Trim Regions for Online Computation of From-Region Potentially Visible Sets

PHILIP VOGLREITER, Graz University of Technology, Austria and VRVis Forschungs GmbH, Austria
BERNHARD KERBL, TU Wien, Austria and Inria, Université Côte d’Azur, France
ALEXANDER WEINRAUCH, Graz University of Technology, Austria
JOERG H. MUELLER, Graz University of Technology, Austria
THOMAS NEFF, Graz University of Technology, Austria
MARKUS STEINBERGER, Graz University of Technology, Austria
DIETER SCHMALSTIEG, Graz University of Technology, Austria

Fig. 1. (a) The view in the inset is seen by an observer standing at the location indicated by the inset’s leader line. Our method computes a potentially visible set (PVS) corresponding to a viewcell (region) of a given radius around the viewpoint in real time. The six images each show a PVS for view cell sizes of 5-30 cm. The truly visible part of the scene is shown shaded, while false positives are shown in blue and the remaining scene is shown in grey. No false negatives are visible. (b) Note how the width of the visible “corridor” on the floor progressively expands with the viewcell size. (c) The base of the crane, which was a false positive of 5 to 15 cm, becomes a true part of the PVS at 20 cm and above. It is typical that false positives become true positives as the viewcell size expands, since they are “almost visible” when first observed.

For visibility computation, a from-region potentially visible set (PVS) is an established tool in rendering acceleration, but its high computational cost means that a from-region PVS is almost always precomputed. Precomputation restricts the use of PVS to static scenes and leads to high storage cost, in particular, if we need fine-grained regions. For dynamic applications, such as streaming content over a variable-bandwidth network, online PVS computation with configurable region size is required. We address this need with trim regions, a new method for generating from-region PVS for arbitrary scenes in real time. Trim regions perform controlled erosion of object silhouettes in image space, implicitly applying the shrinking theorem known from previous work. Our algorithm is the first that applies automatic shrinking to unconstrained 3D scenes, including non-manifold meshes, and does so in real time using an efficient GPU execution model. We demonstrate that our algorithm generates a tight PVS for complex scenes and outperforms previous online methods for from-viewpoint and from-region PVS. It runs at 60 Hz for realistic game scenes consisting of millions of triangles and computes PVS with tightness matching or surpassing existing approaches.

CCS Concepts:
• Computing methodologies → Visibility.

ACM Reference Format:

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author’s version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in ACM Transactions on Graphics, https://doi.org/10.1145/3592434.
1 INTRODUCTION
Visibility computation is fundamental to computer graphics. In real-time rendering, view frustum culling is often followed by occlusion culling, which attempts to efficiently eliminate as many invisible portions of the scene as possible, before the remainder is submitted to expensive shading. Occlusion culling usually relies on computing a potentially visible set (PVS), i.e., a superset of the exact visible set (EVS). A PVS is preferred over an EVS, because a PVS can be determined more efficiently [Airey et al. 1990].

Existing PVS methods fall into one of two major categories [Cohen-Or et al. 2003]: from-point PVS methods, which are often used in game engines or architectural preview, compute visibility for a given viewpoint in every frame. In contrast, from-region PVS methods partition the space of allowed camera poses into viewcells and compute visibility that is valid for any viewpoint inside the viewcell. Since from-region PVS computation is a 4D problem [Durand 1999], it is usually performed offline, i.e., in a precomputation step.

In contrast, we are interested in online from-region PVS methods, since they facilitate novel forms of remote rendering [Shi and Hsu 2015], where a cloud or edge server streams shaded geometry just in time to a lightweight client, such as a set-top box or a wireless virtual reality (VR) headset [Hladky et al. 2019b; Mueller et al. 2018]. The client only needs an inexpensive and energy-efficient fixed-function pipeline to render textured polygons and can apply latency compensation by using the latest user input to control the viewpoint. Such latency concealment is particularly important for VR.

Such a scenario benefits from the fact that, depending on the maximum speed of user motion, a from-region PVS remains valid for a certain timespan [Wonka et al. 2001]. Within this timespan, the rendering engine can predict which objects may become visible. It can make preparations to render those objects, preload them from nonvolatile storage or transmit them over the network without noticeable latency to the user. If we want to support streaming of dynamic, animated scenes, such as for computer games, scientific simulation, or telepresence, this implies that the from-region PVS must be computed online. Because of the high computational complexity, online computation has hardly been attempted.

An early notable exception is instant visibility [Wonka et al. 2001]. Like our method, it builds on the idea of occluder shrinking proposed by Wonka et al. [2000]. However, the original publication demonstrated this idea only for 2.5D city scenes, exploiting the fact that facades can be used as large convex occluders and allow for straightforward 2D geometric shrinking. Unfortunately, applying the same occluder shrinking to general 3D objects requires a three-dimensional erosion, which is difficult to compute geometrically for arbitrary scenes and viewcells [Décoret et al. 2003].

We show that occluder shrinking for general 3D objects can be computed with a rasterization pipeline, which makes it simple and efficient. Our method has several novel contributions:

1. We present a visibility method based entirely on rasterization for computing occluder shrinking for arbitrary polygonal scenes. Our method imposes very few restrictions on the scene geometry. We do not require manifold geometry or watertight meshes and even support polygon soups and instanced rendering. The only requirement is the ability to identify triangle neighborhoods via shared edges.

2. We extend occluder shrinking to support self-occlusion and occludee shrinking. Our results reveal that these capabilities, which have not been present in previous work, significantly increase the tightness of the potentially visible set.

3. We explain how to combine our visibility method with a layer-by-layer traversal of a scene octree to create a scalable from-region PVS method that runs at interactive frame rates. The octree itself can be created in seconds upon loading the scene, and no further precomputation is required.

To the best of our knowledge, we present the first and only system that computes a from-region PVS on non-manifold 3D scenes in real time. We demonstrate how our method performs favorably on complex scenes with millions of polygons at 50–60 Hz. It outperforms previous approaches with respect to runtime and PVS tightness. Moreover, our method is able to operate under tight real-time constraints, as required by streaming applications.

2 BACKGROUND
As background for our method, we describe previous work on visibility, organized into from-point and from-region approaches.

2.1 From-point visibility
From-point methods must run online to be useful, as the exact viewpoint is only known at the beginning of a new frame [Bittner and Wonka 2003; Cohen-Or et al. 2003]. Consequently, the visibility stage must run synchronously with the rest of the rendering pipeline, usually scheduled after view frustum culling. Such a coarse-to-fine pipeline consisting of frustum culling, followed by occlusion culling and, finally, shading, is typical for deferred rendering engines. In deferred rendering, a from-point EVS is determined at image-space precision by rasterizing into a visibility buffer [Burns and Hunt 2013] with depth and primitive id attachments.

If scenes are large enough, it can pay off to run an inexpensive from-point PVS method that removes occluded parts of the scene quickly, ideally in large chunks at meshlet-level or object-level precision. As the final visibility per pixel can always be resolved with a
regular depth buffer, the tightness of the PVS is of primary interest for the choice of visibility algorithm.

Most visibility algorithms work progressively by selecting a potential occluder and removing the parts of the scene it occludes, the occludees. Occluder selection in commercial rendering engines is usually greedy, picking occluders such that they cover the largest subset of remaining occludees. However, this can be suboptimal: Significant occlusion savings frequently emerge only from the interplay of many – often small – occluders. Therefore, it is vital to consider occluder fusion [Schaufler et al. 2000; Wonka et al. 2000].

