criteria. A complete model with $O(n^4)$ complexity and output sensitive construction is yet to be practical [15]. Another strategy is to compute visibility between pairs of polygons [9, 22, 34], using constructive solid geometry in the 5D ambient space of the Plucker coordinate representation of lines. The algorithms have high computational complexity and have not been demonstrated on large datasets.

Early work on from-point visibility focuses on antialiasing. The solution was to compute a visibility subdivision for each pixel, defined by the triangle fragments visible at each pixel [6, 7, 41]. The solution is inefficient because fragments of hidden triangles are added and then removed from the visibility subdivision. We compute the visibility subdivision exclusively from visible triangles, which amounts to an efficient way of computing an accurate image that takes into account the contributions of all visible triangles, no matter how small their footprint. Pixel-free from-point visibility algorithms are also inefficient because they compute occluded intersections [20]; typical running times are $O((n+k)\log n)$ or $O(n\log n+k+t)$ for n triangles with k edge intersections and t triangle intersections on the image plane. Output sensitive algorithms are restricted to special input [28, 40]. From-point visibility was implemented on the GPU [2], but with a running time quadratic in the number of triangles.

Beam tracing [23] analyzes from-point visibility continuously by using conical or frustum-like beams. The unsampled gaps between rays are avoided, but beam-triangle intersection is costly. Beam-tracing has also been used for shadow [1, 36] and sound [8, 37] rendering, using acceleration schemes based on adaptive beam splitting. Beam tracing has been recently revisited for its ability to integrate visibility, which supports the differentiable rendering used for example in inverse graphics [46]; the complexity of the polygonal regions of the visibility subdivision of the beam is capped at four vertices, but that increases the number of regions; most importantly, the visibility subdivisions are computed from all triangles, unnecessarily adding and then removing the contributions of triangles that are only visible with respect to the triangles considered so far, and that ultimately turn out to be hidden. We bypass the need for beams, replacing them with the smallest number of rays needed to capture the visible triangles over a solid angle; we do not trace rays, but rather evaluate visibility along them by projection onto their sampling locations; finally, we compute the visibility subdivision exclusively from visible triangles, and with numerical robustness.

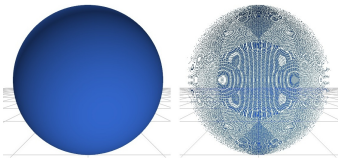

Fig. 2. An image of a finely tessellated sphere (left) and a frame rendered from the visible triangles found by the image, using the same viewpoint with a slightly different view direction (right). The frame shows that the visible set is far from complete.

Conservative Visibility algorithms are exact algorithms that run on a visibility problem that was conservatively simplified, e.g. through extended projections [16], or occluder erosion [13]. Our aggressive algorithm produces a visible set that is almost complete, so adding the triangles that are not hidden by the aggressive set yields a good conservative visible set. Per-frame occlusion culling improves rendering performance by batch discarding triangles that are hidden in the current output frame [5]. Triangles are grouped inside containers with simple geometry, the containers are rendered on a partial z-buffer of the output frame obtained from known big blockers, and the triangles of hidden containers are discarded. Occlusion culling methods can also be aggressive by fusing blockers heuristically [45].

We discuss the hierarchical visibility culling algorithm [3] since it addresses our problem of from point visibility. The algorithm first finds a subset of the hidden triangles using a hierarchical spatial subdivision. A subdivision region is ruled as hidden, together with its descendants, by finding large convex polygonal occluders, and by checking whether the region is in the occlusion shadow of any such occluder. Then the triangles in the remaining subdivision regions, i.e., those that have not been ruled as hidden, are rendered with z-buffering to compute the output image. The algorithm has the merit of removing some of the hidden triangles efficiently, in group, without considering them individually. However, for this, it relies on the assumption that the dataset has large polygonal blockers, which have to be found in a preprocessing step. The algorithm is conservative, in the sense that it finds all but not only visible triangles. The algorithm does not compute the exact set, it merely z-buffers the conservative set to obtain the output image. Our algorithms do not group triangles, but they find a quality aggressive visible set with most visible and no hidden triangles, or they find the exact set.

Irregular Framebuffers. We advocate abandoning the uniform sampling of conventional images in favor of adding sampling locations deterministically to guarantee that all visible triangles are found. The benefits of irregular framebuffers have been noted before in contexts that include: pixel-accurate shadow mapping [26], where the shadow map estimates light visibility precisely at the point samples captured by the output image; point-based rendering [38], where projected reference image samples are not clamped to the output image pixel grid but rather located precisely within the output image pixel using a pair of offsets; and focus plus context visualization where focus regions are sampled at a higher rate [19].

3 Guaranteed-Quality Aggressive Visibility

An image is an appealing tool for computing visibility. Rendering an image is equivalent to probing dataset visibility efficiently with millions of rays, one for each pixel. However, the visible set found by an image can be incomplete because triangles can have small footprints due to high dataset complexity, to large distances to the eye, or to grazing viewing angles. Figure 2 shows that a conventional image misses most visible triangles of the front surface of a finely tessellated sphere. A slightly different view direction reveals the many gaps in the aggressive

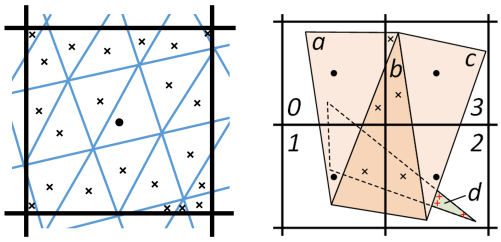

Fig. 3. **(Left)** Finding all front-surface triangles. The triangles (blue lines) projecting at a pixel (black lines) are shown in wireframe. A sampling location at the pixel center (dot) captures only one of the many visible triangles. Our aggressive algorithm adds sampling locations (diagonal crosses) to sample all triangle fragments at the pixel. **(Right)** Aggressive visibility and extension to exact visibility. The aggressive algorithm processes triangles a–d in that order, finding the visible triangles a, b, and c. Triangle d is missed because it does not have a completely visible fragment and because its only partially visible fragment (in pixel 2) is sampled where d is hidden by b. The exact algorithm finds that the fragment of d in pixel 2 is not hidden by a–c and adds sampling locations (red straight crosses) in the visible part of the fragment.

visible set. A single sampling location per pixel, e.g., at the center of the pixel, captures only one of the many visible triangles whose projections overlap with the pixel (Figure 3, left). Alleviating the problem by increasing image resolution is inefficient, as some triangles will be sampled multiple times, and only palliative, as no resolution can guarantee that all the triangles are found.

3.1 Approach

We improve the quality of the visible set found by an image by augmenting the image with additional sampling locations where visibility is probed. The goal is to minimize the number of additional sampling locations and to maximize the number of triangles found. We construct sampling locations with a greedy approach that guarantees that each *triangle fragment* is sampled. A triangle fragment is the intersection of a triangle with a pixel. Sampling location construction is illustrated in Figure 3, left). The dataset is then rendered over the sampling locations to obtain the visible set.

Our approach *guarantees* finding all completely visible triangles, including all front surfaces, because all their fragments are completely visible and contain sampling locations. Our approach is *efficient*. Whereas the conventional approach of uniformly increasing the image resolution adds sampling locations blindly, our approach adds a sampling location only to sample a fragment that is not sampled by any of the current sampling locations. Our approach increases the image resolution locally, as needed to sample the visibility of dataset regions with higher complexity. It finds all the triangles of the front face of the sphere in Figure 2 at the cost of one sampling location per fragment.

3.2 Algorithm

Our aggressive from-point visibility algorithm (Algorithm 1) takes as input the dataset D, the viewpoint o from which to compute visibility, and a uniform 2D grid of pixels G that corresponds to the reference image where visibility is computed.

(Line 1) Each pixel of G stores a set of sampling locations S, which initially contains only pixel centers. In addition to its 2D coordinates, a sampling location stores the depth and index of the closest triangle sampled so far.

The algorithm takes three passes over the dataset.

(Line 2) The first pass is a conventional rendering over the pixel grid with one sampling location at each pixel center. In Figure 3 (right), this pass finds triangle a in pixels 0 and 1, and triangle c in pixels 2 and 3. The motivation for the first pass is efficiency. It finds nearby triangles with a large footprint, which hide many other triangles. The nearby triangles are used in the second pass to avoid creating sampling locations that cannot find new visible triangles.

