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

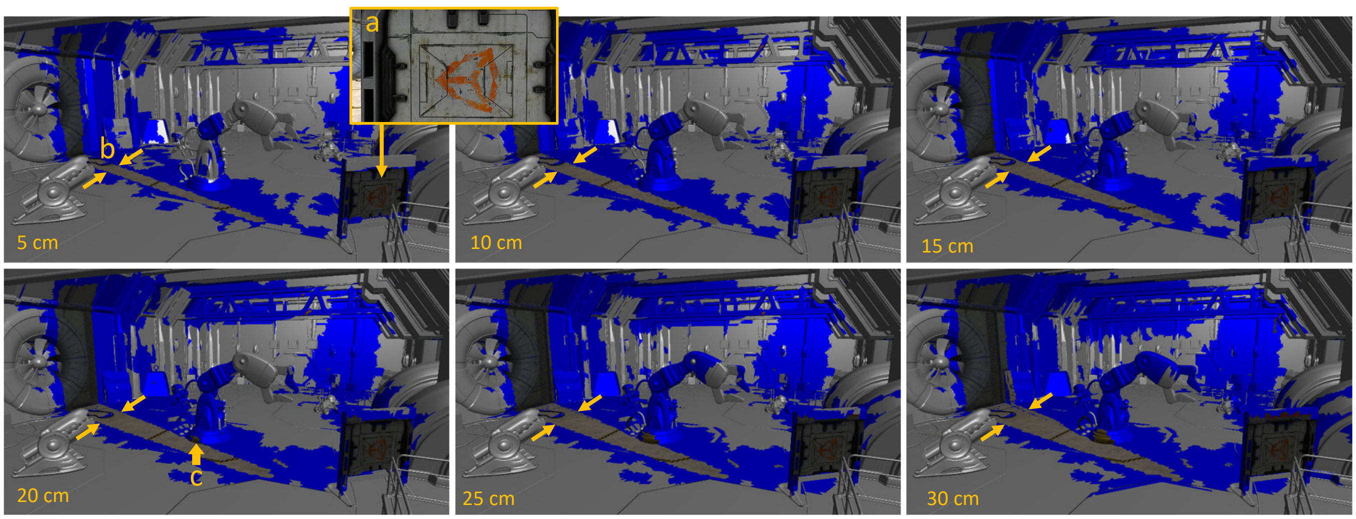

Fig. 1. (a) The view in the inset is seen by an observer standing at the location indicated by the inset's leader line. Our method computes a potentially visible set (PVS) corresponding to a viewcell (region) of a given radius around the viewpoint in real time. The six images each show a PVS for view 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:
Philip Voglreiter, Bernhard Kerbl, Alexander Weinrauch, Joerg H. Mueller, Thomas Neff, Markus Steinberger, and Dieter Schmalstieg. 2023. Trim Regions for Online Computation of From-Region Potentially Visible Sets. *ACM Trans. Graph.* 42, 4 (August 2023), 15 pages. https://doi.org/10.1145/3592434

P. Voglreiter, J. Mueller and T. Neff are affiliated with the Christian Doppler Laboratory of Semantic 3D Vision at Graz University of Technology.
Authors' addresses: Philip Voglreiter, voglreiter@icg.tugraz.at, Graz University of Technology, Austria and VRVis Forschungs GmbH, Austria; Bernhard Kerbl, kerbl@cg.tuwien.ac.at, TU Wien, Austria and Inria, Université Côte d'Azur, France; Alexander Weinrauch, alexander.weinrauch@icg.tugraz.at, Graz University of Technology, Austria; Joerg H. Mueller, joerg.mueller@icg.tugraz.at, Graz University of Technology, Austria; Thomas Neff, thomas.neff@icg.tugraz.at, Graz University of Technology, Austria; Markus Steinberger, steinberger@icg.tugraz.at, Graz University of Technology, Austria; Dieter Schmalstieg, schmalstieg@tugraz.at, Graz University of Technology, Austria.

