FastV: From-point Visibility Culling and Application to Sound Rendering

Anish Chandak†1 Lakulish Antani‡1 Micah Taylor§1 and Dinesh Manocha¶1

[1]Univeristy of North Carolina at Chapel Hill

Abstract

We present an efficient technique to compute the potentially visible set (PVS) of triangles in a complex 3D scene from a viewpoint. The algorithm computes a conservative PVS at object space accuracy. Our approach traces a high number of small, volumetric frusta and computes blockers for each frustum using simple intersection tests. In practice, the algorithm can compute the PVS of CAD and scanned models composed of millions of triangles at interactive rates on a multi-core PC. We also use the visibility algorithm to accurately compute the reflection paths from a point sound source. The resulting sound propagation algorithm is $10-20X$ *faster than prior accurate geometric acoustic methods.*

1. Introduction

Visibility computation is a widely-studied problem in computer graphics and related areas. Given a scene, the goal is to determine the set of primitives visible from a single point (i.e. from-point visibility), or from any point within a given region (i.e. from-region visibility). At a broad level, these algorithms can be classified into object space and image space algorithms. The object space algorithms operate at object-precision and use the raw primitives for visibility computations. The image space algorithms resolve visibility based on a discretized representation of the objects and the accuracy typically corresponds to the resolution of the final image. These algorithms are able to exploit the capabilities of rasterization hardware and can render large, complex scenes composed of tens of millions of triangles at interactive rates using current GPUs.

In this paper, we primarily focus on from-point, object space conservative visibility, whose goal is to compute a superset of visible geometric primitives. Such algorithms are useful for walkthroughs, shadow generation, global illumination and occlusion computations. Another application for object space visibility algorithms is accurate computation of reflection paths for acoustic simulation or sound rendering. Given a point sound source, 3D models of the environment with material data, and the receiver's position, geometric acoustic (GA) methods perform multiple orders of reflections from the obstacles in the scene to compute the impulse response (IR). Sample-based propagation algorithms, such as stochastic ray-tracing for GA can result in statistical errors or inaccurate IRs [Len93]. As a result, we need to use object space visibility techniques, such as beam tracing [FCE*98, LSLS09], to accurately compute the propagation paths. However, current object space visibility algorithms only work well on simple scenes with tens of thousands of triangles or with large convex occluders. There is a general belief that it is hard to design fast and practical object space visibility algorithms for complex 3D models [Gha01].

Main Results: We present a novel algorithm (FastV) for conservative, from-point visibility computation. Our approach is general and computes a potentially visible set (PVS) of scene triangles from a given view point. The main idea is to trace a high number of 4-sided volumetric frusta and efficiently compute simple connected blockers for each frustum. We use the blockers to compute a far plane and cull away the non-visible primitives, as described in Section 3.

Our guiding principle is to opt for simplicity in the choice of different components, including frustum tracing, frustum-intersection tests, blocker and depth computations. The main

† achandak@cs.unc.edu
‡ lakulish@cs.unc.edu
§ taylormt@cs.unc.edu
¶ dm@cs.unc.edu

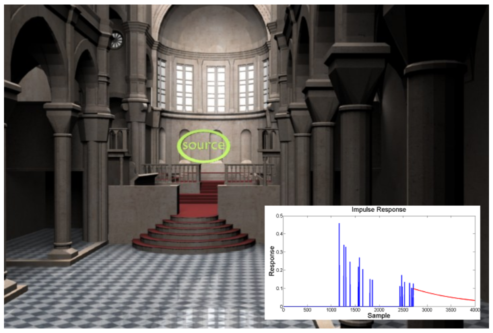

Figure 1: ***Fast Acoustic Simulation**: We use FastV for computing accurate reflection paths in this Cathedral model with 80K triangles. Our propagation algorithm performs three orders of reflections from the source and computes the IR at the receiver in less than 5 seconds on a 16-core PC. To the best of our knowledge, ours is the first efficient and accurate propagation algorithm to handle models of this complexity.*

contribution of the paper is primarily in combining known algorithms (or their extensions) for these parts. Overall, FastV is the first practical method for visibility culling in complex 3D models due to the following reasons:

1. Generality: Our approach is applicable to all triangulated models and does not assume any large objects or occluders. The algorithm proceeds automatically and is not susceptible to degeneracies or robustness issues.

2. Efficiency: We present fast and conservative algorithms based on Plücker coordinates to perform intersection tests. We use hierarchies along with SIMD and multi-core capabilities to accelerate the computations. In practice, our algorithm can trace $101-200$K frusta per second on a single 2.93 Ghz Xeon Core on complex models with millions of triangles.

3. Conservative: Our algorithm computes a conservative superset of the visible triangles at object-precision. As the frustum size decreases, the algorithm computes a tighter PVS. We have applied the algorithm to complex CAD and scanned models with millions of triangles and simple dynamic scenes. In practice, we can compute conservative PVS, which is within a factor of $5-25\%$ of the exact visible set, in a fraction of a second on a 16-core PC (as described in Section 4).