Algorithm 1 Aggressive from-point visibility

Input: dataset triangles D, viewpoint o, grid of pixels G
Output: aggressive set of visible triangles V_0

1: **for all** pixels $p \in G$ **do** $p.S = \{Center(p)\}$
2: **Render** D from o over G // PASS 1
3: **for all** triangles $t \in D$ **do** // PASS 2
4: $t' = Project(t, o, G)$
5: **for all** pixels p that intersect t' **do**
6: **if** $\nexists\ s \in p.S$ such that $s \in t'$ **then**
7: fragment $f = t' \cap p$
8: sampling location $s = Centroid(f)$
9: N = triangles visible at centers of p and its neighbors
10: **if** t is closer to o than every $n \in N$ at s **then**
11: $p.S = p.S \cup \{s\}$
12: **Render** D from o over G // PASS 3
13: **for all** sampling locations s of all pixels in G **do**
14: $V_0 = V_0 \cup \{s.triangle\}$
15: **return** V_0

(Lines 3–11) The second pass adds sampling locations to ensure that all triangle fragments are sampled. For each pixel p intersected by the projection t' of a triangle t (line 5), the algorithm checks whether p already has a sampling location s that samples t (line 6). Since s must be inside p, this is equivalent to checking whether p has a sampling location inside the fragment f of t at p. If not, a sampling location s is created at the centroid of f (lines 7–8). s is added to the set of sampling locations of p unless t is hidden at s by a triangle n that is visible at the center of p or of one of its four horizontal and vertical neighbors (lines 9–11). This test performs occlusion culling relative to the visible triangles found in the first pass.

In Figure 3 (right), a generates one sampling location in pixel 3 (cross) and b generates one in each of the four pixels. Triangle d generates no sampling locations, as the fragments of d in pixels 1 and 2 already contain sampling locations that were added for b. The fragment of d in pixel 0 does not contain a sampling location because d is hidden at the centroid of its fragment by a, which was found at step 1. A sampling location at the centroid of the fragment of d in pixel 0 would be wasteful as it would only reconfirm that a is visible, with no chance of elucidating the visibility status of d.

(Line 12) The third pass renders the dataset over the sampling locations defined by the second pass. Each triangle is projected onto the pixel grid G. For every pixel that the projection intersects, the triangle is z-buffered over the sampling locations that are inside the projection.

(Lines 13–15) The closest triangles recorded by the sampling locations after the third pass are collected to form the visible set.

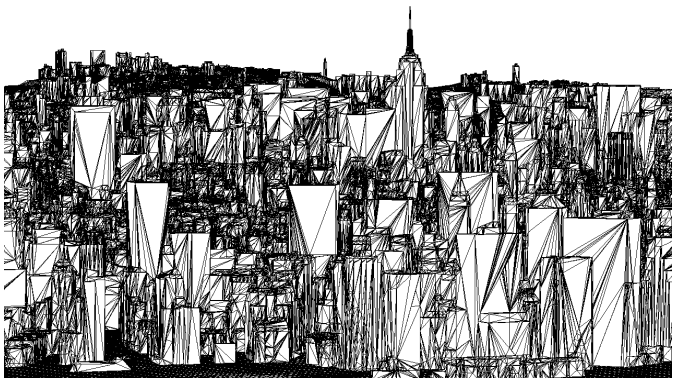

Fig. 4. Visibility subdivision for top-left image in Figure 1.

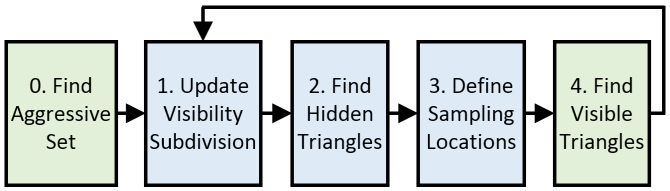

Fig. 5. Iterative from-point visibility approach combining sample-based (green) and continuous (blue) visibility analysis.

3.3 Complexity

The complexity of the aggressive algorithm (Algorithm 1) is dominated by the point-in-triangle tests in lines 6 and 12. Our implementation tests every site in a pixel against every triangle that intersects the pixel. The cost for a pixel with t triangles and $s \leq t$ sites is $O(ts)$. We can reduce the number of tests by storing the sites in a quadtree, retrieving the ones inside the bounding box of the triangle, and testing if they are in the triangle. In our experiments, quadtrees reduce the number of tests by 20% to 80%, but their construction cost outweighs these savings. Maximum-spread *kd*-trees [18] have better asymptotic complexity than quadtrees, but perform similarly on step 12 in our experiments, and are inapplicable to step 6 because they cannot be built incrementally. Range trees with dynamic fractional cascading [32] are best in theory because they have optimal $O(\log s)$ query time, but they use many pointers, which degrades GPU performance. Another option is to use a simplex search to find the sites in a triangle. In our experiments, only 10% to 20% of the sites in the bounding box of a triangle are in the triangle, so the potential savings are large. Unfortunately, the best known simplex algorithms take $O(\sqrt{s})$ time per query or use $O(s^2)$ memory [12, Chapter 16]. Moreover, they are complicated to implement, have large constant factors, and cannot be built incrementally.

4 Efficient and Robust Exact Visibility

The from-point visibility subdivision is the partition of the image plane induced by the projected edges of the visible triangles. The regions of the partition are polygons in which a single triangle is visible. The region boundaries are the visible segments of the projected triangle edges. Figure 4 shows the subdivision of the *Manhattan* dataset shown in Figure 1.

4.1 Approach

We employ a novel *hybrid* approach to exact visibility computation that combines sample-based with continuous visibility analysis (Figure 5).

Algorithm 2 Exact from-point visibility

Input: dataset D, viewpoint o, reference image G, aggressive visible set V_0
Output: exact set of visible triangles V

```
1:  V = V_0, U = D − V_0, VS = ∅
2:  for all triangles t ∈ V_0 do AddTriangle(VS, t, o, G)
3:  while U ≠ ∅ do
4:      for all pixels p ∈ G do p.S = ∅
5:      for all triangles t ∈ U do
6:          if t is hidden by VS then
7:              U = U − {t}
8:          else
9:              AddSamplingLocations(VS, t, o, G)
10:     Render U from o over G
11:     for all sampling locations s of all pixels in G do
12:         t = s.triangle
13:         if t ∉ V then
14:             V = V ∪ {t}, U = U − {t}, AddTriangle(VS, t, o, G)
15: return V
```

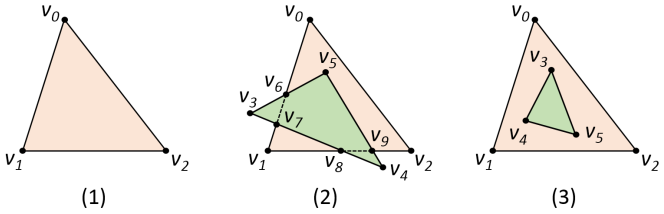

Fig. 6. Adding a triangle to a visibility subdivision: current visibility subdivision (1), addition of an intersecting (2) and of an overlapping (3) triangle.

The aggressive visibility algorithm is run (step 0) then the aggressive visible set is extended iteratively to the exact set (steps 1-4). At each iteration, the visibility subdivision is updated, first with the aggressive visible set and subsequently with the newly found visible triangles (step 1). The other triangles are tested against the visibility subdivision, some are proven to be hidden, and the rest are deemed undecided (2). Additional sampling locations are defined where the undecided triangles might be visible (3). The undecided triangles are rendered over these sampling locations to reveal additional visible triangles (4). The iteration ends when no undecided triangles remain. A sampling location created at step 3 is guaranteed to find a new visible triangle: either the undecided triangle that generated it or a closer triangle. Therefore each iteration reduces the undecided set and the algorithm converges.

4.2 Algorithm

The exact visibility algorithm appears in Algorithm 2. The inputs are those of the aggressive algorithm plus the aggressive visible set.

(Line 1) Initialize the visible set V to the aggressive visible set V_0 and place the other triangles in the undecided set U. The visibility subdivision VS is initialized to empty.

(Line 2) Construct an initial visibility subdivision VS from the aggressive visible set V_0 by adding the triangles one at a time. The construction is illustrated in Figure 6. The initial visibility subdivision has three edges that bound the red triangle (1). When an intersecting (green) triangle is added (2), the visibility subdivision is updated to define two regions where the red triangle is visible, $v_0v_6v_5v_9v_2$, and $v_1v_8v_7$, and one region where the green triangle is visible, $v_3v_7v_8v_4v_9v_5v_6$. If the new triangle is contained in a single region and is in front of that triangle (3), the region acquires a hole and the new triangle creates a new region. In the example, the region where the red triangle is visible is bounded by two edge loops, $v_0v_1v_2$ and $v_3v_5v_4$, and the region where the green triangle is visible has one edge loop, $v_3v_4v_5$. Subdivision construction is described in Section 4.3.