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.

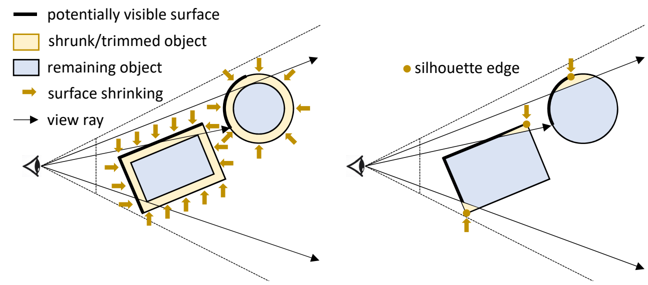

Fig. 2. (left) Geometric occluder shrinking needs to erode the entire object geometry to determine disocclusions, which is difficult to do analytically in 3D. (right) It is sufficient to only chisel away areas at the silhouette edges, which we call *trim regions*. This can be done using screen-space rasterization.

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 potent 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 CPU, 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].

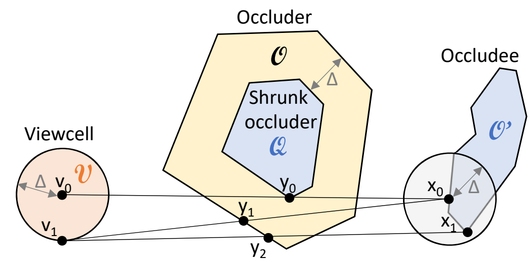

Fig. 3. The erosion theorem states that determining from-point visibility with respect to a shrunk occluder Q is equivalent to from-region visibility of the original occluder O. Any ray emitted from within a viewcell $\mathcal{V}$ formed around $\mathbf{v}_0$ cannot observe an occludee O', if O' is blocked by Q from $\mathbf{v}_0$. Even the extremal point $\mathbf{v}_1$ in $\mathcal{V}$ cannot see the extremal point $\mathbf{x}_1$ 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