Accurate Sound Propagation: We use our PVS computation algorithm to accurately compute the reflection paths from a point sound source to a receiver, as described in Section 5. We use a two phase algorithm that first computes image-sources for scene primitives in the PVS computed for primary (or secondary) sources. This is followed by finding valid reflection paths to compute actual contributions at the receiver. We have applied our algorithm to complex models with tens of thousands of triangles. In practice, we observe performance improvement of up to 20X using a single-core implementation over prior accurate propagation methods that use beam tracing.

2. Previous Work

The problem of visibility has been extensively studied in computer graphics, computational geometry, acoustic simulation and related areas for more than four decades. We refer the readers to excellent recent surveys [Dur99, COCSD03]. Due to space limitations, we only give a brief overview of some object space and sampling-based methods.

Object space visibility computations: There is extensive work on object-precision algorithms, including methods for hidden surface removal [Gha01] and exact visibility computation from a point using beam tracing [HH84, FCE*98, ORM07] or Plücker coordinates [Nir03]. Many exact algorithms have also been proposed for region-based visibility [Dur99, DD02, Nir03, BW05]. There is considerable literature on conservative visibility computations from a point [BHS98, CT97, HMC*97, LG95] or from a region [KCCO00, LSCO03, Tel92]. Some of these algorithms have been designed for special types of models, e.g. architectural models represented as cells and portals, 2.5D urban models, scenes with large convex occluders, etc. It is also possible to perform conservative rasterization [AMA05] on current GPUs to compute an object-precision PVS from a point.

Image space or sample-based visibility computations: These methods either make use of rasterization hardware or ray-shooting techniques to compute a set of visible primitives [COCSD03]. Most of these methods tend to be either approximate or aggressive [NB04, WWZ*06]. Current GPUs provide support for performing occlusion queries for from-point visibility and are used for real-time display of complex 3D models on commodity GPUs [KS00, MBW08].

3. FastV: Visibility Computation

In this section, we present our conservative visibility computation algorithm. The inputs to our algorithm are: a view point ($\mathbf{v} \in \Re^3$), a set of scene primitives (Π), and a viewing frustum (Φ), with an apex at $\mathbf{v}$. Our goal is to compute a subset of primitives $\pi \subseteq \Pi$ such that every primitive $p \in \Pi$, which is hit by some ray $r \in \Phi$ is included in the computed subset π. The subset π is called the potentially visible set (PVS). The smallest such PVS is the set of exactly visible primitives (π_{exact}). The subset π computed by our algorithm is conservative, i.e., $\pi \supseteq \pi_{exact}$. For the rest of the paper, we assume that the primitives are triangles, though our algorithm can be modified to handle other primitives. We also assume that the connectivity information between the scene triangles is precomputed. We exploit this connectivity for efficient computation; however our approach is also applicable to polygon soup models. In order to perform fast intersection

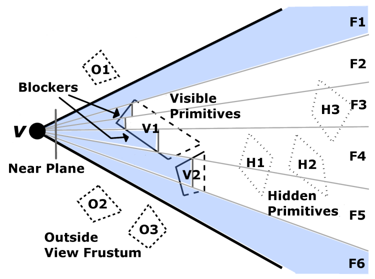

Figure 2: ***Overview****: We divide the view-frustum with an apex at* **v***, into many small frusta. Each frustum is traced through the scene and its far plane is updated when it is blocked by a connected blocker. For example, frustum F_5 is blocked by primitives of object V_2 but frustum F_1 has no blockers.*

tests, we store the scene primitives in a bounding volume hierarchy (BVH) of axis-aligned bounding boxes (AABBs). This hierarchy is updated for dynamic scenes.

3.1. Overview

We trace pyramidal or volumetric beams from the viewpoint. Prior beam tracing algorithms perform expensive exact intersection and clipping computations of the beam against the triangles and tend to compute π_{exact}. Our goal is to avoid these expensive clipping computations, and rather perform simple intersection tests to compute the PVS. Moreover, it is hard to combine two or more non-overlapping occluders (i.e. occluder fusion) using object space techniques. This is shown in Figure 2, where object H_1 is occluded by the combination of V_1 and V_2. As a result, prior conservative object space techniques are primarily limited to scenes with large occluders.

We overcome these limitations by tracing a high number of relatively small frusta and computing the PVS for each frustum independently. In order to compute the PVS for each frustum, we try to compute a *blocker* that is composed of connected triangles (see Figure 3). The blockers are computed on the fly and need not correspond to a convex set or a solid object; rather they are objects that are homeomorphic to a disk. We use simple and fast algorithms to perform intersection tests and blocker computation.