(Lines 3–14) Iterate until there are no more undecided triangles. Each iteration first clears the sets of sampling locations S stored at each pixel p in G (line 4), as a sampling location is not useful beyond the iteration when it was created and found its visible triangle. Then the undecided triangles are processed one at a time (lines 5–9). If an undecided triangle t is hidden by the visibility subdivision VS, t is removed from further consideration (line 7). Otherwise, sampling

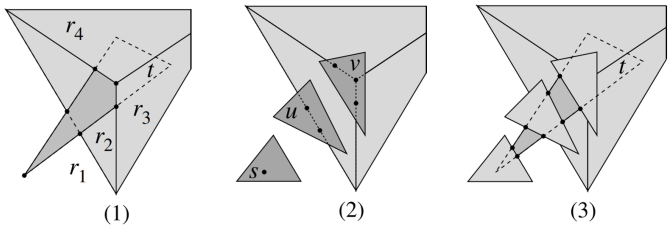

Fig. 7. (1) Six sampling locations (dots) created for undecided triangle t at first iteration, (2) triangles s, u, and v visible at those sampling locations, and (3) eight sampling locations (dots) created for t at second iteration.

locations are created for each region r of VS where t is visible (line 9). The visibility test is described in Section 4.3.

(Lines 9–14) If a triangle t is not hidden by VS, generate sampling locations at the visible parts. The sampling locations are inside t near the endpoints of the edges that t would create if it were added to VS. In Figure 7 (1), t is hidden in regions r_3 and r_4, but not in r_1 and r_2, and six sampling locations are created. The remaining undecided triangles are rendered over the new sampling locations (line 10). The newly found visible triangles are added to the visible set, and are used to update the visibility subdivision (lines 11–14).

Visibility is determined over several iterations. A sampling location q generated for an undecided triangle t might be covered by another undecided triangle that is closer than t at q. If all sampling locations generated for an undecided triangle t are won by other undecided triangles, the visibility of t will not be determined at the current iteration. In Figure 7, the six sampling locations generated for t at the first iteration are won by s, u, and v (2), so t remains undecided. The updated visibility subdivision does not hide t, so another eight sampling locations are generated for t (3).

The algorithm is guaranteed to converge because every iteration creates a sampling location at which an undecided triangle is visible, thereby shrinking the undecided set. *The algorithm is fast* for several reasons. (1) The expensive construction of the visibility subdivision is shielded from the full dataset complexity and only has to run on the visible set. (2) The first iteration finds almost all the visible triangles and rules out almost all the hidden triangles because the aggressive visible set is almost complete. (3) A triangle that fails to be decided at the current iteration is sampled with at least three sampling locations close to the contour of the part of t that is visible in the current visibility subdivision. It is unlikely that t be visible, yet hidden at all of these sampling locations.

4.3 Visibility subdivision construction

The visibility subdivision is encoded using the half-edge subdivision representation [12, Chapter 2]. The edges contain pointers to the triangles that are visible to their left. Edges are added, split, and removed as described in the textbook. A triangle is added to the visibility subdivision with Algorithm 3. The algorithm updates the vertices and edges of the subdivision. The faces are not needed for enumerating the visible set. When they are needed, for example for computing the quality of the aggressive visible set, they are computed by a standard algorithm after the visibility subdivision is complete.

(Line 1) Project the input 3D triangle t to 2D triangle t' with vertices $(v_0v_1v_2)$.

(Line 2) Find the set E of visibility subdivision edges that intersect the edges of t'. In Figure 6, $E = \{v_0v_1, v_1v_2, v_2v_3\}$ in part (2) and E is empty in part (3).

(Line 3–6) If E is empty, add the edges of t' to VS (Line 6) if there is no triangle s in VS that covers t' (Lines 4-5). In Figure 6 (3), the covering triangle is $s = v_0v_1v_2$ and t' is in front of s, so its edges are added to the visibility subdivision.

(Lines 8–10) If E is not empty, compute a set S of edges obtained by splitting the intersecting edges of VS and t' (Lines 9–10). In Figure 6 (2), $S = \{v_0v_6,\ v_6v_7,\ v_7v_1,\ v_1v_8,\ v_8v_9,\ v_9v_2,\ v_6v_5,\ v_5v_9,\ v_3v_7,\ v_7v_8,\ v_8v_4,\ v_4v_9,\ v_9v_5,\ v_5v_6,\ v_6v_3\}$.

(Line 11) Remove the hidden edges from S. These are v_6v_7 and v_8v_9 in Figure 6 (2).

(Line 12) Add S to the visibility subdivision.

For the example in Figure 6 (2), there are three regions: $v_0v_6v_5v_9v_2$, $v_1v_8v_7$, and $v_3v_4v_5$; in (3), there are two regions: $v_0v_1v_2$ with hole $v_3v_5v_4$, and $v_3v_4v_5$ without holes.

The hidden edge test in line 11 has two cases. A subedge of a subdivision edge is hidden if it is behind t. A subedge of t' is hidden if it is behind the triangle that is visible to the left of the subdivision edges that contain its endpoints. For example, v_5v_6 is compared to the red triangle, v_6v_3 is compared to the background, and neither is hidden.

The hidden triangle test in line 6 of Algorithm 2 employs the second case of the hidden edge test. A triangle is hidden when its three edges are hidden. An edge is hidden unless it is in front of the triangle to the left of a subdivision edge that it intersects. A triangle that intersects no subdivision edges is declared hidden, which is correct because the test is applied solely to triangles outside the aggressive visible set, which cannot be completely visible.

Algorithm 3 The addition of one triangle to the visibility subdivision

Input: current visibility subdivision VS, triangle to be added t, viewpoint o, reference image G
Output: updated visibility subdivision VS

```
1:  t'(v0, v1, v2) = Project(t, o, G)
2:  E = VS ∩ t'
3:  if E = ∅ then
4:      s = FindCoveringTriangle(VS, t')
5:      if (s = ∅) ∨ (t' in front of s) then
6:          VS = VS ∪ {v0v1, v1v2, v2v0}
7:  else
8:      S = ∅
9:      for all edges e ∈ E do
10:         for all edges f in {v0v1, v1v2, v2v0} do S = S ∪ Split(e, f)
11:     Remove hidden edges from S
12:     VS = VS ∪ S
13: return VS
```

4.4 Complexity

The complexity of the exact visibility algorithm (Algorithm 2) is proportional to the number of calls to *AddTriangle* (line 14) and to the "is hidden" test (line 6). *AddTriangle* is called once per visible triangle. The number of calls "is hidden" tests equals the total number of undecided triangles in all the iterations of the algorithm. In practice, this sum is close to the number of triangles outside the aggressive visible set because almost every other triangle is classified as hidden or visible in the first iteration. Adding these bounds shows that the total number of calls roughly equals the number of triangles in the dataset.

The cost of a call to *AddTriangle* or of a "is hidden" test is dominated by the cost of finding the subdivision edge loops that intersect a projected triangle. The rest of the call takes linear time in the number of edges that are found. We implement edge finding with a quadtree that stores the bounding boxes of the subdivision edges. Quadtrees perform well although other algorithms have better worst-case complexity [12, Chapter 10]. Far fewer edges intersect the triangles than their bounding boxes, but finding these edges directly is impractical. Just finding the edges that have an endpoint in a triangle takes $\sqrt{n}$ time for n edges, as discussed in Section 3.3. Finding the rest is even harder.

4.5 Robustness

The key to the robust implementation of computational geometry algorithms is correct predicate evaluation. Floating point evaluation is fast, but rounding error can yield incorrect signs, which can cause large errors or even program crashes. Sign errors are most likely for *degenerate* predicates whose true sign is zero. Predicates can be evaluated correctly using arbitrary precision integer arithmetic, but this is slow. Moreover, degenerate predicates complicate predicate evaluation by introducing a third case at every branch.

We implement the exact visibility algorithm robustly using the ACP (Adaptive Controlled Perturbation) robustness technique. ACP is a state of the art form of Exact Computational Geometry that is faster and more general than prior work [33].

ACP consists of an input perturbation prior to running the algorithm and of a predicate evaluation algorithm. The input perturbation adds to each vertex coordinate a random number uniformly selected in $[-\delta, \delta]$, with $\delta = 10^{-8}$. This perturbation is negligible in terms of visibility, prevents degenerate predicates with high probability, and changes the value of most predicates by $O(\delta)$.