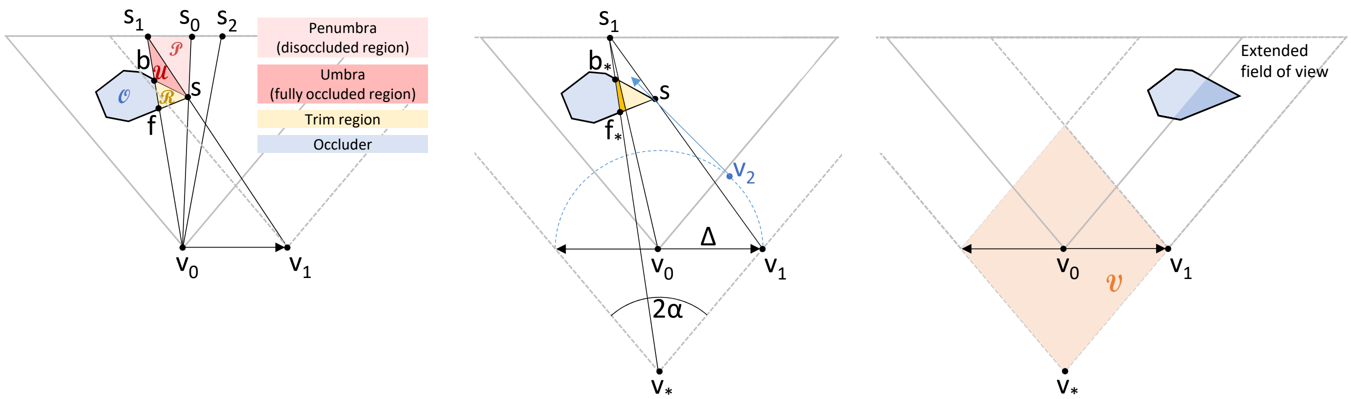

Fig. 4. **(left)** A camera located at position $\mathbf{v}_0$ looks at an object. A view ray extending towards $\mathbf{s}_1$ hits the object at $\mathbf{f}$ and exits again at $\mathbf{b}$. These two points form an occlusion interval that prevents rendering of objects behind $\mathbf{f}$. Another view ray grazes the object at a silhouette $\mathbf{s}$, hitting the far plane at $\mathbf{s}_0$. Upon moving the camera by Δ to the new position $\mathbf{v}_1$, a view ray that grazes at $\mathbf{s}$ now hits the far plane at $\mathbf{s}_1$. Camera movement effectively disoccludes the "penumbra" region between $\mathbf{s}$, $\mathbf{s}_0$ and $\mathbf{s}_1$. **(middle)** In the adapted view frustum that includes all view frusta of the viewcell centered at $\mathbf{v}_0$, the viewpoint moves back to $\mathbf{v}_*$. Just like other camera movements, this changes the occlusion intervals related to a silhouette $\mathbf{s}$, and we need to adapt the trim region so that it includes $\mathbf{f}_*$ and $\mathbf{b}_*$. 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 $\mathbf{v}_0$. A viewcell allows translations Δ around $\mathbf{v}_0$ and creates new view frustums with offset clipping planes (dashed grey line). We create a new view frustum with center at $\mathbf{v}_*$ 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).

of camera movements under which one triangle (2D) is covered by another triangle (2D). Their approach limits occluder fusion to explicitly connected primitives and requires unbounded linked lists per pixel, which makes the method scale poorly to large scenes.

Our method is a re-interpretation of occluder shrinking proposed by Wonka et al. [2000]. Here, an occluder is eroded so its projection to image space shrinks. View rays (2D) from the original viewpoint can pass the shrunk occluder projection (2D) in the same way as if they were shot from a translated viewpoint inside a region (Figure 2).

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 $\mathbf{v}_0$ to a surface point $\mathbf{x}_0$ is blocked by an occluder O (Figure 3). Any ray towards $\mathbf{x}_0$ starting at a viewpoint $\mathbf{v}_1$ taken from a spherical viewcell $\mathcal{V}(\mathbf{v}_0, \Delta) = \{\mathbf{v}_1, \text{ s.t. } |\mathbf{v}_0 - \mathbf{v}_1| < \Delta\}$ passes O at a distance smaller than Δ. This occlusion relationship is invertible: One can determine a shrunk occluder Q by eroding O with $\mathcal{V}(\mathbf{o}, \Delta)\ \forall \mathbf{o} \in \mathit{boundary}(O)$. If Q still blocks the line from $\mathbf{v}_0$ to $\mathbf{x}_0$ (see mark $\mathbf{y}_0$ in Figure 3), then O blocks any ray segment from $\mathbf{v}_1$ to $\mathbf{x}_0$ (see mark $\mathbf{y}_1$). Hence, we have identified an occlusion of $\mathbf{x}_0$ from $\mathcal{V}(\mathbf{v}_0, \Delta)$.

Décoret et al. [2003] further show that this occlusion actually holds for all *occludees* $\mathbf{x}_1 \in \mathcal{V}(\mathbf{x}_0, \Delta)$ as well (see mark $\mathbf{y}_2$). This observation implies that occludees can also be shrunk, reducing an occlusion test for a shaft connecting points in $\mathcal{V}(\mathbf{v}_0, \Delta)$ to points in $\mathcal{V}(\mathbf{x}_0, \Delta)$ to a line-segment-only occlusion test from $\mathbf{v}_0$ to $\mathbf{x}_0$.

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 $\mathbf{v}_0$ 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 $\mathbf{v}_0$ towards $\mathbf{s}_1$. The ray observes an occlusion interval between $\mathbf{f}$ and $\mathbf{b}$. For a view ray that grazes the silhouette $\mathbf{s}$, the occlusion interval collapses into a single point at $\mathbf{s}$.