Frustum Tracing: We use a simple 4-sided frustum, which is represented as a convex combination of four corner rays intersecting at the apex. Each frustum has a near plane, four side planes, and a far plane. The near plane and the four side planes of a frustum are fixed and the far plane is parallel to the near plane. The depth of the far plane from the view point is updated as the algorithm computes a new blocker for a frustum. Our algorithm sub-divides Φ into smaller frusta using uniform or adaptive subdivision and computes a PVS for each frustum. Eventually, we take the union of these different PVSs to compute a PVS for Φ.

Algorithm: The algorithm computes the PVS for each frustum independently. We initialize the far plane associated with a frustum to be at infinity and update its value if any connected blocker is found. The algorithm traverses the BVH to efficiently compute the triangles that potentially intersect with the frustum. We perform fast Plücker intersection tests between the frustum and a triangle to determine if the frustum is completely inside, completely outside, or partially intersecting the triangle. If the frustum is partially intersecting, we reuse the Plücker test from the frustum-triangle intersection step to quickly find the edges that intersect the frustum (see Section 3.3). We perform frustum-triangle intersection with the neighboring triangles that are incident to these edges. We repeat this step of finding edges that intersect with the frustum and performing intersection tests with the triangles incident to the edge till the frustum is completely blocked by some set of triangles. If a blocker is found (see Section 3.2), we update the far plane depth of the frustum. Triangles beyond the far plane of the frustum are discarded from the PVS. If there is no blocker associated with the frustum, then all the triangles overlapping with the frustum belong to the PVS. Additionally, we compute the PVS for each frustum in parallel as described in Section 3.5.

3.2. Frustum Blocker Computation

We define a blocker for a frustum as a set of connected triangles such that every ray inside the frustum hits some triangle in the frustum blocker (see Figure 3(a)). When we intersect a frustum with a triangle, the frustum could partially intersect the triangle. In such a case, we walk to the neighboring triangles based on that intersection and try to find a blocker for the frustum (see Figure 3). We compute all the edges of the triangle that intersect with the frustum. For every intersecting edge, we walk to the neighboring triangle incident to the edge and perform frustum-triangle intersection test with the neighbor triangle.

The intersection and walking steps are repeated until one of the following conditions is satisfied:

a All triangles incident to every intersecting edge found during the frustum blocker step have been processed. This implies that we have found a blocker.

b A *free-edge*, i.e. an edge with only one incident triangle, or a *silhouette edge*, i.e. an edge with incident triangles facing in opposite directions as seen from the viewpoint, intersects with the frustum. In that case, we conclude that the current set of intersecting triangles does not constitute a blocker.

Note that our termination condition (b) for blocker computation is conservative. It is possible that there may exist a frustum blocker with a silhouette edge, but we need to perform additional computations to identify such blockers [NRJS03, Lai06]. In this case, we opt for simplicity, and rather search for some other blocker defined by a possibly

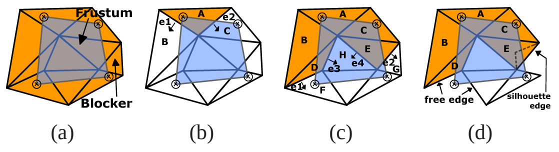

Figure 3: ***Frustum Blocker Computation****: (a) Example of a blocker with connected triangles. (b)-(c) Intersection and Walking: Identify intersecting edges (e1, e2, e3, and e4) and walk to the adjacent triangles (denoted by arrows from edge to the triangle). (d) Abort frustum blocker computation if a free-edge or a silhouette-edge is found.*

different set of triangles. Or we subdivide the frustum and the current object will become a blocker for a smaller sub-frustum.

If we terminate the traversal test due to condition (a), we have successfully found a frustum blocker. All triangles in the frustum blocker are marked visible and the far plane depth associated with the frustum is updated. Note that the depth of the far plane of the frustum is chosen such that all triangles in the frustum blocker lie in front of the far plane. If we terminate due to condition (b), then the algorithm cannot guarantee the existence of a frustum blocker. All triangles processed during this step are still marked visible but the far plane depth is not updated.

3.3. Frustum Intersection Tests

A key component of the algorithm is performing the intersection tests of the scene primitives with a frustum. The algorithm traverses the BVH and performs intersection tests between a frustum and the AABBs associated with the BVH. We use the technique proposed by Reshetov et al. [RSH05] to perform fast intersection tests between a frustum and an AABB. For every leaf node of the hierarchy we perform the intersection test with the frustum and triangle(s) associated with that leaf node. In order to perform the intersection test efficiently, we represent the corner rays of a frustum and the oriented edges of the triangle using Plücker coordinates [Sho98]. The orientation of a ray as seen along the edges of a triangle governs the intersection status of the ray with the triangle (see Figure 4(a)). Similarly, the orientation of four corner rays of the frustum as seen along the edges of a triangle governs the intersection status of the frustum with the triangle. We can determine with object-precision accuracy whether the frustum lies completely inside the triangle, completely outside the triangle, or partially intersects the triangle [CLT*08].