The algorithm for visibility subdivision construction and update (lines 2 and 14 in Algorithm 2) employs two predicates. The first predicate is $LT(a,b,c)$ for 2D points a,b,c. It equals -1 or 1 when the path abc is a right or left turn, and is degenerate when the points are collinear.

The predicate expression is $(c-b)\times(a-b)$ with $u\times v = u_x v_y - u_y v_x$. The 2D points are the projections of dataset vertices, hence are rational expressions in the 3D vertex coordinates and in the camera parameters. The predicate is used to implement two geometric tests. A point p is inside a counterclockwise triangle abc if $\mathrm{LT}(a,b,p) = 1$, $\mathrm{LT}(b,c,p) = 1$, and $\mathrm{LT}(c,a,p) = 1$. Line segments ab and cd intersect if $\mathrm{LT}(a,b,c) = -\mathrm{LT}(a,b,d)$ and $\mathrm{LT}(c,d,a) = -\mathrm{LT}(c,d,b)$. The second predicate determines the order of 3D points a and b along a ray with direction vector u. The predicate expression is $(a-b)\cdot u$ with $u\cdot v = u_x v_x + u_y v_y + u_z v_z$.

ACP evaluates predicates with double precision floating point interval arithmetic, which provides an interval that contains the true value of the predicate. The sign is determined unless the interval contains zero. Ambiguity is rare because the interval width is on the order of the floating point rounding unit 1.1×10^{-16}, whereas the exact value is $\mathrm{O}(\delta)$. We resolve ambiguous cases by increasing the precision of the interval arithmetic, and thus shrinking the interval. We start with a precision of 212 bits and increase it in increments of 53 bits until zero is excluded. The extended precision arithmetic uses the MPFR library [17].

Perturbation prevents degeneracy due to input in special position, such as three collinear triangle vertices. However, it cannot prevent degeneracy due to algebraic relations among derived quantities. Suppose an input triangle abc intersects an input edge uv at p. Let $\hat{u}$, $\hat{v}$, and $\hat{p}$ be the projections of u, v, and p. The predicate $\mathrm{LT}(\hat{u},\hat{v},\hat{p})$ is degenerate for all inputs. We call this type of degeneracy an *identity*. The most complicated identity in the exact visibility algorithm is three 2D edges that intersect at a point because they are the projections of the intersection edges of three triangles that intersect at a point.

ACP provides the first practical technique for detecting identities. The technique applies to all types of degeneracy, input and identity, and works with our without input perturbation. ACP evaluates an ambiguous predicate modulo several prime numbers and declares the predicate degenerate if all the residues are zero. ACP monitors the probability of a false degeneracy due to this probabilistic algorithm, so the user can increase the number of primes if the risk grows too large. Once an identity is detected, it is handled with special-case logic.

The robust implementation of the exact visibility algorithm ensures a correct output at a moderate computational cost. Visible triangles are never omitted from the visible set, hidden triangles are never included, and the program never crashes.

5 Spherical Particles

We have described our visibility algorithms for datasets modeled with triangles. The algorithms support any geometric primitive that can be tessellated. For example, the spherical particles used in a smoothed particle hydrodynamics (SPH) simulation can be tessellated and the resulting triangle meshes can be processed with our algorithms. However, for some applications it suffices to determine visibility at particle level, which saves the cost of computing partial particle visibility. We have extended our aggressive algorithm to support spherical particles directly. We greatly reduces the number of geometric primitives considered by the algorithm, as even a coarse regular octahedron tessellation implies eight triangles for every particle. Also, a coarse tessellation reduces the accuracy of the visible set.

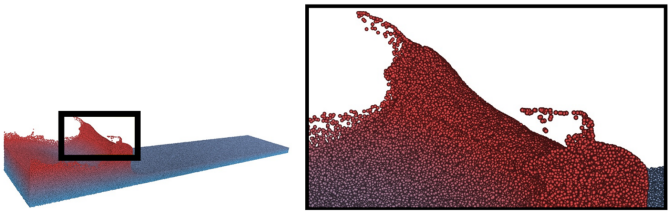

Fig. 8. *Left*: reference view on which our aggressive from-point particle visibility algorithm was run. *Right*: frame rendered from the visible particles; the zoom factor is 4x and there are 0.09% incorrect pixels

Fig. 9. Reference views on which our algorithms were run (*left*) and sample output frames from aggressive visible set (*right*).

The aggressive algorithm has to ensure that all *particles* are sampled. Algorithm 1 requires three changes. First, the algorithm has to determine the grid pixels that are covered by the projection of the particle, as needed for each of the three rendering passes (lines 2, 5, and 12). This is done with a conservative circular approximation of the elliptic projection of the particle centered at the particle center projection. Second, the algorithm has to tell whether a sampling location is inside the projection of the particle, as needed for each of the three rendering passes (lines 2, 6, 12). This is done exactly in 3D by checking whether the distance from the particle center to the sampling location ray is less than the particle radius. Third, the algorithm has to find the centroid of a particle fragment, which is the projection of the particle center if the center projects inside the pixel, or else the average of the intersection points between the pixel frame and the circular approximation of the particle projection. Our aggressive from-point particle visibility algorithm, produces high-quality frames, even for zoomed-in views (Figure 8).

6 Results

We have tested our algorithms on several datasets and viewpoints (Table 1, Figure 9, Figure 1). The datasets are modeled with triangles except for *Water* and *Fusion*, which are modeled with spherical particles. The *Impact* dataset was generated by a finite element simulation that modeled the dataset with beam, thin shell, thick shell, and hexahedral elements, which were then triangulated for visualization. The *Water* model was simulated with the Arbitrary Lagrangian Eulerian method and visualized with a triangle mesh. The resolution of the input images to our visibility algorithms is the same as the resolution of the output frames: 720×1280. For the *Manhattan midtown* view, we computed visibility in all directions, using a cubemap with a resolution of $6\times1024\times1024$, and averaged the results over its faces.

6.1 Quality

The quality of the visible set computed by our aggressive algorithm is given in rows 4 to 8 of Table 1. Row 4 gives the percentage of

Table 1. Experiment datasets (rows 1-3), aggressive algorithm quality (4-8), efficiency (9-10), and running time (11-12), and exact algorithm efficiency (13-15), and running time (16).

	Manhattan		Grass		Forest	Impact		Isosurface	Water	Fusion
	downtown	midtown	low	high		outside	inside			
1. Visual reference	Fig 1.1	Video	Fig 1.2	Fig 9.1	Fig 9.2	Fig 9.3	Fig 9.4	Fig 1.3	Fig 8	Fig 9.5
2. Dataset triangles	3.96M	3.96M	54.9M	54.9M	47.4M	2.08M	2.08M	497M	2.17M	500K
3. Visible set triangles	1.9%	2.3%	0.4%	14%	9.5%	7.0%	1.0%	3.0%	5.0%	9.0%
4. Vis. set completeness	89.0%	92.9%	83.3%	79.3%	72.3%	82.9%	79.4%	90.1%	97.8%	95.6%
5. Vis. subd. completeness	99.98%	99.99%	99.82%	98.19%	98.42%	99.94%	99.94%	99.66%	-	-
6. Average zoom factor	3.4x	3.4x	3.4x	3.4x	3.4x	4.8x	3.9x	3.4x	3.4x	2.4x
7. Average pixel error	0.03%	0.05%	0.11%	2.64%	2.11%	0.03%	0.01%	0.27%	0.09%	0.04%
8. Maximum pixel error	0.08%	0.17%	0.20%	3.71%	3.04%	0.12%	0.03%	0.61%	0.13%	0.07%
9. Average SL / pixel	1.7	1.4	6.7	37.1	28.4	2.0	1.3	209	2.0	3.7
10. Maximum SL / pixel	65	832	1507	736	1383	108	28	1670	27	29
11. Running time CPU [s]	1.7	0.4	6.1	9.5	12.2	1.4	0.5	464	3.4	1.2
12. Running time CUDA [s]	0.4	0.1	1.1	1.4	1.6	0.5	0.3	-	-	-
13. Visible tris (i1)	96.6%	95.7%	89.1%	81.2%	73.0%	96.1%	98.6%	94.7%	-	-
14. Decided tris (i1)	99.7%	98.5%	99.8%	90.4%	86.2%	98.3%	99.9%	95.3%	-	-
15. Iterations to convergence	2	2	3	3	3	4	3	3	-	-
16. Running time CPU [s]	25	27	160	12871	9000	90	15	146	-	-

visible triangles found. For an accurate estimate of the quality of the visible set, we provide the visibility subdivision completeness (row 5), defined as the percentage of the reference image area where the aggressive algorithm computes visibility correctly. Given a reference image point p, visibility is computed correctly at p if the triangle visible at p is part of the aggressive set. We compute the visibility subdivision completeness by comparing the visibility subdivisions built from the aggressive and the exact visible sets. The completeness is 98.2% and 98.4% for *Grass high* and *Forest*, which have tiny triangles and high depth complexity, and 99.7% or better in all the other cases.