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

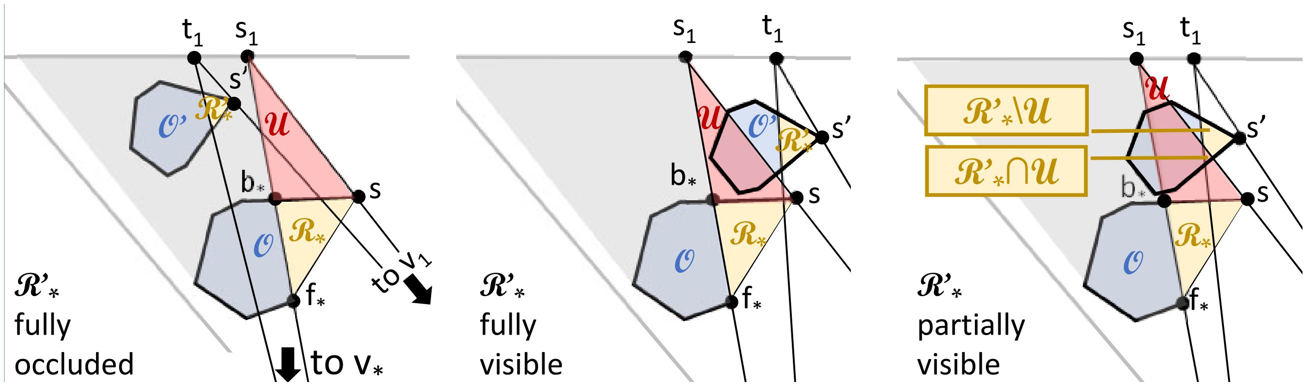

Fig. 5. (left) An occluder O spawns a trim region at silhouette $\mathbf{s}$. After removing the trim region, the remaining parts of O still occlude portions in the back (gray and red areas). A trim region (yellow) of the occludee O' at $\mathbf{s}'$ is entirely occluded by O and does not contribute to the PVS. (middle) No intersection with the umbra produces a fully disoccluded occludee, while an intersection (right) produces a partly disoccluded occludee.

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 $\mathbf{v}_0$, a view ray grazes the silhouette $\mathbf{s}$ and hits the far plane at $\mathbf{s}_0$. Rays aimed to the left of the silhouette encounter occlusion intervals, while rays to the right, such as from $\mathbf{v}_0$ to $\mathbf{s}_2$, do not hit the occluder.

Let the camera move away from the original view point $\mathbf{v}_0$ along a vector Δ to a new position $\mathbf{v}_1$. A ray emitted at $\mathbf{v}_1$ grazes $\mathbf{s}$ and hits the far plane at $\mathbf{s}_1$. Compared to $\mathbf{v}_0$, several rays emitted at $\mathbf{v}_1$ now reach the far plane behind the occluder between $\mathbf{s}_0$ and $\mathbf{s}_1$. We call this phenomenon *disocclusion around* $\mathbf{s}$, because of the role the silhouette $\mathbf{s}$ plays. The *penumbra* $\mathcal{P}$ is the disoccluded area formed by $\mathbf{s}$, $\mathbf{s}_0$ and $\mathbf{s}_1$, while the *umbra* $\mathcal{U}$ is the fully occluded area formed by $\mathbf{b}$, $\mathbf{s}$ and $\mathbf{s}_1$.

3.3 Trim regions

Our goal is to compute from-region visibility in a viewcell centered at $\mathbf{v}_0$ that supports camera translation by a distance of up to $|\Delta|$. This means that we must force the same disocclusion by shrinking O that a camera translation by Δ would cause.