A popular approach for occluder fusion is to aggregate occluders into an occluded volume, similar to a shadow volume. Occludees are tested for containment in the occluded volume. The occluded volume can be created as an explicit geometric structure, e.g. as a BSP tree [Bittner et al. 1998], shadow volume [Hudson et al. 1997] or bounding volume hierarchy [Chandak et al. 2009]. However, in from-viewpoint methods, it is usually more straightforward to rely on the depth buffer as an occlusion volume, especially if an existing depth buffer can be reused [Lee et al. 2018]. The depth buffer can be utilized as an occluded volume by creating a depth pyramid [Greene et al. 1993] or by using GPU occlusion queries [Bittner et al. 2004].

Alternatively, efficient occlusion culling can be facilitated by using virtual occluders, i.e., hallucinated objects that are fully contained in the occluded volume [Durand et al. 2000; Koltun et al. 2000; Schaufler et al. 2000]. Such virtual occluders are often custom-made [Persson 2012], which often complicates content generation and largely precludes exploitation of occluder fusion in dynamic scenes.

Apart from the supported occluder types, the efficiency of occlusion culling is strongly influenced by the execution environment: CPU, GPU, or a mixed CPU-GPU environment. Unfortunately, each of these cases faces inherent problems: CPU-only methods [Chandrasekaran et al. 2016; Collin 2011; Hasselgren et al. 2016], which rasterize depth on the GPU, suffer from limited pixel throughput, requiring overly coarse rasterization that may be prone to geometric aliasing. Mixed CPU-GPU methods suffer from CPU-GPU synchronization latency [Bittner et al. 2004], high CPU overhead for fine-grained draw calls [Mattausch et al. 2008; Serpa and Rodrigues 2019], or low bandwidth of GPU-to-CPU readbacks, at least on systems with a discrete GPU [Hill and Collin 2011]. GPU-only methods strive to coordinate all GPU tasks without relying on the CPU for synchronization. This approach has been restricted so far to scenes consisting of a large set of uniform meshes or instances [Haar and Aaltonen 2015; Shopf et al. 2008]. Our method falls into the GPU-only class, but avoids the typical drawbacks.

2.2 From-region visibility

Streaming applications always require from-region visibility, since the server cannot know the client’s exact new viewpoint in advance. From-region methods are computationally much more expensive than from-point methods, so running them online is generally avoided. During offline computation, the space of possible viewpoints can be subdivided into static viewcells, and a PVS per viewcell can be computed [Teller and Séquin 1991].

Fig. 3. The erosion theorem states that determining from-point visibility with respect to a shrunk occluder \( Q \) is equivalent to from-region visibility of the original occluder \( O \). Any ray emitted from within a viewcell \( V \) formed around \( v_0 \) cannot observe an occludee \( O' \), if \( O' \) is blocked by \( Q \) from \( v_0 \). Even the extremal point \( v_1 \) in \( V \) cannot see the extremal point \( x_1 \) of \( O' \).

Early from-region methods restrict the type of scene to 2.5D geometry [Wonka et al. 2000], require watertight manifold meshes [Schaufler et al. 2000] or need large known occluders [Cohen-Or et al. 1998]. All these assumptions impose rather severe limitations on the range of possible applications. Thus, later work has concentrated on supporting more general scenes [Bittner et al. 2009; Laine 2005]. Today, popular rendering engines, such as Unreal Engine, provide tools for pre-computing a from-region PVS [Epic Games 2022].

The challenging characteristics of from-region visibility come from its need to sample at least four dimensions: two for the viewpoint, usually assumed to lie in a plane or on a 2D manifold, and two for the ray direction. Existing strategies all have in common that they split the 4D domain into 2D sub-spaces, where simpler planar problems can be solved. A solution to the 4D PVS can then be constructed by logically combining the sub-space visibility: In order to be occluded in the PVS, it is necessary that an occludee be occluded in all of the sub-spaces. The most obvious choice of sub-space is to sample from-point EVS at multiple locations in the viewcell and create the union of all EVS instances.

If dense sampling is considered too expensive, sufficient accuracy can be ensured by a variety of strategies, including adaptive sampling [Bittner et al. 2009; Mattausch et al. 2006; Nirenstein and Blake 2004], sampling 2D slices of the ray-space [Bittner et al. 2005; Koltun et al. 2001; Leyvand et al. 2003], or imposing a restriction to 2.5D scenes [Décoret et al. 2003; Koltun et al. 2000; Wonka et al. 2000] or to voxelized scenes [Hong et al. 1997; Schaufler et al. 2000].

Even with these optimizations, computation costs tend to be very high, and the PVS is usually precomputed. Alas, offline computation requires a lot of memory for storing results [Freitag et al. 2017] and cannot handle dynamic scenes, in particular, if we need fine-grained regions or if we want to adaptively vary the viewcell size.

Online PVS computation overcomes these problems and lends itself to streaming [Hladky et al. 2019a; Mueller et al. 2018]. First, only the PVS for the current viewcell needs to be stored in memory. Second, we can choose the viewcell’s size and shape based on current rendering performance, network bandwidth or the user’s speed [Wonka et al. 2001]. Third, a viewcell with a shape extrapolated from the current viewpoint allows ahead-of-time selection of relevant portions of the scene and timely delivery for rendering [Correa et al. 2003]. Hladky et al. [2019a] investigate the set
We first lay out the fundamental principles of object erosion with \( \Delta \) (Section 3.1) and investigate disocclusions (Section 3.2) around object silhouettes as a result of camera translations. This forms the basis of 3D occluder erosion using trim regions (Section 3.3). Next, we investigate the shape of the viewcell (Section 3.4) and derive how to correctly handle multiple consecutive occlu- dces along a line of sight (Section 3.5). We also discuss how to determine erosion of an entire polygonal object based on finding the optimal direction (Section 3.6) for each trim region along an occluder silhouette.

3 TRIM REGIONS

We first lay out the fundamental principles of object erosion with trim regions. We start by summarizing the proof of Wonka et al. [2000]. Assume that a ray segment from a viewpoint \( v_0 \) to a surface point \( x_0 \) is blocked by an occluder \( O \) (Figure 3). Any ray towards \( x_0 \) starting at a viewpoint \( v_1 \) taken from a spherical viewcell \( V(v_0, \Delta) = \{v_1 \text{ s.t. } |v_0 - v_1| < \Delta\} \) passes \( O \) at a distance smaller than \( \Delta \). This occlusion relationship is invertible: One can determine a shrunk occluder \( Q \) by eroding \( O \) with \( V(o, \Delta) \forall o \in \text{boundary}(O) \). If \( Q \) still blocks the line from \( v_0 \) to \( x_0 \) (see mark \( y_0 \) in Figure 3), then \( O \) blocks any ray segment from \( v_1 \) to \( x_0 \) (see mark \( y_1 \)). Hence, we have identified an occlusion of \( x_0 \) from \( V(v_0, \Delta) \).

Décoret et al. [2003] further show that this occlusion actually holds for all occludees \( x_1 \in V(x_0, \Delta) \) as well (see mark \( y_2 \)). This observation implies that occludees can also be shrunk, reducing an occlusion test for a shaft connecting points in \( V(v_0, \Delta) \) to points in \( V(x_0, \Delta) \) to a line-segment-only occlusion test from \( v_0 \) to \( x_0 \).

Applying occluder shrinking in practice requires computing a 3D erosion of an arbitrary polygonal object. But if we only want to compute occlusion from a compact viewcell, we can avoid the need to erode an entire object by concentrating only on its silhouette, where much simpler local “trimming” operations suffice.