We verify the quality of our aggressive visibility algorithm in a second way by rendering sequences of thousands of frames from the aggressive visible sets. The frames are rendered from the reference viewpoint with view direction and zoom factor changes. Row 6 gives the average zoom factor over the sequence. Rows 7 and 8 give the average and maximum percentages of incorrect pixels per frame over the sequence. A pixel is incorrect if it is not set by the same triangle that sets it when the frame is rendered from the entire dataset. The errors are small even though the sequences include zooming in. The maximum zoom factors for *Manhattan*, *Grass*, and *Forest* are 7x, 17x, and 10x. As our aggressive algorithm guarantees finding all triangles of a front surface, the few incorrect pixels occur between surfaces where they are less noticeable. The aggressive algorithm avoids holes in a visible surface efficiently in a purely sample-based fashion without computing triangle mesh continuity. For example, our aggressive set might omit a grass blade that is barely visible at the silhouette of a front grass blade, but it will never leave an objectionable hole in the middle of the front grass blade (Figure 10). We also refer the reader to the video that visualizes the error over a sequence of frames.

Our exact algorithm finds all visible triangles, from which accurate frames are rendered. Compared to rendering the entire dataset, finding the exact set brings not only the efficiency advantage of not rendering hidden triangles, but also the quality advantage of avoiding aliasing artifacts no matter how high the dataset complexity. Indeed, the visibility subdivision computed by the exact algorithm allows aggregating an output pixel color correctly from the contributions of *all* triangles visible at the pixel, appropriately weighted by their fractional pixel coverage [6]. By comparison, even when the dataset is rendered with high levels of conventional antialiasing, objectionable artifacts remain (Figure 11). We also refer the reader to the supplemental material that includes the exact and conventional 4x4 supersampled frames from Figure 11.

Fig. 10. Error pixels highlighted in red for a frame with a 17x zoom factor, rendered from our aggressive set (Figure 1, row 2 & col. 2).

6.2 Efficiency

The efficiency of the aggressive algorithm is summarized in rows 9 and 10 of Table 1. The number of sampling locations per pixel depends on the average image footprint of the triangles and on the presence of large blockers, so it is small for datasets like *Manhattan*, and larger for datasets like *Isosurface*. The optimal set of sampling locations has exactly one sampling location per visible triangle. The set of sampling locations constructed by our algorithm misses some visible triangles and has some useless members. Our algorithm can create useless sampling locations in several scenarios: (1) a visible triangle covers several pixels and a sampling location is created at each pixel; (2) a sampling location is constructed for each of several hidden triangles, and all find the same visible triangle; (3) two overlapping triangles are sampled with two sampling locations, as opposed to with one sampling location at the region of overlap. For *Grass high*, there are 9.7M visible triangles, so there should be at least 9.7M / (1280×720) = 10.5 sampling locations per pixel. Our algorithm completes 98.2% of the visibility subdivision with 37.1 sampling locations per pixel. For *Forest*, the algorithm completes 98.4% of the visibility subdivision with 28.4 sampling locations per pixel, compared to the optimal number of 7.1.

The efficiency of the exact algorithm is summarized in rows 13 to 15 of Table 1. Row 13 gives the completeness of the visible set after the first iteration (i1). The algorithm iterates until there are no more undecided triangles. Most work is done by the first iteration, which decides most triangles in the initial undecided set (row 14). The algorithm converges in at most four iterations (row 15).

6.3 Running time

We have developed multicore CPU and CUDA GPU implementations of the aggressive algorithm. The CPU running times are measured on an Intel i7-7700 CPU at 3.60GHz with 32GB memory, except that *Isosurface* is too large hence is tested on a machine with 24 2.27GHz Intel X7560 cores. The CUDA running times are measured on an NVIDIA GeForce RTX 3060 Laptop GPU with 6GB of VRAM. The

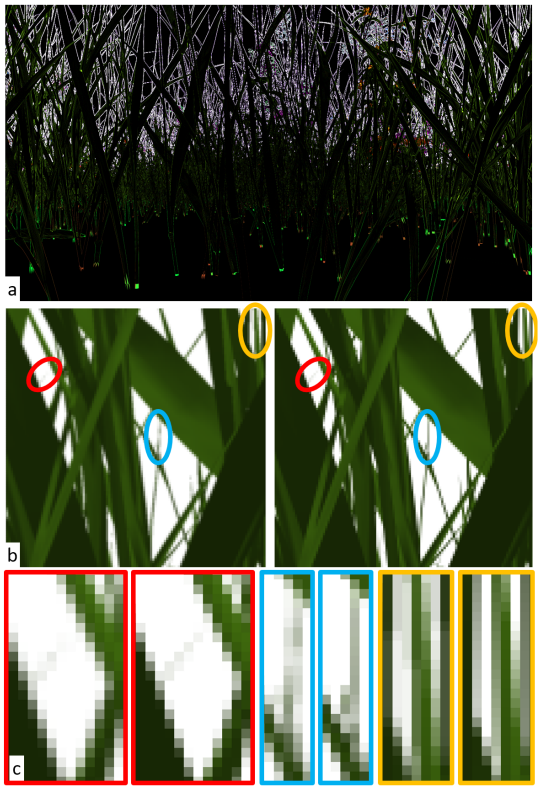

Fig. 11. (a) Difference between conventional 4x4 antialiasing and accurate antialiasing using the exact visible set computed using our from-point visibility algorithm, for the reference *Grass* image in Figure 1; 45.5% of the pixels of the conventional image have an incorrect color value; the average and maximum per pixel color channel errors are 2.96 and 117.65 on the [0, 255] scale. (b) and (c) Magnified detail of conventional antialiasing (left) and our accurate antialiasing (right). Even at the 4x4 supersampling level, conventional antialiasing does not sample the complex dataset adequately: the thin grass blades are interrupted (c, red), blurry (blue), and noisy (yellow).

running times are given in rows 11 and 12 of Table 1. The *Water* and *Fusion* times are not comparable to the others because these models are particle based.

The CUDA implementation proceeds in 8 stages. Stage 1 projects the vertices onto the image plane, using one thread per vertex. Stage 2 counts the triangles that intersect each pixel, using one thread per triangle. Stage 3 allocates each pixel arrays of triangles indices and of sampling locations, both of length equal to the number of triangles that intersect it, using one thread per pixel. Stage 4 assigns triangles to each pixel, using one thread per triangle. Stage 5 projects the triangles of each pixel onto the pixel center and records the visible one, using one thread per pixel. Stage 6 constructs the sampling locations for each pixel, using 256 threads per pixel. Stage 7 projects the triangles of each pixel onto the sampling locations of the pixel and records the visible triangle at each sampling location, using 256 threads per pixel. Stage 8 collects the visible triangles from the sampling locations, using one thread per triangle. Stage 7 is the slowest and stage 6 is second slowest; together they take 90% of the running time.

The running time does not increase quadratically with the number of sampling locations per pixel, as predicted by the complexity analysis in Section 3.3. For the CPU implementation, the time per million triangles is between 0.1s and 0.7s, and is uncorrelated with the number of sampling locations per pixel. For example, *Grass high* has 37 locations on average versus 7 for *Grass low*, yet is only 50% slower. The absence of an empirical quadratic dependency on the number of sampling locations per pixel was also confirmed indirectly by the lack of speedup observed when storing sampling locations in a quadtree. We explain this by the fact that the lists of sampling locations are relatively short, and constant factors dominate. The million triangle times for the CUDA implementation are between 0.02s and 0.24s, and decrease as the input size increases.

The running times for the exact visibility algorithm are measured on one core of the 24-core machine (row 16 of Table 1). They exclude the running time of the aggressive algorithm that provides the initial visible set. Not adding hidden triangles to the visibility subdivision during its incremental construction makes the exact visibility algorithm practical. It is slowest for *Grass High*, which requires many small updates to the initial visibility subdivision due to dataset fragmentation and high depth complexity. The nearly half a billion triangles of *Isosurface* are processed in less than 3min because the front surface hides almost everything else. However, the exact algorithm exceeds the memory of our computer, so for *Isosurface* we tested a non robust implementation that uses all 24 cores. Building the visibility subdivision for the *Manhattan* dataset from all the triangles takes four times as long, which confirms the benefit of restricting the visibility construction to visible triangles.