Revisit Figure 4, left: We established that the disocclusion at silhouette $\mathbf{s}$ reveals $\mathcal{P}$. At position $\mathbf{v}_0$, the portion of the occluder delineated by $\mathbf{f}$, $\mathbf{b}$ and $\mathbf{s}$, which we call the *trim region* $\mathcal{R}$, is responsible for blocking view rays extending towards $\mathcal{P}$, while view rays extending from $\mathbf{v}_1$ towards $\mathcal{P}$ are not blocked. Consequently, if we remove $\mathcal{R}$ from the occluder, we obtain the visibility for $\mathbf{v}_1$ directly from testing view rays emitted at $\mathbf{v}_0$ 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 $\mathbf{v}_0$. Note that erosion with the trim region also covers all possible camera positions between $\mathbf{v}_0$ and $\mathbf{v}_1$, which we shall call the *viewcell* $\mathcal{V}$. Regardless of where we place the camera in $\mathcal{V}$, no view ray that grazes $\mathbf{s}$ can hit the far plane outside $\mathcal{P}$.

So far, we have only considered the effect of a translation by Δ. If the camera moves along $-\Delta$, the silhouette $\mathbf{s}$ becomes irrelevant, since it does not cause disocclusions. A disocclusion around $\mathbf{s}$ is only relevant if the angle between the normal $\mathbf{n}$ at $\mathbf{s}$ and the vector Δ is acute, *i.e*, $\mathbf{n} \cdot \Delta > 0$. Consequently, if we want to form a viewcell as a neighborhood around $\mathbf{v}_0$ that extends in arbitrary directions, we have to consider *all silhouettes* of the occluder observed from $\mathbf{v}_0$.

3.4 View frustum adaptation for viewcells

Moving the camera from $\mathbf{v}_0$ to any $\mathbf{v}_1$ necessarily changes the view frustum, so that portions of the scene clipped away for the original view frustum at $\mathbf{v}_0$ are now included in the new view frustum established at $\mathbf{v}_1$. As suggested by Wonka et al. [2001, Figure 6], a new frustum valid for all viewpoints $\mathbf{v} \in \mathcal{V}$ can be created by moving the viewpoint back to $\mathbf{v}_*$ by a distance z along the negative optical axis of the original frustum formed at $\mathbf{v}_0$ (Figure 4, middle). The distance $z = |\Delta|/\tan\alpha$ is a function of Δ and the subtended angle 2α 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 $\mathbf{v} \in \mathcal{V}$ with a single frustum at $\mathbf{v}_*$ requires adaptation of the trim region, as a camera at $\mathbf{v}_*$ observes different occlusion intervals. Figure 4, middle, shows a new view ray from $\mathbf{v}_*$ towards $\mathbf{s}_1$. As the silhouette $\mathbf{s}$ and its projection $\mathbf{s}_1$ remain unchanged, we must only form a new trim region $\mathcal{R}_*$

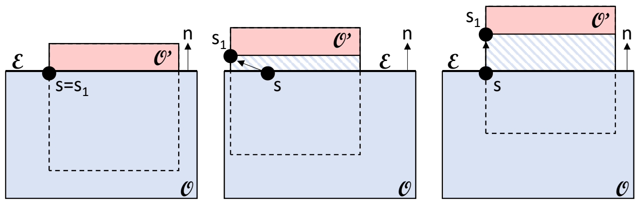

Fig. 6. Two objects are depicted from the camera's point of view. The blue object partially occludes the red object, with the silhouette edge $\mathcal{E}$ of the blue object forming the boundary in perspective space. As the camera is translated, a larger portion of the red object becomes disoccluded. The rate of disocclusion at $\mathcal{E}$ is highest in the direction of the edge normal n.