To that aim, we define occlusion intervals (Section 3.1) and investigate disocclusions (Section 3.2) around object silhouettes as a result of camera translations. This forms the basis of 3D occluder erosion using trim regions (Section 3.3). Next, we investigate the shape of the viewcell (Section 3.4) and derive how to correctly handle multiple consecutive occludees along a line of sight (Section 3.5). We also discuss how to determine erosion of an entire polygonal object based on finding the optimal direction (Section 3.6) for each trim region along an occluder silhouette.

3.1 Occlusion intervals

Let us first define the concept of occlusion in our terms. In a classic pinhole camera model, view rays emanate from a camera at position \( v_0 \) and interact with a scene. Conceptually, each pixel targeted in rasterization corresponds to a view ray that extends to the far plane. If the view ray intersects a solid object along its path, we call this event an occlusion, since the object blocks visibility of any other objects further along the ray. Depth buffering retains the closest hit along each view ray as the visible primitive for the given pixel.

We define an occlusion interval as a segment along the view ray between the entrance and exit points of the ray with respect to an occluder object. In Figure 4, left, an occluder blocks a view ray from \( v_0 \) towards \( v_1 \). The ray observes an occlusion interval between \( f \) and \( b \). For a view ray that grazes the silhouette \( s \), the occlusion interval collapses into a single point at \( s \).

Note that the use of a far plane at finite distance is a significant difference in formulation from the original occluder shrinking paper [Wonka et al. 2000], which assumes the far plane at infinity. Their assumption is overly conservative, since every practical scene

Fig. 4. (left) A camera located at position \( v_0 \) looks at an object. A view ray extending towards \( v_1 \) hits the object at \( f \) and exits again at \( b \). These two points form an occlusion interval that prevents rendering of objects behind \( f \). Another view ray grazes the object at a silhouette \( s \), hitting the far plane at \( s_b \). Upon moving the camera by \( \Delta \) to the new position \( v_1 \), a view ray that grazes at \( s \) now hits the far plane at \( s_f \). Camera movement effectively disoccludes the “penumbra” region between \( s, s_0 \) and \( s_1 \). (middle) In the adapted view frustum that includes all view frusta of the viewcell centered at \( v_0 \), the viewpoint moves back to \( v_1 \). Just like other camera movements, this changes the occlusion intervals related to a silhouette \( s \), and we need to adapt the trim region so that it includes \( f \) and \( b \). The new trim region includes both the light yellow and the dark yellow area. (right) The original view frustum (solid grey line) is centered at viewpoint \( v_0 \). A viewcell allows translations \( \Delta \) around \( v_0 \) and creates new view frustums with offset clipping planes (dashed grey line). We create a new view frustum with center at \( v_1 \), that exactly aligns with the offset clipping planes. With this new frustum, we can also process parts of the scene that would be clipped from the original view frustum (gray area inside the object).
has a finite extent. With a finite far plane, we can minimize the trimming applied to the occluder to the strictly necessary amount, leading to stronger occlusion effects and a smaller PVS than if a far plane at infinity was used. Finite far clipping reduces the computational cost with a negligible impact on conservativeness, as narrow erosions around far-away silhouettes contribute little to the overall PVS (see Section 6).

3.2 Disocclusion

The erosion theorem [Décoret et al. 2003] holds that visibility after a camera translation is equivalent to visibility without a camera translation when considering a shrunk (eroded) occluder. Any camera translation, as considered in the occlusion theorem, gradually changes the occlusion intervals along view rays. Provided the camera moves far enough, a view ray emitted from the new camera position may not observe the same occluder or any occluder at all.

Consider the example in Figure 4, left. At camera position $v_0$, a view ray grazes the silhouette $s$ and hits the far plane at $s_0$. Rays aimed to the left of the silhouette encounter occlusion intervals, while rays to the right, such as from $v_1$ to $s_2$, do not hit the occluder.

Let the camera move away from the original view point $v_0$ along a vector $\Delta$ to a new position $v_1$. A ray emitted at $v_1$ grazes $s$ and hits the far plane at $s_1$. Compared to $v_0$, several rays emitted at $v_1$ now reach the far plane behind the occluder between $s_0$ and $s_1$. We call this phenomenon disocclusion around $s$, because of the role the silhouette $s$ plays. The penumbra $\mathcal{P}$ is the disoccluded area formed by $s$, $s_0$ and $s_1$, while the umbra $\mathcal{U}$ is the fully occluded area formed by $b$, $s$ and $s_1$.

3.3 Trim regions

Our goal is to compute from-region visibility in a viewcell centered at $v_0$ that supports camera translation by a distance of up to $|\Delta|$. This means that we must force the same disocclusion by shrinking $\mathcal{O}$ that a camera translation by $\Delta$ would cause.

Revisit Figure 4, left: We established that the disocclusion at silhouette $s$ reveals $\mathcal{P}$. At position $v_0$, the portion of the occluder delineated by $f$, $b$ and $s$, which we call the trim region $\mathcal{R}$, is responsible for blocking view rays extending towards $\mathcal{P}$, while view rays extending from $v_1$ towards $\mathcal{P}$ are not blocked. Consequently, if we remove $\mathcal{R}$ from the occluder, we obtain the visibility for $v_1$ directly from testing view rays emitted at $v_0$ against the trimmed occluder. Primitives inside $\mathcal{U}$ are guaranteed to be invisible; only primitives in $\mathcal{P}$ can be observed and be part of a PVS at $v_0$. Note that erosion with the trim region also covers all possible camera positions between $v_0$ and $v_1$, which we shall call the viewcell $\mathcal{V}$. Regardless of where we place the camera in $\mathcal{V}$, no view ray that grazes $s$ can hit the far plane outside $\mathcal{P}$.

So far, we have only considered the effect of a translation by $\Delta$. If the camera moves along $-\Delta$, the silhouette $s$ becomes irrelevant, since it does not cause disocclusions. A disocclusion around $s$ is only relevant if the angle between the normal $n$ at $s$ and the vector $\Delta$ is acute, i.e., $n \cdot \Delta > 0$. Consequently, if we want to form a viewcell as a neighborhood around $v_0$ that extends in arbitrary directions, we have to consider all silhouettes of the occluder observed from $v_0$.

3.4 View frustum adaptation for viewcells

Moving the camera from $v_0$ to any $v_1$ necessarily changes the view frustum, so that portions of the scene clipped away for the original view frustum at $v_0$ are now included in the new view frustum established at $v_1$. As suggested by Wonka et al. [2001, Figure 6], a new frustum valid for all viewpoints $v \in \mathcal{V}$ can be created by moving the viewpoint back to $v_*$ by a distance $z$ along the negative optical axis of the original frustum formed at $v_0$ (Figure 4, middle). The distance $z = |\Delta|/\tan \alpha$ is a function of $\Delta$ and the subtended angle $2\alpha$ of the frustum. For a frustum with an aspect ratio not equal to one, the maximum subtended angle of the vertical and horizontal directions may be taken to ensure a conservative computation.

Replacing the set of frusta formed by $v \in \mathcal{V}$ with a single frustum at $v_*$ requires adaptation of the trim region, as a camera at $v_*$ observes different occlusion intervals. Figure 4, middle, shows a new view ray from $v_*$ towards $s_1$. As the silhouette $s$ and its projection $s_1$ remain unchanged, we must only form a new trim region $\mathcal{R}$, as shown in Figure 5.
spanned by $s, f$, and $b$. Depending on the location of $s$, $R_s$ can be larger or smaller than $R$ or $R_s = \emptyset$ in extreme cases.