6.4 Comparison to Prior Art

We compare our aggressive algorithm to two prior approaches.

6.4.1 Comparison to uniform sampling

The first approach aggregates the visible set by uniformly sampling the space of visualization rays with conventional images. We compare our algorithm to computing visibility with a conventional image with resolutions ranging from the resolution of the grid used by our algorithms to 256×256 times the grid resolution (Table 2). For each row, the table highlights the uniform sampling level values that bracket the aggressive algorithm value, e.g. 1 and 4 bracket 1.7 for *Manhattan* row 1. The aggressive algorithm generates more complete visible sets and visibility subdivisions than uniform sampling for the same number of sampling locations per pixel. To match the aggressive visibility set, uniform sampling requires between 64 and 256 samples per pixel in the most favorable test (*Grass* and *Forest*), and between 1024 and 4096 samples per pixel in the least favorable test (*Isosurface*); to match subdivision completeness, it requires between 16 (*Grass*) and 4096 (*Manhattan* and *Isosurface*) samples per pixel.

We measured the running times of uniform supersampling and of our aggressive algorithm on the GPU, using the conventional GPU pipeline and our CUDA implementation, respectively. The exception is *Isosurface* which is too large for the laptop GPU, so we ran our aggressive algorithm on the 24-core CPU implementation, and uniform supersampling on a server with two 3.47GHz Intel Xeon CPUs, 48GB of RAM, and four 6GB NVIDIA GeForce GTX 1060 GPUs. The aggressive algorithm is faster than uniform sampling that achieves comparable visible set and visibility subdivision completions, i.e., the shaded cells in rows 4 are to the left of the shaded cells in rows 2 and 3. The aggressive algorithm is slower than uniform supersampling with a comparable number of samples per pixel for *Manhattan*, it has similar performance for *Grass*, and then is faster for *Forest*, which requires large supersampling factors, i.e., between 4x4 and 8x8. The CPU running time on *Isosurface* is comparable to the GPU time for the samplings that bracket its output quality.

6.4.2 Comparison to state-of-the-art sample-based visibility

The second prior art approach to which we compare our algorithms is *Guided Visibility Sampling* (GVS), an aggressive from region visibility algorithm that samples heuristically the space of visualization rays that originate from a rectangle or a box [42]. Early rays generated stochastically guide the generation of subsequent rays based on two heuristics: *adaptive border sampling*, which looks for new visible triangles adjacent to visible triangles that were already found, and *reverse sampling*, which looks for visible triangles in unsampled but accessible space defined by depth discontinuities. The iterative search for visible triangles is terminated heuristically based on a predetermined ray budget or on a minimum threshold for the rate of visible triangle discovery. GVS was recently updated to GVS++, which has improved heuristics and performance optimizations, and which takes advantage

Table 2. Comparison to uniform supersampling. The grey shaded cells indicate the supersampling levels whose values bracket the aggressive algorithm value (green shaded cells).

Manhattan downtown	**Aggr.**	**1 × 1**	**2 × 2**	**4 × 4**	**8× 8**	**16 × 16**	**32 × 32**	**64 × 64**	**128 × 128**	**256 × 256**
1. SL / pixel	1.7	1	4	16	64	256	1024	4096	16384	65536
2. Visible triangles [%]	89.0	35.2	49.6	64.5	77.9	87.8	93.6	96.6	98.1	98.9
3. Vis. sub. compl. [%]	99.98	97.4	99.6	99.9	99.9	99.9	99.9	∼100	∼100	∼100
4. Time [s]	0.4	0.06	0.2	0.88	3.2	12	30	122	400	1100
Grass low	**Aggr.**	**1 × 1**	**2 × 2**	**4 × 4**	**8× 8**	**16 × 16**	**32 × 32**	**64 × 64**	**128 × 128**	**256 × 256**
1. SL / pixel	6.7	1	4	16	64	256	1024	4096	16384	65536
2. Visible triangles [%]	83.3	31.0	47.5	63.6	77.2	87.0	93.0	96.3	98.1	99.0
3. Vis. sub. compl. [%]	99.82	96.3	99.0	99.7	99.9	99.9	99.9	99.9	99.9	∼100
4. Time [s]	1.1	0.17	0.61	2.2	7.9	25	98	380	1134	4055
Forest	**Aggr.**	**1 × 1**	**2 × 2**	**4 × 4**	**8× 8**	**16 × 16**	**32 × 32**	**64 × 64**	**128 × 128**	**256 × 256**
1. SL / pixel	28.4	1	4	16	64	256	1024	4096	16384	65536
2. Visible triangles [%]	72.3	6.90	20.1	42.2	64.3	80.1	89.4	94.6	97.2	98.6
3. Vis. sub. compl. [%]	98.42	60.3	80.3	94.1	97.8	98.8	99.3	99.9	99.9	∼100
4. Time [s]	1.6	0.18	0.63	2.2	8.1	31	110	391	1132	4222
Isosurface	**Aggr.**	**1 × 1**	**2 × 2**	**4 × 4**	**8× 8**	**16 × 16**	**32 × 32**	**64 × 64**	**128 × 128**	**256 × 256**
1. SL / pixel	193	1	4	16	64	256	1024	4096	16384	65536
2. Visible triangles [%]	91.2	2.46	9.84	33.5	62.9	80.6	89.3	93.2	94.8	96.2
3. Vis. sub. compl. [%]	99.5	5.23	20.75	64.0	93.1	98.7	99.3	99.5	99.7	99.9
4. Time [s]	464.4	33.4	34.6	35.8	36.4	145.6	582.4	2329	9318	37273

of the Vulkan graphics API and of RTX ray tracing [29]. Since GVS++ is a from-region visibility algorithm, and since our algorithms solve from-point visibility, we make the comparison possible in two ways: (1) by forcing GVS++ to start rays from a single point, and (2) by running our aggressive algorithm multiple times, from viewpoints that sample a given view region.

GVS++ runs out of memory for all datasets except for *Manhattan*. We have used the default GVS++ settings recommended by its authors, and we have also experimented with parameter settings as described below. We note that while GVS++ has seven or more parameters that the application developer has to set, our aggressive algorithm has a single parameter, i.e., the resolution of the pixel grid where the visible triangles are found, with a default value equal to the resolution of the output image.

(1) For a first set of experiments we restricted GVS++ to from-point visibility by reducing the view region to a view point.

In terms of quality of the visibility solution, for the midtown viewpoint, our aggressive algorithm completes the visibility subdivision at the 99.9860% level with 8.7 million sampling locations. GVS++ completes the visibility subdivision at a similar level of 98.9857%, but with 983 million rays. In other words, in this equal quality comparison, our aggressive algorithm has a massive efficiency advantage, sampling visibility with *two order of magnitude* fewer rays than GVS++. In terms of number of visible triangles found, GVS++ finds 399,401 of the 406,036 triangles in the exact visible set, whereas our aggressive algorithm finds 374,172 of them. This confirms that the number of visible triangles found does not accurately quantify the quality of the visibility solution, as defined by the visibility subdivision completion. Our exact from-point visibility algorithm allows for the computation of the visibility subdivision completion, providing an important tool for the evaluation of approximate visibility algorithms. GVS++ does not converge on the exact visible set even after casting *one billion* rays.

In terms of running time, using the *time* utility, the CUDA implementation of our aggressive algorithm takes 0.8s real time (wall clock time), and GVS++ takes 5s. Using NVIDIA's Nsight GPU profiler, the GPU time is 0.7s for ours and 4.7s for GVS++ (across 4 GPU contexts: 0.1s + 2.9s + 1.7s). GVS++ reports 0.8s which in our experiments does not corroborate with total actual time nor with GPU time. We conclude that our CUDA implementation of our aggressive algorithm is at least as fast as GVS++.

Furthermore, the GVS++ implementation is not robust: the visible set it finds also contains 3,232 triangles that are *not* visible. GVS++ does not provide the visibility subdivision, as needed for applications of visibility such as antialiasing, and the visibility subdivision has to be computed from the visible triangles in an additional step. Finally, compared to our exact algorithm, GVS++ fails to find all visible triangles and to complete the visibility subdivision even after casting nearly one billion rays.

(2) For a second set of experiments we compared our approach to GVS++ on from-region visibility. The region is modeled as a box, which GVS++ takes directly as input. To compute from-region visibility with our approach we ran our aggressive from-point algorithm on viewpoints selected randomly inside the view box. The example in Figure 12 uses the *Manhattan* dataset and a 30m view cube centered at the midtown viewpoint, which our approach sampled with 10,000 viewpoints. For each viewpoint, the view direction is random and the pixel grid resolution is 1000×1000. GVS++ shoots visibility rays from within the box provided as input.