In practice, the Plücker test is conservative and it can wrongly classify a frustum to be partially intersecting a triangle even if the frustum is completely outside the triangle (as shown in Figure 4(b)). This can affect the correctness of our algorithm as we may wrongly classify an object as a blocker due to these conservative intersection tests. We make

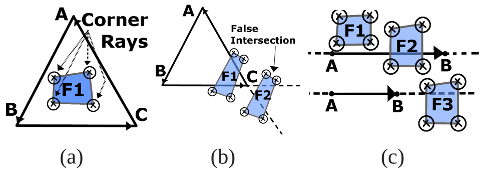

Figure 4: ***Conservative Plücker Tests****: (a) All four corner rays of the frustum F_1 have the same orientation as seen along every directed edge of the triangle ABC. Thus, F_1 is completely-inside ABC. (b) Intersection between a frustum and a triangle can be conservative. F_2 will be classified as partially intersecting. (c) Different cases of frustum-edge intersections: F_1 does not intersect the edge AB, F_2 intersects AB. F_3 is falsely classified as intersecting with AB.*

sure that atleast one of the corner rays is inside the blocker to avoid such cases.

Frustum-Edge Intersection: When a frustum partially intersects with a triangle, we can quickly determine which edges of the triangle intersect the frustum. We reuse the Plücker test between the frustum and the triangle to find the edges of the triangle that intersect the frustum. As shown in Figure 4(c), a frustum intersects with an edge if all four corner rays of the frustum do not have the same orientation as seen along the edge. This test may falsely classify an edge as intersecting even if the frustum does not intersect the edge, as shown in Figure 4(c) and thereby make our algorithm conservative. This test is also used in Section 3.2 to compute a set of triangles that may block the frustum completely.

Far Plane Depth Update: The far plane associated with a frustum is updated whenever a blocker is found. The blocker may correspond to a single triangle or multiple triangles. If a frustum lies completely inside a triangle, the triangle blocks the frustum. We, therefore, mark the triangle as visible and update the depth of the far plane of the frustum as shown in Figure 5(a). The frustum intersects the triangle at points $\mathbf{H_1}$ and $\mathbf{H_2}$, and $\mathbf{d_1}$ and $\mathbf{d_2}$ are the projected distances of $|V\mathbf{H_1}|$ and $|V\mathbf{H_2}|$ along the near plane normal. We set the far plane depth of the frustum as the maximum of the projected distances. In other cases, the blocker may be composed of multiple triangles. We update the far plane depth of the frustum as shown in Figure 5(b). We compute the far plane depth for every triangle in the frustum blocker, assuming the frustum is completely inside the triangle. In Figure 5(b), d and d' are the far plane depths for triangles T_1 and T_1', respectively, of the frustum blocker. The far plane depth of the frustum is set to the maximum of far plane depths computed for the triangles in the frustum blocker, which is d' in this case.

3.4. Frustum Subdivision

Our algorithm implicitly assumes that the size of connected blockers is larger that the cross-section of the frusta. The simplest algorithm subdivides a frustum in a uniform man-

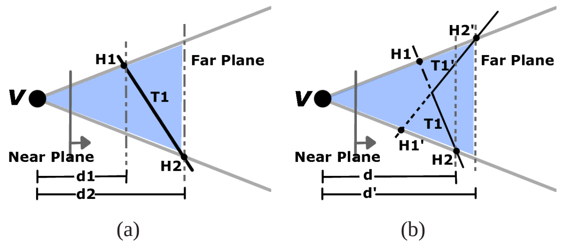

Figure 5: ***Updating Far Plane Depth**: (a) Frustum lies completely inside triangle T_1. The depth of the far plane is set to the maximum of $\mathbf{d_1}$ and $\mathbf{d_2}$. (b) Triangles T_1 and T_1' constitute the blocker. We compute the far plane depths of each triangle and use the maximum of the depth values.*

ner. This approach is simpler to implement and also simpler to parallelize on current multi-core and many-core architectures, in terms of load balancing. However, many complex models (e.g. CAD models) have a non-uniform distribution of primitives in 3D. In that case, it may be more useful to perform adaptive subdivision of the frusta. In that case, we use the AD-FRUSTUM representation [CLT*08], which uses a quadtree data structure. We use the following criteria to perform subdivision. If the PVS associated with a frustum is large, we recursively compute the PVS associated with each sub-frustum. Whenever the algorithm only computes a partial blocker of connected triangles using the intersection tests, we estimate its cross-section area and use that area to compute the sub-frusta. There are other known techniques to estimate the distribution of primitives [WWZ*06] and they can be used to guide the subdivision. As compared to uniform subdivision, adaptive techniques reduce the total number of frusta traced for PVS computation [CLT*08]. Moreover, we use spatial coherence to reduce the number of intersection tests between the parent and child frusta.