The new frustum also contains the double-cone-shaped viewcell (orange area in Figure 4, right) that our algorithm supports, which consists of a cone with radius $\Delta$ which has its its apex at $v_s$ and its base in the plane containing $v_0$ and $v_1$, and another cone mirrored around that plane. When computing the PVS from $v_s$, we can freely move the camera within $V$. If we only required that the viewpoint be at most $\Delta$ from $v_0$, we could use a sphere centered at $v_0$ with radius $|\Delta|$ as $V$ (dashed blue semicircle in Figure 4, middle). However, we must also require that view rays emitted from within the viewcell which graze $s$ must not enter $U$ (as the blue view ray at $v_2$ in Figure 4, middle). This is only guaranteed for viewpoints inside the double-cone viewcell. Previous work [Hladky et al. 2019] demonstrates that increasing the field of view during the PVS computation on the frustum covers the rotational dimensions of the viewcell.

3.5 Multiple trim regions along one view ray

In the discussion of disocclusion above, we have only considered points on the far plane, which represents the maximum extent of the visible scene. In practice, we are interested in object pairs encountered anywhere along a view ray, provided that they form an occluder-occludee relationship.

Let $v_s$ be the viewpoint of the enlarged viewing frustum corresponding to a viewcell of diameter $\Delta$. From $v_s$, we determine the trimmed occluder $O_s = O \setminus R_s$ and find the corresponding fully occluded region $U_s$, enclosed by $s, s_1$, and $b_s$ (Figure 4).

Scene points inside the umbra are not visible from anywhere in the viewcell $V$. We are interested in the occluder fusion of $O$ with a second object $O'$ at a distance farther than $O$ from $v_s$. For this purpose, we determine the trim region $R'_s$ of $O'$ with respect to $v_s$ and classify its placement inside or outside of $U$.

If $O_s$ fully occludes $R'_s$, despite being trimmed by $R_s$ (Figure 5, left), we can unconditionally discard $O'$. Conversely, if $R'_s$ is not occluded by $O$, at all (and does not intersect $U$), as shown in Figure 5, middle, $R'_s$ creates its own umbra $U'$, which is disjoint from $U$.

If $U$ partially intersects $R'_s$, we require a more complex geometric analysis. Recall that we have made sure that, even if we trim $O$ by $R_s$, no view ray from $v_1$ can enter $U$. Consequently, any portion of $O'$ that lies within $U$ must not be disoccluded by trimming $O'$. In other words, a ray must ignore any objects encountered while passing through an umbra, but the ray is not blocked or terminated. Upon leaving the umbra, ray casting continues, and further objects encountered along the ray are considered visible. For $O'$, this means that its trim region must be formed so that it allows rays to pass only if they are neither constrained by a previous umbra nor by $O'$ after $O'$ is trimmed at its silhouette $s'$.

Consider the example in Figure 5, right. The occluder silhouette $s$ generates a trim region and an associated umbra. A part of the occludee trim region at silhouette $s'$ overlaps the umbra of $s$. Clearly, parts of the trim region for $s'$ cannot be observed from $v_1$, since they are inside the umbra of $s$. Consequently, we must not discard the occlusion intervals for those rays. However, we must not classify the region $R'_s \cap U$ as visible, either.

The remaining portion of the trim region at $s'$, i.e., $R'_s \setminus U$, is retained and used to trim $O'$. Trimming $O'$ with respect to $R'_s \setminus U$ (rather than just $R'_s$) corresponds to the occludee shrinking optimizations proposed by Décoret et al. [2003]. Our method is the first to turn this idea into a working implementation, and our results demonstrate the performance improvements it enables.

The resulting trimmed occluder $O' \setminus (R'_s \setminus U)$ is merged with previously encountered trimmed occluders and applied to determine visibility of objects further away. If consecutive trim regions for a single view ray do not overlap in depth, we compute occluder-occludee relationships as described. In real-life scenes, however, inaccurate models or non-manifold meshes may intersect, which can lead to trim region overlap. In such an event, we fuse overlapping trim regions so that the closest $f$ and the farthest $b$ of the overlapping occlusion intervals form a single trim region.

Note how we have avoided shrinking the entire occluder, as typically required in previous work on occluder shrinking or virtual occluders. In comparison to occluder shrinking, occluder trimming can be implemented in a much cheaper way. Nevertheless, it supports detailed object self-disocclusions and can handle non-convex objects. One caveat is that the disocclusion under consideration is caused by the locally identified silhouette. If the camera offset wanders too far, a different silhouette may be revealed, and a new trim region would be required to analyze it. Hence, we can expect correct results within a modestly sized viewcell, but an increasing amount of outliers the larger the viewcell becomes. This crucial relationship of viewcell size and PVS will be studied in Section 6.

3.6 Optimal trimming direction

So far, we have considered resolving only a single occluder-occludee relationship at a silhouette location $s$. To determine a complete PVS,
we must globally trim along the entire silhouette of an occluder and consider all possible occluder-occludee relationships. In particular, while we assume \(|\Delta|\) to be held constant, the direction of \(\Delta\) leading to the largest disocclusion (needed to estimate a conservative PVS) will depend on the relative position of occluder and occludee.

Figure 6 shows an occluder \(O\) (blue) that is rendered in front of an occludee \(O'\) (red). Silhouettes of polygonal meshes in three-dimensional space are formed by chains of edges that separate front and back polygons [Benichou and Elber 1999]. In our example, \(O\) and \(O'\) are separated by the silhouette edge \(E\) of \(O\).

Consider the effect of a camera translation \(\Delta\) on the relation between the two objects. The projection \(s_1\) of a point \(s\) on \(E\) to the far clipping plane moves dependent on the direction of \(\Delta\). After projection to the 2D image plane, a maximum disocclusion (farthest distance between \(s\) and \(s_1\)) is obtained when aligning \(\Delta\) with the normal \(n\) of the silhouette edge \(E\) that points away from the occluder. With this observation, we obtain the direction \(-n\) in which to displace \(E\) to obtain the trim region which maximizes disocclusion.

Unfortunately, as we have seen in Section 3.5, the order in which occluders are trimmed can influence the shape of the trim regions. Consequently, we cannot simply apply trimming to all occluders indiscriminately, i.e., in random order. We first need to establish a depth order of the scene as seen from \(v\), and use this order to incrementally resolve occluder-occludee relationships to determine exactly which regions need to be trimmed away and which portions of the scene become part of the PVS.

4 VISIBILITY CULLING ALGORITHM

In this section, we describe the implementation of the trim region algorithm, which, like most visibility algorithms, is inherently a depth-sorting problem. Global sorting of all occlusion intervals and trim regions for the entire scene is certainly possible [Hladky et al. 2019a], but very inefficient. Instead, we propose to apply a divide & conquer strategy to achieve better scalability, especially by leveraging early ray termination. For this purpose, we wrap the scene geometry with an octree and extract view-dependent layers of nodes from the octree at runtime. Each layer contains the maximum number of nodes that have been fully disoccluded by removing the layers in front. Since nodes inside a layer do not occlude each other, we may process the primitives of the entire layer in parallel.

The overall algorithm consists of five major phases (Figure 8), labeled P0-P4, where P0 runs on the CPU, while P1-P4 each consist of one or more rasterization or compute passes on the GPU. While P0 and P4 are executed once per frame (before and after processing the layers, respectively), P1-P3 are executed once per layer.