The top graph shows that our approach accumulates ∼13 billion sampling locations to find 499,277 visible triangles. With the default parameters (the *GVS++ default* series in the graph), GVS++ stops after querying visibility with 3.45 billion rays, finding 476,159 visible triangles. We have also experimented with GVS++ parameter configurations to make the termination condition as difficult to satisfy as possible (the *GVS++ max* series in the graph). Specifically, the *GVS++ max* configuration terminates if no triangles have been found by 20 consecutive iterations, where each iteration shoots 100 million random rays. The *GVS++ max* run terminates after querying visibility with 56 billion rays, and finds 487,083 visible triangles. The x axis of the graph is truncated at 14 billion visibility queries for illustration clarity, as the *GVS++ max* is essentially flat from 14 to 56 billion visibility queries. Although *GVS++ max* makes 56 billion queries compared to the 13 billion of our method, the visible set found by *GVS++ max* is still 12 thousand triangles short of that found by our method. Neither of the two GVS++ lines overtake the line for our method, which means that our method finds visible triangles more efficiently no matter the visibility query budget. The graph also shows the visible set accumulated with a conventional framebuffer (*Uniform sampling*). The total number of visibility queries is 10 billion, i.e, 10,000 views with 1000×1000 pixels each, and the number of visible triangles is consistently and substantially lower than for our method. Our method extends the set of sampling locations from 10 to 13 billion to find another 60,148 visible triangles.

The bottom graph shows the number of visible triangles found as a function of time. *GVS++ default* terminates after 5.9s to find 476,159 visible triangles (point a on the graph), which is surpassed by our method in 146s (point b). *GVS++ max* terminates after 109s to find 487,083 visible triangles (point c), which is surpassed by our method in 374s (point d). For lower time budgets, in the 10s range, the default configuration of GVS++ is more time efficient. For medium time

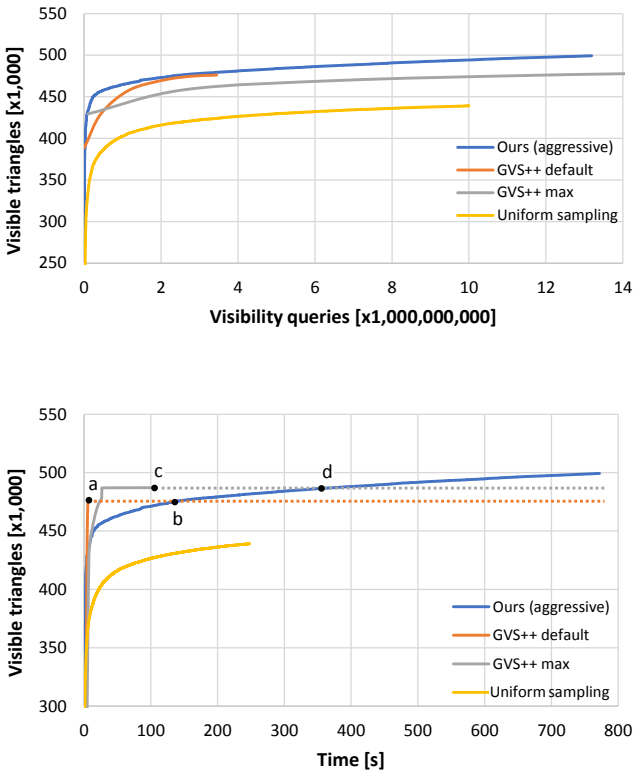

Fig. 12. Efficiency comparisons between our approach and the prior art approaches GVS++ and regular sampling, on from-region visibility.

budgets, in the 100s range, *GVS++ max* is more time efficient. For time budgets of 400s and above, our approach has the time efficiency advantage.

7 Conclusions and Future Work

We have described a novel approach to from-point visibility based on image generalization. The regular sampling grid of a conventional image is enhanced with sampling locations defined by dataset geometry. A few additional sampling locations per pixel reveal most visible triangles. We provide a visibility approximation that guarantees that no completely visible triangle is omitted, no matter how small its footprint, which guarantees front surface continuity. We couple sample-based and continuous visibility analyses of the image plane to complete the visible set efficiently. Our exact visibility algorithm is implemented robustly. The exact visible set allows us to render *accurate* images by taking into account all triangles visible at a pixel, overcoming the aliasing artifacts that pervade visualizations of complex datasets even when high levels of conventional antialiasing are used.

Our aggressive algorithm finds some partially visible triangles, but makes no guarantee about them. Future work could strengthen the quality guarantee by finding all the visible triangles that are partially occluded by a front surface. This could largely be achieved by sampling each triangle at its vertices. Future work should examine the trade-off between the cost of the additional sampling locations and the increase in visibility subdivision completeness.

The grid resolution of the aggressive algorithm influences the quality of the visible set and the efficiency of its computation. A higher grid resolution implies smaller triangle fragments, which increases the probability that a partially visible triangle has a completely visible fragment, and thus that the triangle is found. Furthermore, a higher grid resolution reduces the maximum number of sampling locations per pixel, which has performance implications as described above. We set the grid resolution to the intended output image resolution, based on the consideration that the resulting approximate visible set should not make errors larger than one output image pixel. Whereas this is a reasonable approach, it remains heuristic. When coupled with a hierarchical data structure for storing the sampling locations, coarser grid resolutions are possible, all the way to the extreme case of not using a grid at all. At the other end of the spectrum, very high resolution grids can limit the number of sampling locations per pixel, but increase the number of pixels. For example, reducing the grid resolution from 1280x720 to 640x360 changes the running times by a factor of 0.41 for *Manhattan downtown*, of 1.74 for *Grass low*, of 1.16 for *Grass high*, of 1.2 for *Forest*, of 0.47 for *Impact outside*, and of 0.36 for *Impact reverse*. There is a speedup when the triangles are large and a slowdown when they are small. The number of visible triangles decreases consistently with factors of .96, .94, .92 , .92, .96, and .98. In addition to the empirical approach of trying multiple grid resolutions to optimize the visible set accuracy versus running time, future work could compute the optimal grid resolution for a given viewpoint based on the geometric properties of the dataset, such as depth complexity and complexity uniformity.

We have used our robust exact from-point visibility algorithm to estimate the quality of approximate visible sets, such as those produced by our aggressive algorithm and GVS++, in terms of visibility subdivision completeness and numerical robustness. Future work could use our exact algorithm to investigate the quality of other approximate visibility algorithms, including the numerical robustness of industrial grade rasterizers and ray tracers. For example, one could investigate the number of false positives when uniformly sampling a complex dataset at high resolution with a conventional GPU rendering pass.

Another topic for future work is extending the exact visibility algorithm to particle datasets, thereby avoiding the time and memory cost of tesselation. The projection of a particle has curved edges, which are harder to manipulate than line segments. The curves must be decomposed into monotone segments and curve intersection points must be isolated. These algebraic computations are impractical for large datasets using current algorithms. So far we have considered a single GPU and future work could examine speeding up visibility computation using multiple GPUs. Our method breaks up the from-region visibility computation problem into many, small, and uniform-sized sub-problems and therefore has the potential for speedup scalability with the number of GPUs. GVS++ could leverage multiple GPUs by voxelizing the view region, and future work should compare the efficiency of parallel implementations of the two approaches.

Another direction of future work is to apply our visibility algorithms to solve higher level problems in graphics and visualization. For example, a hard shadow algorithm has to evaluate how much of an output image pixel is visible from the point light source, which can be done by reprojecting a pixel sample to the shadow map to query its visibility. Our exact algorithm allows evaluating such queries accurately, whereas using an incomplete (aggressive) visible set from the point light source leads to an incorrect output pixel color that is too bright. Our current work compares our algorithms to prior art analytically, in terms of asymptotic computational load, and empirically, in terms of of actual number of visibility queries and running time. As battery powered computing platforms proliferate, such as laptops, tablets, phones, and all-in-one VR headsets, future work should also compare approaches in terms of power consumption.