3.5. Many-core Implementation

Our algorithm is highly parallelizable as the PVS for each frustum can be computed independently. However, the union of these different PVSs have to be performed in a thread safe manner. This can be done by maintaining an array of bits, one bit per triangle, and marking a bit visible only when the corresponding triangle is found visible. The bits are reset only once at the start of the algorithm. In this case the time to query if a triangle is visible is $O(1)$ but enumerating the visible triangles is $O(N)$, where N is the number of triangles in the scene. To improve upon this we maintain a per thread hash map to compute PVS per thread. The PVS per thread is combined in the end to compute the final PVS. The average time to query if a triangle is visible is $O(1)$ and the time to enumerate the visible triangles is $O(K)$, where K is the number of visible triangles.

Model		PVS	PVS	Time
Name	Tris	Ratio	Size	(ms)
Armadillo	345K	1.16	98K	30
Blade	1.8M	1.05	190K	90
Thai	10M	1.06	210K	66
SodaHall	1.5M	1.22	2.1K	15
PowerPlant	12M	1.25	15K	130
Flamenco	40K	1.11	7K	16

Table 1: *Performance Results: From-point conservative visibility computation of models with varying complexity (CAD models, scanned models and dynamic scenes). All the timings were computed on a 16-core 64-bit Intel X7350@2.93 GHz. The PVS ratio provides a measures of how conservative is the computed PVS with respect to exact visibility.*

4. Results and Analysis

In this section, we present our results on from-point conservative visibility (Section 4.1). We also compare our approach with prior visibility computation algorithms. Our results were generated on a workstation with 16-core, 64-bit Intel X7350@2.93 GHz processors. We generate timings by using different number of cores $(1-16)$ for visibility computations and sound propagation. We also use SSE instructions to accelerate frustum intersection tests and OpenMP to parallelize on multiple cores.

4.1. Visibility Results

We demonstrate our results on computing from-point object space conservative PVS on a variety of models ranging from simple models (like soda hall, armadillo, blade) to complex models (like power plant and thai statue) to a dynamic model (flamenco animation). These models are shown in Figure 6. Our results are summarized in Table 1. We are not aware of any prior method that can compute the exact visible set on these complex models. Therefore, we compute an approximation to π_{exact} by shooting frusta at $4K \times 4K$ resolution. The *PVS-ratio* refers to: (size of PVS) / (size of π_{exact}), and is a measure of how conservative is the computed PVS. In all benchmarks, we are able to compute a conservative approximation to the PVS at interactive rates on multi-core PC. The frame sequences used for generating average results are shown in accompanying video. Further, we show that our approach converges well to π_{exact} as we shoot higher number of frusta (see Figure 7). Detailed results on convergence for each model are provided in the Appendix. Also, our approach maps well on multi-core architectures. We observe linear scaling in performance as the number of cores increase (see Figure 8).

4.2. Analysis

We analyze our algorithm and compare it with prior techniques. The accuracy of our algorithm is governed by the

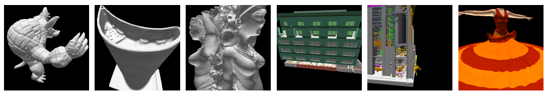

Figure 6: *Benchmarks: Left to right: (a) Armadillo (345K triangles). (b) Blade (1.8M triangles). (c) Thai Statue (10M triangles). (d) Soda Hall (1.5M triangles). (e) PowerPlant (12M triangles). (f) Flamenco (40K dynamic scene).*

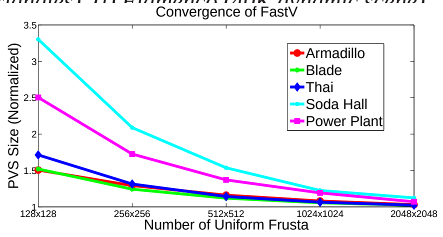

Figure 7: *PVS ratio vs. #Frusta: As the number of frusta increase, the PVS computed by our approach converges to π_{exact}. This graph shows the rate of convergence for different benchmarks. The CAD models have a higher ratio as compared to scanned models.*

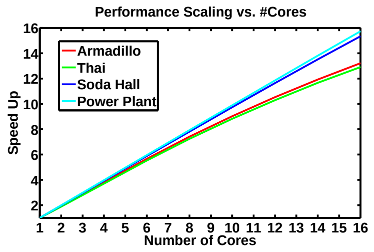

Figure 8: *Performance scaling vs. #Cores: The performance of our system scales linearly with the #cores. We have benchmarked our system on upto 16 cores.*

accuracy of the intersection tests, which exploit the IEEE floating-point hardware. Our approach is robust and general, and not prone to any degeneracies.