P0 generates drawbuffers corresponding to the layers of the octree as seen from the current point of view (Section 5). P1 preprocesses the geometry of the current layer and prepares geometric primitives for the subsequent stages. This includes the geometry of the scene itself, the trim regions and the umbrae. P2 rasterizes the geometry prepared in P1. While P1 and P2 are merely preparatory steps, P3 implements the core trim region logic. It uses information rasterized in P2 to form and analyze occlusion intervals, ultimately determining potentially visible geometry. Even more importantly, it determines whether a ray is fully occluded and can be terminated after the current layer. Finally, P4 harvests visibility information accumulated during traversal of the layers to produce the final PVS. Throughout the algorithm, the following data structures (see Table 1) are used to communicate between the phases:

- The trim region stencil indicates the state of a given pixel (i.e., view ray), which can be untrimmed (no trim region has been encountered yet), trimmed (at least one trim region has been encountered) or sealed (after encountering a trim region, the pixel has been found occluded by another primitive and need not be investigated further). The trim region stencil is initialized with untrimmed before the first layer.
- The active pixel stencil indicates if new information regarding a given pixel location has been found in the current layer. The active pixel stencil is cleared before the first layer, set in P2 for pixels that receive new data, and reset in P3 for all pixels that have been processed.
- The visibility buffer contains the visible fragments (stored as depth+id) of rasterized primitives that have not been trimmed. After finishing the last layer, the ids stored in the visibility buffer form the bulk of the primitives contained in the PVS.
- The trimmed primitives buffer contains a list of primitive ids that partially or fully lie in a trim region and therefore have been suppressed from rasterization into the visibility buffer. The trimmed primitive buffer collects these ids and is later merged with the visibility buffer into the final PVS.

The visibility buffer collects visible primitives in screen areas not covered by a trim region, where the first frontface already leads to full occlusion of the view ray. In areas covered by a trim region, multiple frontfaces may be classified as visible; these are all inserted into the trimmed primitives buffer directly.

4.1 Geometry generation

Phase P1 consists of a single compute pass that generates four draw buffers from the geometric primitives associated with the current

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer's primitives</td>
<td>Draw buffer</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trim region quads</td>
<td>Draw buffer</td>
<td>W</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FF primitives</td>
<td>Draw buffer</td>
<td>W</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BF primitives</td>
<td>Draw buffer</td>
<td>W</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Umbra quads</td>
<td>Draw buffer</td>
<td>W</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trim regions</td>
<td>Depth k-buffer</td>
<td>W</td>
<td>R</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>Frontfaces</td>
<td>D+ID k-buffer</td>
<td>W</td>
<td>R</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>Backfaces</td>
<td>Depth k-buffer</td>
<td>W</td>
<td>R</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>Umbrae</td>
<td>Depth k-buffer</td>
<td>W</td>
<td>R</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>Trim region stencil</td>
<td>Image</td>
<td>RW</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active pixel stencil</td>
<td>Image</td>
<td>RW</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visibility buffer</td>
<td>Depth+ID</td>
<td>W</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trimmed primitives</td>
<td>Buffer</td>
<td>W</td>
<td>RW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex pixels</td>
<td>Draw buffer</td>
<td>RW</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Overview of the buffers used by the algorithm (FF/BF = frontfacing/backfacing in clip space, D+ID = depth + ID, R = read, W = write, RW = read/write)
Fig. 8. The trim region algorithm commences in five phases: P1 determines the octree layers on the GPU. P1 generates the draw buffers for each layer, and P2 renders them. P3 resolves the resulting trim sequences. P4 gathers the results into the final PVS.

layer. First, a draw buffer containing the trim region quads is created by extruding each silhouette edge inwards, along the negative normal of the edge in image space (Figure 7). We only include silhouettes as observed from \( v_i \). In our experiments, including all edges that could potentially become silhouettes within the view cell did not significantly improve the accuracy of the PVS, but processing these redundant entries considerably reduced the performance.

Next, the layer’s primitives are transformed into clip space and culled using the Cohen-Sutherland [Hill and Kelley 2006] method. Surviving primitives are classified as front-facing or back-facing, and stored in draw buffers for FF and BF primitives, respectively.

A final draw buffer contains umbra quads, which represent the separating faces between umbra and penumbra (Figure 4, left). These are spanned between the two vertices of the silhouette edge and the projections of the opposing extruded edge on the far plane. If a primitive lies between this face and the front face of the occlusion interval corresponding to the trim region, it is guaranteed to be occluded with respect to the trimmed occluder, so we discard it.

4.2 Geometry rasterization

In phase P2, we rasterize the draw buffers generated in P1. Since we need to keep all rasterized fragments until the resolve phase P3, each rasterization target a \( k \)-buffer with (at least) a depth attachment, accessed via an atomically operated counter. The dimension \( k \) is chosen to reflect the expected maximum depth complexity per layer. If a pixel has been marked sealed in a previous layer, we discard all rasterization results for that pixel, as these results do not contribute to the PVS. For every pixel written into a \( k \)-buffer, the corresponding location in the active pixel stencil is set.

During rasterization of trim region quads, if a pixel is found to be yet untrimmed, we set the trim region stencil to trimmed. For
rasterization of front-facing primitives, we store not only depth, but also primitive id in the $k$-buffer. Moreover, the front faces are rasterized into the visibility buffer as well.

### 4.3 Occlusion interval resolve

Phase P3 resolves the $k$-buffers accumulated in the previous phase to determine which primitives of the current layer should be added to the PVS. It only acts on pixels that have been marked as active in P2. Since the octree only provides approximate ordering, we first need to locally sort the geometry encountered along each view ray into matching occlusion intervals. Hence, we sort the frontface and backface $k$-buffers by depth into a temporary trim sequence.

**Forming and validating occlusion intervals.** We scan the sequence for simple occlusion intervals, i.e., frontface/backface pairs that are immediately adjacent in the sequence. Given an ideal scene (consisting of watertight non-intersecting polyhedra), we would only encounter simple occlusion intervals. Alas, real scenes assembled from a polygon soup can lead to multiple consecutive front faces or back faces. To deal with such poorly conditioned input, we adopt a robust strategy that avoids false negatives (i.e., missed geometry that should be in the PVS), while minimizing false positives (i.e., occluded geometry that is needlessly included in the PVS). We first greedily search for simple occlusion intervals with arbitrary depth $[f; b]$. If another frontface $f'$ precedes $f$ in the sequence, the interval is extended to $[f'; b]$. Extending an interval at the front can only introduce false positives, but not false negatives. Multiple consecutive back faces do not matter, since the backface will later be replaced with an umbra. In a subsequent step, we validate the intervals identified so far. A validated interval must contain at least one fragment $t$ from the trim region $k$-buffer, i.e., $f \leq t \leq b$.

**Fitting occlusion intervals with umbrae.** For each validated interval, we replace its backface with the corresponding umbra. All entries of the trim sequence contained in the extended interval from front face to umbra are not visible and can be ignored. Finding the replacement is simple, since there is a 1:1 correspondence of trim region to umbra. As the umbra typically has a larger depth than the backface it replaces, we must sort the sequence again by depth. After resorting, we assign $+1$ to the frontface of a validated interval, and $-1$ to the umbra. All other entries in the sequence are assigned 0. A prefix sum over the sequence reveals if any entity in the trim sequence is affected by an umbra. If the prefix sum is larger than 0, such an entry can be marked for deletion. Please see the examples in Figure 9.

**View ray termination.** Early ray termination is facilitated by looking for a terminator face in the trim sequence. A terminator face is a front face or back face that is not contained in a validated interval (prefix sum is zero). Since such a terminator face represents untrimmed geometry, it includes all further entries, and we need not continue investigating the ray. Starting at the smallest depth value, we visit the entries of the sequence in order until we encounter a terminator face. Upon visiting an interval, all entries between front face and umbra are marked for deletion. This step is important to avoid premature identification of terminator faces, in particular when entries resulting from non-manifold geometry do not form proper front face/back face pairs.