A final direction of future work is to use the image generalization paradigm to develop more general visibility algorithms. Our from-point algorithms compute visibility for a 2D visualization ray space. We have already shown that our algorithms can provide a good approximation of visibility for the 4D ray space of from-rectangle visibility, by aggregating visibility over a set of viewpoints sampling the view rectangle. However, sampling the view rectangle is both redundant, as the visible sets of nearby viewpoints have substantial overlap, and approximate, as no view rectangle sampling resolution can guarantee finding all visible triangles. We envision generalizing the zero-dimensional image plane sampling locations used in from-point visibility to 1D and 2D sampling locations that can capture the triangles visible from a segment and a rectangle efficiently, accurately, and non-redundantly. The exact visibility algorithm can be generalized to an interval of viewpoints by constructing the initial subdivision and updating it at viewpoints where structural changes occur, for example a triangle becomes visible or vanishes. The ultimate goal is to develop 3D visibility sampling

locations, suitable for the 5D visualization ray space of from-rectangle visibility of dynamic datasets.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. 2212200 and 2219842.

References

[1] O. Apostu, F. Mora, D. Ghazanfarpour, and L. Aveneau. Analytic ambient occlusion using exact from-polygon visibility. *Computers & Graphics*, 36(6):727–739, 2012.

[2] T. Auzinger, M. Wimmer, and S. Jescke. Analytic visibility on the gpu. *Computer Graphics Forum*, 32(2pt4):409–418, 2013.

[3] J. Bittner, V. Havran, and P. Slavik. Hierarchical visibility culling with occlusion trees. In *Computer Graphics International, 1998. Proceedings*, pp. 207–219. IEEE, 1998.

[4] J. Bittner, O. Mattausch, P. Wonka, V. Havran, and M. Wimmer. Adaptive global visibility sampling. *ACM Transactions on Graphics (TOG)*, 28(3):94, 2009.

[5] J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum*, 23(3):615–624, 2004.

[6] L. Carpenter. The a-buffer, an antialiased hidden surface method. *ACM SIGGRAPH Computer Graphics*, 18(3):103–108, 1984.

[7] E. Catmull. A hidden-surface algorithm with anti-aliasing. *ACM SIGGRAPH Computer Graphics*, 12(3):6–11, 1978.

[8] A. Chandak, C. Lauterbach, M. Taylor, Z. Ren, and D. Manocha. Ad-frustum: Adaptive frustum tracing for interactive sound propagation. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1707–1722, 2008.

[9] S. Charneau, L. Aveneau, and L. Fuchs. Exact, robust and efficient full visibility computation in plücker space. *The Visual Computer*, 23(9-11):773–782, 2007.

[10] D. Cohen-Or, Y. L. Chrysanthou, C. T. Silva, and F. Durand. A survey of visibility for walkthrough applications. *Visualization and Computer Graphics, IEEE Transactions on*, 9(3):412–431, 2003.

[11] J. Cui, P. Rosen, V. Popescu, and C. Hoffmann. A curved ray camera for handling occlusions through continuous multiperspective visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 16(6):1235–1242, 2010.

[12] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, third ed., 2008.

[13] X. Décoret, G. Debunne, and F. Sillion. Erosion based visibility preprocessing. In *Proceedings of the 14th Eurographics workshop on Rendering*, pp. 281–288. Eurographics Association, 2003.

[14] F. Durand. *3D Visibility: analytical study and applications*. PhD thesis, Université Joseph Fourier, 2010.

[15] F. Durand, G. Drettakis, and C. Puech. The 3d visibility complex. *ACM Transactions on Graphics (TOG)*, 21(2):176–206, 2002.

[16] F. Durand, G. Drettakis, J. Thollot, and C. Puech. Conservative visibility preprocessing using extended projections. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 239–248. ACM Press/Addison-Wesley Publishing Co., 2000.

[17] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple precision binary floating point library with correct rounding. *ACM Transactions on Mathematical Software*, 33:13, 2007.

[18] J. Friedman, J. Bentley, and R. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3:209–226, 1977.

[19] G. W. Furnas. Generalized fisheye views. *SIGCHI Bull.*, 17(4):16–23, 1986.

[20] M. T. Goodrich. A polygonal approach to hidden-line and hidden-surface elimination. *CVGIP: Graphical Models and Image Processing*, 54(1):1–12, 1992.

[21] C. J. Gribel, R. Barringer, and T. Akenine-Möller. High-quality spatio-temporal rendering using semi-analytical visibility. *ACM Transactions on Graphics (TOG)*, 30(4):54, 2011.

[22] D. Haumont, O. Mäkinen, and S. Nirenstein. A low dimensional framework for exact polygon-to-polygon occlusion queries. In *Proceedings of the Sixteenth Eurographics conference on Rendering Techniques*, pp. 211–222. Eurographics Association, 2005.

[23] P. S. Heckbert and P. Hanrahan. Beam tracing polygonal objects. *ACM SIGGRAPH Computer Graphics*, 18(3):119–127, 1984.

[24] J. Hladky, H.-P. Seidel, and M. Steinberger. The camera offset space: Real-time potentially visible set computations for streaming rendering. *ACM Trans. Graph.*, 38(6), nov 2019. doi: 10.1145/3355089.3356530

[25] W. Hunt, M. Mara, and A. Nankervis. Hierarchical visibility for virtual reality. *Proc. ACM Comput. Graph. Interact. Tech.*, 1(1), jul 2018. doi: 10.1145/3203191

[26] G. S. Johnson, J. Lee, C. A. Burns, and W. R. Mark. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Transactions on Graphics (TOG)*, 24(4):1462–1482, 2005.

[27] T. R. Jones and R. N. Perry. Antialiasing with line samples. In *Rendering Techniques 2000*, pp. 197–205. Springer, 2000.

[28] M. J. Katz, M. H. Overmars, and M. Sharir. Efficient hidden surface removal for objects with small union size. *Computational Geometry*, 2(4):223–234, 1992.

[29] T. Koch and M. Wimmer. Guided visibility sampling++. *Proc. ACM Comput. Graph. Interact. Tech.*, 4(1), apr 2021. doi: 10.1145/3451266

[30] N. Max and K. Ohsaki. Rendering trees from precomputed z-buffer views. In *Rendering Techniques' 95*, pp. 74–81. Springer, 1995.

[31] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pp. 39–46. ACM, 1995.

[32] K. Mehlhorn and S. Naher. Dynamic fractional cascading. *Algorithmica*, 5(2):215–241, 1990.

[33] V. Milenkovic and E. Sacks. Efficient predicate evaluation using probabilistic degeneracy detection. *International Journal of Computational Geometry & Applications*, 32(01n02):39–54, 2022.

[34] F. Mora and L. Aveneau. Fast and exact direct illumination. In *Computer Graphics International 2005*, pp. 191–197. IEEE, 2005.

[35] S. Nirenstein and E. H. Blake. Hardware accelerated visibility preprocessing using adaptive sampling. *Rendering Techniques*, 2004:15th, 2004.

[36] R. Overbeck, R. Ramamoorthi, and W. R. Mark. A real-time beam tracer with application to exact soft shadows. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, pp. 85–98. Eurographics Association, 2007.

[37] S. Pałka, B. Głut, and B. Ziółko. Visibility determination in beam tracing with application for real-time sound simulation. *Computer Science*, 15, 2014.

[38] V. Popescu, J. Eyles, A. Lastra, J. Steinhurst, N. England, and L. Nyland. The warpengine: An architecture for the post-polygonal age. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 433–442. ACM Press/Addison-Wesley Publishing Co., 2000.

[39] J. Shade, S. Gortler, L.-w. He, and R. Szeliski. Layered depth images. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pp. 231–242. ACM, 1998.

[40] M. Sharir and M. H. Overmars. A simple output-sensitive algorithm for hidden surface removal. *ACM Transactions on Graphics (TOG)*, 11(1):1–11, 1992.

[41] K. Weiler and P. Atherton. Hidden surface removal using polygon area sorting. *ACM SIGGRAPH Computer Graphics*, 11(2):214–222, 1977.

[42] P. Wonka, M. Wimmer, K. Zhou, S. Maierhofer, G. Hesina, and A. Reshetov. Guided visibility sampling. *ACM Transactions on Graphics (TOG)*, 25(3):494–502, 2006.

[43] C. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, eds., *Handbook of discrete and computational geometry*, chap. 41, pp. 927–952. CRC Press, Boca Raton, FL, second ed., 2004.

[44] J. Yu and L. McMillan. General linear cameras. In *Computer Vision-ECCV 2004*, pp. 14–27. Springer, 2004.

[45] H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff III. Visibility culling using hierarchical occlusion maps. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 77–88. ACM Press/Addison-Wesley Publishing Co., 1997.

[46] Y. Zhou, L. Wu, R. Ramamoorthi, and L.-Q. Yan. Vectorization for fast, analytic, and differentiable visibility. *ACM Trans. Graph.*, 40(3), jul 2021. doi: 10.1145/3452097