Conservative approach: We compute a conservative PVS for every frustum. This follows from our basic approach to compute the blockers and far planes for each frustum. In practice, our approach can be overly conservative in some cases. The underlying blocker computation algorithm is conservative. Moreover, we don't consider the case when the union of two or more objects can serve as a blocker. This is shown in Figure 2) with two disjoint occluders, V_1 and V_2. Instead of using more sophisticated algorithms for blocker computation, we found it cheaper to subdivide the frustum into sub-frusta and compute blockers for them. As a result, we can make our approach less conservative by using more frusta and the PVS (π) converges to π_{exact} (see Figure 7).

Model connectivity and triangle soup models: Our algorithm exploits the connectivity information in the model to compute the blockers, which are formed using connected triangles. If the connectivity information is not available, then the algorithm would subdivide the frustum such that each blocker would consist of only one triangle.

4.3. Comparisons

Our approach performs volumetric tracing, which is similar to *beam tracing*. However, we don't perform exact clipping operations to compute an exact representation of the visible surface. Rather we only estimate the triangles belonging to the PVS by identifying the blockers for each frustum. Beam tracing algorithms can also be accelerated by using spatial data structures [FCE*98, LSLS09], but they have mostly been applied to scenes with large occluders (e.g. architectural models). Recently, Overbeck et al. [ORM07] presented a fast beam tracing algorithm that is based on spatial hierarchies. We performed a preliminary comparison with an implementation of this beam tracing algorithm with FastV. We chose multiple key frames from the armadillo model sequence (see Video) and compared the PVS computed by FastV algorithm (with $4K \times 4K$ uniform frusta) with the PVS computed by the beam tracer. We observed that FastV's PVS converges to within $1-10\%$ of the exact from-point beam tracing PVS (see Appendix). Furthermore, FastV appears to be more robust than the beam tracing solution as the beams may leak between the triangles due to numerical issues (see Video). It is not clear whether current beam tracing algorithms can robustly handle complex models like the powerplant. In terms of performance, FastV is about $5-8$ times faster on a single core on the armadillo model, as compared to [ORM07]. Moreover, FastV is relatively easier to parallelize on multi-core architectures.

Most of the prior object space conservative visibility culling algorithms are designed for scenes with large occluders [BHS98, CT97, HMC*97, LG95]. These algorithms can work well on special types of models, e.g. architectural models represented using cells and portals or urban scenes. In contrast, our approach is mainly designed for general 3D models and doesn't make any assumption about large occluders. It is possible to perform conservative rasterization using current GPUs [AMA05]. However, it has the overhead of rendering additional triangles and CPU-GPU communication latency.

It is hard to make a direct comparison with image space approaches because of their accuracy. In practice, image space approaches can exploit the rasterization hardware or fast ray-tracing techniques [RSH05] and would be faster than FastV. Moreover, image space approaches also perform occluder fusion and in some cases may compute a smaller set of visible primitives than FastV. However, the main issue with the image space approaches is deriving any tight bounds on the accuracy of the result. This is highlighted in the appendix, where we used ray tracing to approximate the visible primitives. In complex models like the powerplant, we need a sampling resolution of at least $32K \times 32K$ to compute a good approximation of the visible primitives. At lower resolutions, the visible set computed by the ray tracing algorithm doesn't seem to converge well.

5. Geometric Sound Propagation

In this section, we describe our geometric sound propagation algorithm based on FastV. Given a point sound source, the CAD model with material properties (i.e. the acoustic space), and the receiver position, the goal is to compute the impulse response (IR) of the acoustic space. Later the IRs are convolved with an audio signal to reproduce the sound. We use our PVS computation algorithm described above for fast image-source computation that only takes into account specular reflections [AB79, FCE*98, LSLS09]. In practice, this approach is only accurate for high frequency sources.

Each image source radiates in free space and considers secondary sources generated by mirroring the location of the input source over each boundary element in the environment. For each secondary source, the specular reflection path can be computed by performing repeated intersections of a line segment from the source position to the position of the receiver. In order to accurately compute all propagation paths, the algorithm creates image-sources (secondary sources) for every polygon in the scene. This step is repeated for all the secondary sources upto some user specified (say k) orders of reflection. Clearly, the number of image sources are $O(N^{k+1})$, where N is the number of triangles in the scene. This can become expensive for complex models.

We use our PVS computation algorithm to accelerate the computation for complex scenes. We use a two stage algorithm. In the first stage, we use our conservative visibility culling algorithm and compute all the secondary image sources up to the specified orders of reflection. Since we overestimate the set of visibility triangles, we use the second stage to perform a validation step. For the first stage, we use a variant of Laine et al.'s [LSLS09] algorithm and only compute the secondary image-sources for the triangles that are visible from the source. Specifically, we shoot primary frusta from the sound source. For every primary frustum we compute its PVS. We then reflect the primary frustum against all visible triangles to create secondary frusta, which is similar to creating image-sources for visible triangles. This step is repeated for secondary frusta upto k orders of reflection. In the second stage, we construct paths from the listener to the sound source for all the frusta which reach the listener. As our approach is conservative, we have to ensure that this path is a valid path. To validate the path, we intersect each segment of the path with the scene geometry and if an intersection is found the path is discarded.