Note that every validated interval is visited, irrespective of whether its entries are already partially marked for deletion for being inside another interval. The frontface of every validated interval is added to the trimmed primitives buffer. If we find a terminator face, we set the trim region stencil to sealed and abort further searching. If the terminator face is a frontface, we add it to the trimmed primitives buffer. If we cannot find a terminator candidate, the residual trim sequence is simply left in place to be resolved in subsequent layers. In particular, an unfinished interval (a front face at the end of trim sequence) is not classified as a terminator face immediately, but instead carried over to the next layer.

**Sorting implementation.** We have two options to sort the $k$-buffer entries on the GPU, either sequentially with one thread per pixel or in parallel with multiple threads collaboratively working on a single pixel. Since the $k$-buffer length can drastically vary across screen space and between layers, neither strategy is optimal. Hence, we employ a hybrid sorting strategy that involves two compute passes. The first pass launches one thread per pixel and inspects the length of the trim sequence. If it is shorter than a threshold (empirically determined as $N = 8$), the thread sorts its sequence sequentially. Otherwise, we flag the pixel for parallel sorting by writing the pixel location to a “complex pixels” draw buffer (Figure 8, P3) with compute indirect commands. The second compute pass spawns one compute group for each complex pixel to apply efficient bitonic sorting. Since compute groups typically operate in lockstep (for a group of 32 threads), the trim sequence length is ideally a multiple of 32. Each primitive type fills a $k$-buffer of
fixed length 32, which we consider during the octree build step to prevent overflows. Clearly, there is a trade-off between memory consumption and performance: Deeper octrees subdivide the scene into smaller chunks and require smaller k-buffers, but potentially do not utilize thread groups efficiently.

4.4 PVS gathering

After all layers, a compute shader collects the final PVS in phase P4. It visits all pixels of the visibility buffer and appends them to the trimmed primitives buffer, which becomes the final PVS buffer.

5 OCTREE LAYER PEELING

Our visibility culling algorithm peels layers off an octree covering the scene. Each layer consists of the nodes that can be traversed in parallel, since they do not overlap in image space. We impose an upper limit on the number of primitives allowed per octree node [Greene 1995]. Hence, the depth complexity of trim regions inside a node is bound by a small number. The layer generation algorithm described in this section runs in <1 ms on a single CPU core for octrees with thousands of nodes.

5.1 Octree generation

The octree is created by subdividing nodes based on the axis-aligned bounding boxes (AABB) of the objects in the scene. If a node contains an AABB that would fit fully into a child node, the node is subdivided, unless a maximum depth is reached. Then, we alternate between two procedures until convergence:

1. Sorting. We sort the primitives of an object into the corresponding octree nodes. A primitive that overlaps multiple nodes is sorted into each of the relevant nodes.
2. Balancing. Since we must traverse the octree using neighborhood relationships, an excessive level difference between neighbors is inefficient [Duchaineau et al. 1997]. Therefore, we constrain the difference between neighbors to no more than two levels (i.e., a node may have up to 16 neighbors per face). Violating nodes are subdivided, until no more violations are found.

We only consider static objects during octree generation. At runtime, we lazily sort any dynamic (e.g., animated) objects that possibly traverse octree node borders into the respective octree nodes according to their current bounding box in each PVS frame.

5.2 Layer generation

The purpose of the octree layer generation is to identify a minimum partitioning of octree nodes into layers, such that nodes in a layer do not overlap in image space and thus can be traversed in parallel.

This requirement can be trivially fulfilled if nodes are processed sequentially [Greene et al. 1993], but at a high cost of one separate drawcall per node [Serpa and Rodrigues 2019]. Finding the minimal number of layers (and, hence, drawcalls) is a combinatorial problem significantly more complex than the sequential solution.

The method of Laine [2005] uses a simple FIFO queue to find the minimal layers. Dequeued nodes are tried repeatedly, until all their dependencies have been visited (Figure 10). Unfortunately, the number of times a node has to be re-visited in Laine’s method grows rapidly with octree size, leading to poor runtime performance.

We avoid this problem by enqueuing only nodes with fulfilled dependencies. As a start node, the one closest to the viewpoint among those nodes intersecting the near plane is enqueued. From then on, nodes are processed in FIFO order. A node is dequeued, and the layer count for each neighbor is updated to at least the current node’s layer plus one. Next, we mark the neighbors’ faces which touch the current node as fulfilled dependencies. Neighbors which have all their dependencies fulfilled are enqueued. If a processed node is touching nodes which have a higher degree of subdivision than the node itself, these nodes are considered strictly in layer order, i.e., only nodes directly touching the current node are examined. In addition, we cull all nodes against the view frustum at $v_6$.

6 RESULTS

We evaluated the performance of our method for various scenes and parameter choices. Tests were run on a desktop computer (CPU: Intel i7-7770 with 64 GB RAM, GPU: NVidia GeForce RTX 4090, Windows 10). We assumed a field of view for the target frames of 60° and an extended field of view of 90°, allowing head rotations up to ±15°. We performed both the PVS computation and the rasterization of target frames at a resolution of 1920 × 1080 pixels.

We used the test scenes shown in Figure 11. For every scene, we recorded an animated camera path, each with a length of 400–600 frames, for reproducible measurements. All scenes were enclosed in an octree with a subdivision depth of 3, except City, with a depth of 4.

In our test paths, we assume a running speed of 3 m/s and a frame rate of 60 Hz, so the camera moves 5 cm between two frames. The viewcell radius $|\Delta|$ was varied from 5 cm to 30 cm, as proposed by Hladky et al. [2019a]. The height $z = |\Delta|/\tan \alpha$ of the viewcell cone, which corresponds to the forward motion (Figure 4, middle) is equal to $|\Delta|$ for $2\alpha = 90^\circ$. Hence, for a forward motion of 5 cm per frame, a PVS with a viewcell size of $\Delta$ is valid for a segment of $|\Delta|/5$ frames.
6.1 Comparison to state-of-the-art methods

As discussed in Section 2, there is a limited choice of methods that can compute a from-region PVS online. For example, it was recently demonstrated [Hladky et al. 2019a] that the 2.5D Instant Visibility method of Wonka et al. [2001] is unable to handle generic 3D scenes properly. Hence, we compare our trim region method (TR) to two recent methods that work on 3D scenes, namely, camera-offset space (COS) and shading atlas streaming (SAS), and to one from-viewpoint method, occlusion queries (OQ), which benefits from dedicated hardware support on the GPU.

COS [Hladky et al. 2019a] is a recent visibility method that targets the same use cases as ours. Its occlusion data structure is based on global per-pixel linked lists of all rasterized fragments. Unfortunately, technical limitations prevented us from running the original COS code on current hardware. In order to make a meaningful comparison to the performance numbers reported originally, we used some of the original scenes and the same settings. According to public benchmarks [Wilson et al. 2023], we estimate that the GPU we used (NVIDIA RTX 4090) is approximately 2× faster than the one used in the COS paper (NVIDIA Titan Xp). We report the original times and hypothetical times accelerated by this factor.

SAS [Mueller et al. 2018] is a streaming rendering system which relies on extrapolating a user’s viewpoint several frames into the future using first-order prediction, followed by sampling an EVS via rendering a standard visibility buffer for each extrapolated position. A from-region PVS is approximated as the union of EVS samples. Occlusion query methods like CHC++ are from-point methods and compute a visible set from a single point $v_0$.