5.1. Results

We present our results on accurate geometric sound propagation in this section. Table 2 summarizes our results. We

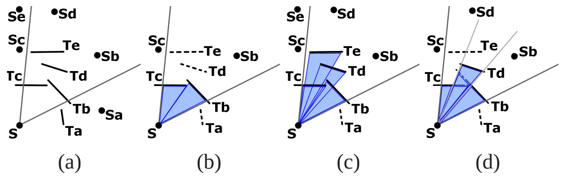

Figure 9: *Geometric sound propagation approaches: Given a sound source, S, and triangles (a, b, c, d, and e) the image source method (see 9a) creates image-sources of S against all primitives in the scene. Beam tracing method (see 9b) computes image-sources for only exactly visible triangles, b and c in this case. Accelerated beam tracing approach computes image-sources for all triangles inside the beam volume (see 9c), i.e., b, c, d, and e in this case. Our implementation (see 9d) computes image-sources for triangles b, c, and d in the PVS as computed according to the technique described in previous sections.*

perform geometric sound propagation on models of varying complexity from 438 triangles to 212K triangles. We use three benchmarks presented in accelerated beam tracing (ABT) algorithm [LSLS09]. We also used two additional complex benchmarks with 80K and 212K triangles. We are not aware of any implementation of accurate geometric propagation that can handle models of such complexity.

Model	Tris	Time (sec)	Speed Up (ABT)
Simple Room	438	.16	10.1X
Regular Room	1190	.93	22.2X
Complex Room	5635	6.5	11.8X
Sibenik	78.2K	72.0	–
Trade Show	212K	217.6	–

Table 2: *Accurate sound propagation: The performance of sound propagation algorithms for three orders of reflection on a single core. We observe $10-20X$ speedup on the simple models over accelerated beam tracing (ABT) [LSLS09].*

5.2. Comparison with Prior Approaches

Most accurate geometric acoustic methods can be described as variants of the image-source method. Figure 9 compares different accurate geometric sound propagation methods. The main difference between these methods is in terms of which image-sources they choose to compute [FCE*98, LSLS09]. A naïve image source method computes image sources against all triangles in the scene [AB79]. Beam tracing methods compute the image-sources for exactly visible triangles from a source. Methods based on beam tracing, like accelerated beam tracing [LSLS09], computes image-sources for every triangle inside the beam volume. Our approach, shown in Figure 9(d), finds the conservative PVS from a source and computes image-sources for the triangles in the conservative PVS. Thus, for a given model our approach considers more image-sources compared to beam tracing. It is an efficient compromise between the expensive step to compute exactly visible triangles in beam tracing vs. computing extra image-sources in accelerated beam tracing. We observe $10-20$X speedups over prior accurate geometric sound propagation algorithms. Chandak et al. [CLT*08] also used adaptive frustum tracing for geometric sound propagation. However, they perform discrete clipping and therefore, it is hard to derive good bounds on its accuracy.

6. Limitations and Conclusions

Our approach has some limitations. Since we don't perform occluder fusion, the PVS computed by our algorithm can be overly conservative sometimes. If the scene has no big occluders, we may need to trace a large number of frusta. Our intersection tests are fast, but the conservative nature of the blocker computation can result in a large PVS. The model and its hierarchy are stored in main memory, and therefore our approach is limited to in-core models. Our algorithm is easy to parallelize and works quite well, but is still slower than image space approaches that perform coherent ray tracing or use GPU rasterization capabilities.

Conclusions: We present a fast and simple visibility culling algorithm and demonstrate its performance on complex models. The algorithm is general and works well on complex 3D models. To the best of our knowledge, this is the first from-point object space visibility algorithm that can handle complex 3D models with millions of triangles at almost interactive rates.

Future Work: There are many avenues for future work. We will like to implement the algorithm on a many-core GPU or upcoming Larrabee processor to further exploit the high parallel performance of commodity processors. This could provide capability to design more accurate rendering algorithms based on object-precision visibility computations on complex models (e.g. shadow generation). We will also like to evaluate the trade-offs of using more sophisticated blocker computation algorithms [NRJS03, Lai06]. In terms of sound propagation, our approach can be extended to compute edge diffraction based on uniform theory of diffraction (UTD).

References

[AB79] ALLEN J. B., BERKLEY D. A.: Image method for efficiently simulating small-room acoustics. *The Journal of the Acoustical Society of America 65*, 4 (April 1979), 943–950.

[AMA05] AKENINE-MÖLLER T., AILA T.: Conservative and tiled rasterization using a modified triangle set-up. *journal of graphics tools 10*, 3 (2005), 1–8.

[BHS98] BITTNER J., HAVRAN V., SLAVIK P.: Hierarchical visibility culling with occlusion trees. *Computer Graphics International, 1998. Proceedings* (Jun 1998), 207–219.