6.2 Speed

We measured the runtime as a function of scene and of viewcell size. Figure 13 shows how the overall runtime of TR can be broken down into phases P0 to P4. The overall times to produce a PVS are relatively uniform in the 50-60 Hz range, with a modest increase of around 10-20% when increasing the viewcell size sixfold from 5 cm to 30 cm. Among the phases, P1 (geometry generation) dominates the runtime, consuming about 60-70% of the allotted time. A large portion of this time is likely related to the fact that P1 is at the tip of the GPU pipeline and must wait for the previous layer to complete.

At our assumed movement speed of 5 cm per frame at 60 Hz, we can expect that the PVS remains valid between one frame (16.7 ms) at a viewcell size of 5 cm, and, six frames (100 ms) at 30 cm viewcell size. For the observed worst case in the PVS computation runtimes, 22.26 ms, this means a break-even at a viewcell size of only 6.7 cm (corresponding to 1.3 frames). Using a larger viewcell than 6.7 cm will proportionally increase the benefits of predicting the PVS. Even though this number does not include any additional server-side processing of the PVS, we may assume that our method is well suited for streaming applications in terms of its runtime performance.

Table 2. The k-buffer memory requirements for Trim Regions at a resolution of 1920×1080. Our memory footprint increases linearly with the number of active pixels due to fixed k-buffer depth and iterative layer processing. The average trim sequence length remains manageable even for larger viewcell sizes due to our early stopping and divide-and-conquer strategies.

| Scene     | $|A|$ | Pixels | Sequence Length | Allocated/used memory (mb) |
|-----------|-----|--------|-----------------|----------------------------|
| Viking village | 5  | 410495.1 | 12.2 | 250.5/32.7 |
|           | 10 | 509036.3 | 12.1 | 310.7/41.6 |
|           | 30 | 588554.0 | 15.8 | 359.2/67.4 |
| Robot lab | 5  | 418036.2 | 8.3  | 255.2/22.4 |
|           | 10 | 593784.8 | 8.4  | 362.4/32.7 |
|           | 30 | 792573.7 | 11.2 | 483.8/60.3 |
| Sponza    | 5  | 395954.7 | 10.8 | 241.7/27.5 |
|           | 10 | 599549.3 | 10.9 | 365.9/42.4 |
|           | 30 | 923812.6 | 12.6 | 563.9/77.8 |
| Sun       | 5  | 551648.7 | 16.2 | 336.7/55.8 |
| temple    | 10 | 804346.3 | 16.5 | 490.9/80.8 |
|           | 30 | 1133150.7| 17.7 | 691.6/128.3|
| City      | 5  | 167540.2 | 17.8 | 102.3/18.4 |
|           | 10 | 288516.1 | 17.2 | 176.1/30.7 |
|           | 30 | 541491.4 | 16.1 | 330.5/54.3 |
virtually no errors. Alas, its expensive linked-list representation lets memory consumption grow rapidly, and performance deteriorates rapidly as the viewpoint size increases.

Our memory requirements are mostly driven by the $k$-buffers. However, our divide-and-conquer approach enables us to use fixed-size $k$-buffers. The early stopping strategy significantly reduces trim sequence lengths, i.e., the union of all $k$-buffer entries per pixel. Table 2 provides an analysis of the memory consumption. The viewpoint size dictates the number of pixels relevant for trimming, but has little impact on the average trim sequence length. Compared to COS, our trim sequences are much shorter and less sensitive to the viewpoint size, with fewer active pixels. Iteration over octree layers can work on smaller, distinct sequences iteratively and terminate pixels much earlier. Scenes with higher depth complexity (e.g., City) require a deeper octree to prevent overflow. Fortunately, a deeper octree only impacts runtime, but has hardly any storage costs.

Its high computational cost limits the application of COS to small viewpoint sizes, where it directly competes with TR. In comparison, TR can only solve 4D visibility in the local neighborhood of an object silhouette, but at a much lower computational cost. Please refer to Section 6.4 for an analysis of the resulting quality.

The runtimes of OQ, despite delivering only a from-viewpoint and not a from-region PVS, are substantially higher than those of SAS and TR. The main slowdown that affects hardware occlusion queries is caused by the waiting times induced by the layer-to-layer synchronization. In TR, at most 6 (octree depth 3) or 14 layers (octree depth 4) suffice to generate a reasonably tight PVS, since the P3 phase of TR performs local depth sorting inside each layer. In contrast, OQ requires an average of roughly 80 layers (octree depth 8) to produce a similarly tight PVS and suffers excessively from synchronization.

6.3 PVS size and tightness

Important performance indicators for the efficiency of a PVS method are its size and tightness. We denote the primitive count of a scene as $SC$ and the primitive count of the PVS determined by our TR method as $TRC$. Moreover, we determine the primitive count $GTC$ of the ground truth PVS of a viewpoint by dense sampling, i.e., we compute

the union of primitives contained in 200 EVS samples uniformly spaced in a given viewpoint. As pointed out by Hladky et al. [2019a] and earlier by Wonka et al. [2006], this only approximates a true PVS, but the level of accuracy is sufficient (typically >99%) to allow using it to make meaningful comparisons.

The PVS size of the ground truth is given as $GTC/SC$, and the PVS size of TR is given as $TRC/SC$. This tells us which viewpoint size is still acceptable: A larger viewpoint size means that the PVS remains valid longer, but at the price of having to handle a larger PVS.

Moreover, we investigate the tightness achieved by a particular PVS method. Tightness is related to the rate of false positive primitives, which are included in the PVS, but never actually become visible (see Figure 1). If $FP$ denotes the false positive primitive count of TR, we define the false positive rate as $FP/GTC$, i.e., the factor by which the PVS is increased. Figure 15 reports the average PVS size and tightness for various scenes and viewpoint sizes.

Tightness is reported without ($FP$) and with ($FP^*$) suppression of unreliable primitives. For the latter, we ignore all tiny or slithery triangles in the computation which, after projection to image space, have a size of less than one pixel along any of their three edges. Such primitives frequently produce zero fragments during rasterization, despite having edges that may be hundreds of pixels long and an area equivalent to dozens of pixels. In the $FP^*$ rate, computed as $FP^*/GTC^*$, we omitted such primitives in all counts to better understand the influence of poor geometric conditioning on the results. Robot Lab suffers most from unreliable geometry due to its numerous thin, elongated triangles, e.g., at rounded corners and coplanar, slightly offset planes. The comparably large depth extent

![Fig. 13. Runtime of TR for various scenes and viewpoint sizes $|\Delta|$, reported in ms, broken down by algorithm phase, where P0 is octree layer computation, P1 is geometry generation (which includes the layer-to-layer synchronization overhead), P2 is geometry rasterization, P3 is occlusion interval resolution, and P4 is PVS harvesting.](image1)

![Fig. 14. Runtime comparison for various methods, scenes, and viewpoint sizes $|\Delta|$, reported in ms. "COS*" (yellow) is the time of "COS" (light yellow) scaled by 0.5 to estimate runtime on the modern RTX 4090 GPU used in our experiments (data only available for Viking village and Robot lab). "OQ" (orange line) gives the runtime of the from-viewpoint OQ method, which has to recompute a new PVS for every frame.](image2)

<table>
<thead>
<tr>
<th>Scene</th>
<th>Viking Village</th>
<th>Robot Lab</th>
<th>Sponza</th>
<th>Sun Temple</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>\Delta</td>
<td>$</td>
<td>$FN$</td>
<td>$FN^*$</td>
<td>$</td>
</tr>
<tr>
<td>5</td>
<td>0.031</td>
<td>0.013</td>
<td>0.048</td>
<td>0.046</td>
<td>0.029</td>
</tr>
<tr>
<td>10</td>
<td>0.031</td>
<td>0.014</td>
<td>0.047</td>
<td>0.044</td>
<td>0.030</td>
</tr>
<tr>
<td>30</td>
<td>0.037</td>
<td>0.019</td>
<td>0.050</td>
<td>0.047</td>
<td>0.048</td>
</tr>
</tbody>
</table>
Whenever we face situations that are difficult to resolve (we can expect 1-2 orders of magnitude speed-up when processing at the chosen depth of eight, OQ (with runtimes of 20-70 ms) had (i.e., rendering, streaming, etc.) the resulting TR PVS instead of the (1-12% except for Sponza) of the overall scene size. Consequently, the FP rate suggests that the PVS of TR is 1.7-3\times the size, but is generally a few percent (1-8%) of the total scene size, except for Sponza, which is too small overall to fit the pattern.

We see that the ground-truth PVS size depends on the viewpoint size, but is generally a few percent (1-8%) of the total scene size, except for Sponza, which is too small overall to fit the pattern. The FP rate suggests that the PVS of TR is 1.7-3\times larger than the ground truth, but the TR size still represents a small percentage (1-12% except for Sponza) of the overall scene size. Consequently, we can expect 1-2 orders of magnitude speed-up when processing (i.e., rendering, streaming, etc.) the resulting TR PVS instead of the original scene. These numbers also compare favorably to the state of the art. For example, the PVS size of TR for Robot lab is about 9% of the full scene, while COS reports a PVS size of 17%, i.e., almost twice the size. OQ determines visibility on the granularity of octree nodes, as opposed to primitive (group) granularity, which makes it very dependent on the octree depth. A shallow octree delivers too many false positives to be useful, and a deep octree makes OQ slow. At the chosen depth of eight, OQ (with runtimes of 20-70 ms) had about 2-3\times the FP rate of TR.

The driving factor for false positives is our conservative strategy. Whenever we face situations that are difficult to resolve (e.g., due to non-manifold or non-watertight meshes), our heuristics are tuned towards avoiding false negatives at the expense of more false positives. We close as many intervals as possible by merging unassigned single front- or backface entries into existing intervals and potentially trim more than necessary. In rare cases, a trim region is larger than the associated feature. In a similarly small number of cases, we potentially assign unrelated intervals from unaffected parts of the same object to this trim region, which adds to the FP rates. While it is possible that the face separating \( U \) and \( P \) is closer to the camera than the backface associated with the trim region, this is extremely rare, since it requires the backface at the silhouette being almost parallel to the view ray grazing the silhouette. As before, we trim slightly more conservatively than necessary, at the cost of slightly increasing FP.

### 6.4 PVS correctness

For the correctness of the PVS (and the resulting image quality), we investigate false negatives, i.e., the number \( FN \) of primitives that are incorrectly omitted from the PVS. We are most concerned about the impact of false negatives on the quality of the final image, not necessarily the absolute value of \( FN \). Since we use rasterization for both the PVS computation and for generating the final images, we must expect that the finite numeric precision of a GPU leads to results that differ in certain pixel locations to results that are computed analytically. Consequently, two different GPU models will rarely produce the exact same rasterized image.

Therefore, we focus on reporting pixel error rates, as suggested by Wonka et al. [2006], who state that “The term conservative (or even exact) visibility is actually quite misleading. Most algorithms, though conservative in theory, are not conservative in practice due to numerical robustness problems. This is especially true for algorithms that rely on graphics hardware.” They report a rate of \( \leq 0.005\% \)

---

**Fig. 15.** Average PVS size tightness for various scenes and viewcell sizes \(|\Delta|\). In the top chart, “TR size” and “GT size” indicate the percentage of the full scene placed in the PVS of the trim region method and the ground truth, respectively. In the bottom chart, “FP rate” and “FP* rate” indicate the false positive rate (as a multiple of the ground truth) without and with suppression of unreliable primitives, respectively.

**Fig. 16.** Average false negative pixel count per frame for all possible views. FNG (solid lines) show results with unreliable triangles; FNG* (dashed lines) shows results without unreliable triangles. When applying the cut-off suggested by Wonka et al. [2006] of roughly 100 pixels, a viewcell size of 15-20 cm can usually be supported without any noticeable errors.
false negative pixels (corresponding to 103 pixels in our 1920 × 1080 frames) and found it to be negligible with respect to image quality (i.e., an average observer would not notice the difference).

Figure 16 shows $F_{NG}$, the average false negative pixel count per frame, averaged over all 200 views used for the ground truth of this viewcell. The data is plotted as a function of trim region size, once with unreliable triangles ($F_{NG}$) and once without ($F_{NG}^*$). Table 3 lists the false negative primitive rates $F_N/GTC$ and $F_N^*/GTC^*$.

The data for $F_{NG}$ shows that all our test scenes have less than the desired 0.005% error rate up to trim sizes of around 15-25 cm. It can be seen that the corresponding $F_N$ rates of 3-5% (Table 3) are not linearly related to the pixel errors and reveal little about the resulting visual errors. Residual false negative primitives mostly result from not watertight or non-manifold models, or primitives inside of closed objects. These primitives never contribute to the object’s appearance, but they do complicate the resolving process.

The reader is invited to inspect Figure 17 for side-by-side comparisons between renderings of the full scene and the TR PVS. We chose challenging locations with poorly modeled geometry in the front (e.g., the crane arm in Robot lab contains self-intersecting geometry). The peak signal-to-noise ratio when comparing the left/middle images is 63.3 dB for the Robot lab example and 58.0 dB for the Sun temple example. In comparison, highest quality JPEG compression typically achieves 50 dB, while 20-25 dB is commonly considered acceptable in streaming applications [Thomos et al. 2006].

Concerning the other methods, COS does not suffer from false negatives, but pays for this property with a high runtime. The analytical approach of COS is probably only prone to floating point errors, while TR additionally suffers from the issues that affect all rasterization approaches. Very thin or small triangles potentially do not occupy any fragments during rasterization and may not show up in our PVS. This is a negligible issue for thin trim regions, as they could only disocclude correspondingly thin portions of the scene.

OQ has an acceptable false negatives rate, but mostly because of greedily consuming large octree nodes, leading to high false positives and poor performance. SAS misses a large portion of the visible primitives for larger viewcell sizes (10% of the full scene primitives for 5 cm and 25-30% of the full scene primitives for 30 cm), leading to severe visual artifacts. TR combines a low false negative rate with a tight PVS and good performance.

7 CONCLUSION AND FUTURE WORK

We have presented a system for real-time generation of from-region PVS for 3D scenes previously not addressed at this scale of complexity. Our method constructs trim regions, i.e., volumetric regions that force disocclusion, strategically placed in image space so that a from-point geometry pass can identify a tight from-region PVS. This allows us to use our method in streaming rendering or low-latency virtual reality applications, where the PVS of a dynamic scene must be identified a few frames ahead of time.

We see several directions for future work. Trim regions could be extended to incorporate coarse-to-fine culling, operating first on octree nodes as bounding volumes. Moreover, we could exploit the novel GPU extension for simultaneous multi-viewport rendering to efficiently subdivide the view ray space and reduce discretization errors and required safety measures for trim region sizes. Finally, we want to apply our method to other areas where visibility information is beneficial, such as shadow rendering or global illumination.

ACKNOWLEDGMENTS

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged. VRVis is funded by BMK, BMDW, Styria, SFG, Tyrol and Vienna Business Agency in the scope of COMET – Competence Centers for Excellent Technologies (879730) which is managed by FFG.