[BW05] BITTNER J., WONKA P.: Fast exact from-region visibility in urban scenes. *Eurographics Symposium on Rendering* (2005), 223–230.

[CLT*08] CHANDAK A., LAUTERBACH C., TAYLOR M., REN Z., MANOCHA D.: Ad-frustum: Adaptive frustum tracing for interactive sound propagation. In *Proc. IEEE Visualization* (2008).

[COCSD03] COHEN-OR D., CHRYSANTHOU Y., SILVA C., DURAND F.: A survey of visibility for walkthrough applications. *Visualization and Computer Graphics, IEEE Transactions on 9*, 3 (July-Sept. 2003), 412–431.

[CT97] COORG S., TELLER S.: Real-time occlusion culling for models with large occluders. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics* (New York, NY, USA, April 1997), ACM, pp. 83–ff.

[DD02] DUGUET F., DRETTAKIS G.: Robust epsilon visibility. *Proc. of ACM SIGGRAPH* (2002), 567–575.

[Dur99] DURAND F.: *3D Visibility, Analysis and Applications*. PhD thesis, U. Joseph Fourier, 1999.

[FCE*98] FUNKHOUSER T., CARLBOM I., ELKO G., PINGALI G., SONDHI M., WEST J.: A beam tracing approach to acoustic modeling for interactive virtual environments. In *Proc. of ACM SIGGRAPH* (1998), pp. 21–32.

[Gha01] GHALI S.: A survey of practical object space visibility algorithms. In *SIGGRAPH Tutorial Notes* (2001).

[HH84] HECKBERT P. S., HANRAHAN P.: Beam tracing polygonal objects. In *Proc. of ACM SIGGRAPH* (1984), pp. 119–127.

[HMC*97] HUDSON T., MANOCHA D., COHEN J., LIN M., HOFF K., ZHANG H.: Accelerated occlusion culling using shadow frusta. In *Proc. of ACM Symposium on Computational Geometry* (1997), pp. 1–10.

[KCCO00] KOLTUN V., CHRYSANTHOU Y., COHEN-OR D.: Virtual occluders: An efficient intermediate pvs representation. In *Eurpographics Workshop on Rendering* (2000), pp. 59–70.

[KS00] KLOSOWSKI J., SILVA C.: The prioritized-layered projection algorithm for visible set estimation. *IEEE Trans. on Visualization and Computer Graphics 6*, 2 (2000), 108–123.

[Lai06] LAINE S.: *An Incremental Shaft Subdivision Algorithm for Computing Shadows and Visibility*. Master's thesis, Helsinki University of Technology, March 2006.

[Len93] LENHERT H.: Systematic errors of the ray-tracing algoirthm. *Applied Acoustics 38* (1993), 207–221.

[LG95] LUEBKE D., GEORGES C.: Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *ACM Interactive 3D Graphics Conference* (Monterey, CA, 1995), pp. 105–108.

[LSCO03] LEYVAND T., SORKINE O., , COHEN-OR D.: Ray space factorization for from-region visibility. *Proc. of ACM SIGGRAPH* (2003), 595–604.

[LSLS09] LAINE S., SILTANEN S., LOKKI T., SAVIOJA L.: Accelerated beam tracing algorithm. *Applied Acoustic 70*, 1 (2009), 172–181.

[MBW08] MATTAUSCH O., BITTNER J., WIMMER M.: Chc++: Coherent hierarchical culling revisted. *Proc. of Eurographics* (2008), 221–230.

[NB04] NIRENSTEIN S., BLAKE E.: Hardware accelerated visibility preprocessing using adaptive sampling. In *Eurographics Workshop on Rendering* (2004).

[Nir03] NIRENSTEIN S.: *Fast and Accurate Visibility Preprocessing*. PhD thesis, University of Cape Town, South Africa, 2003.

[NRJS03] NAVAZO I., ROSSIGNAC J., JOU J., SHARIF R.: Shieldtester: Cell-to-cell visibility test for surface occluderis. In *Proc. of Eurographics* (2003), pp. 291–302.

[ORM07] OVERBECK R., RAMAMOORTHI R., MARK W. R.: A Real-time Beam Tracer with Application to Exact Soft Shadows. In *Eurographics Symposium on Rendering* (Jun 2007), pp. 85–98.

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. *ACM Trans. Graph. 24*, 3 (2005), 1176–1185.

[Sho98] SHOEMAKE K.: Pluecker coordinate tutorial. *Ray Tracing News 11*, 1 (1998).

[Tel92] TELLER S. J.: *Visibility Computations in Densely Occluded Polyheral Environments*. PhD thesis, CS Division, UC Berkeley, 1992.

[WWZ*06] WONKA P., WIMMER M., ZHOU K., MAIERHOFER S., HESINA G., RESHETOV A.: Guided visibility sampling. *Proc. of ACM SIGGRAPH* (2006), 494–502.