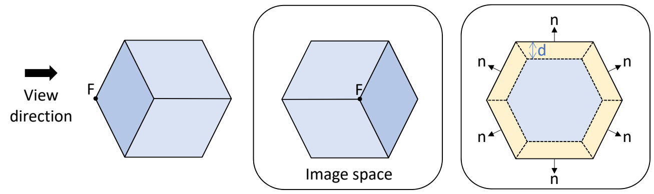

Fig. 7. (left) A cubical occluder seen from the side, (middle) the occluder as seen in image space, (right) trim regions (yellow) generated by displacing silhouette edges by d along their negative normal.

spanned by $\mathbf{s}, \mathbf{f}_*$ and $\mathbf{b}_*$. Depending on the location of $\mathbf{s}$, $\mathcal{R}_*$ can be larger or smaller than $\mathcal{R}$ or $\mathcal{R}_* = \emptyset$ 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 Δ which has its its apex at $\mathbf{v}_*$ and its base in the plane containing $\mathbf{v}_0$ and $\mathbf{v}_1$, and another cone mirrored around that plane. When computing the PVS from $\mathbf{v}_*$, we can freely move the camera within $\mathcal{V}$. If we only required that the viewpoint be at most Δ from $\mathbf{v}_0$, we could use a sphere centered at $\mathbf{v}_0$ with radius $|\Delta|$ as $\mathcal{V}$ (dashed blue semicircle in Figure 4, middle). However, we must also require that view rays emitted from within the viewcell which graze $\mathbf{s}$ must not enter $\mathcal{U}$ (as the blue view ray at $\mathbf{v}_2$ in Figure 4, middle). This is only guaranteed for viewpoints inside the double-cone viewcell. Previous work [Hladky et al. 2019a] 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 $\mathbf{v}_*$ be the viewpoint of the enlarged viewing frustum corresponding to a viewcell of diameter Δ. From $\mathbf{v}_*$, we determine the trimmed occluder $O_* = O \setminus \mathcal{R}_*$ and find the corresponding fully occluded region $\mathcal{U}$, enclosed by $\mathbf{s}, \mathbf{s}_1$ and $\mathbf{b}_*$ (Figure 4).

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

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

If $\mathcal{U}$ partially intersects $\mathcal{R}'_*$, we require a more complex geometric analysis. Recall that we have made sure that, even if we trim O by $\mathcal{R}_*$, no view ray from $\mathbf{v}_1$ can enter $\mathcal{U}$. Consequently, any portion of O' that lies within $\mathcal{U}$ *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 $\mathbf{s}'$.

Consider the example in Figure 5, right. The occluder silhouette $\mathbf{s}$ generates a trim region and an associated umbra. A part of the occludee trim region at silhouette $\mathbf{s}'$ overlaps the umbra of $\mathbf{s}$. Clearly, parts of the trim region for $\mathbf{s}'$ cannot be observed from $\mathbf{v}_1$, since they are inside the umbra of $\mathbf{s}$. Consequently, we must not discard the occlusion intervals for those rays. However, we must not classify the region $\mathcal{R}'_* \cap \mathcal{U}$ as visible, either.

The remaining portion of the trim region at $\mathbf{s}'$, *i.e*, $\mathcal{R}'_* \setminus \mathcal{U}$, is retained and used to trim O'. Trimming O' with respect to $\mathcal{R}'_* \setminus \mathcal{U}$ (rather than just $\mathcal{R}'_*$) 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 $O' \setminus (\mathcal{R}'_* \setminus \mathcal{U})$ 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 $\mathbf{f}$ and the farthest $\mathbf{b}$ 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 $\mathbf{s}$. 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 Δ 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 $\mathcal{E}$ of O.

Consider the effect of a camera translation Δ on the relation between the two objects. The projection $\mathbf{s}_1$ of a point $\mathbf{s}$ on $\mathcal{E}$ to the far clipping plane moves dependent on the direction of Δ. After projection to the 2D image plane, a maximum disocclusion (farthest distance between $\mathbf{s}$ and $\mathbf{s}_1$) is obtained when aligning Δ with the normal $\mathbf{n}$ of the silhouette edge $\mathcal{E}$ that points away from the occluder. With this observation, we obtain the direction $-\mathbf{n}$ in which to displace $\mathcal{E}$ 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 $\mathbf{v}_*$ 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

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

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)

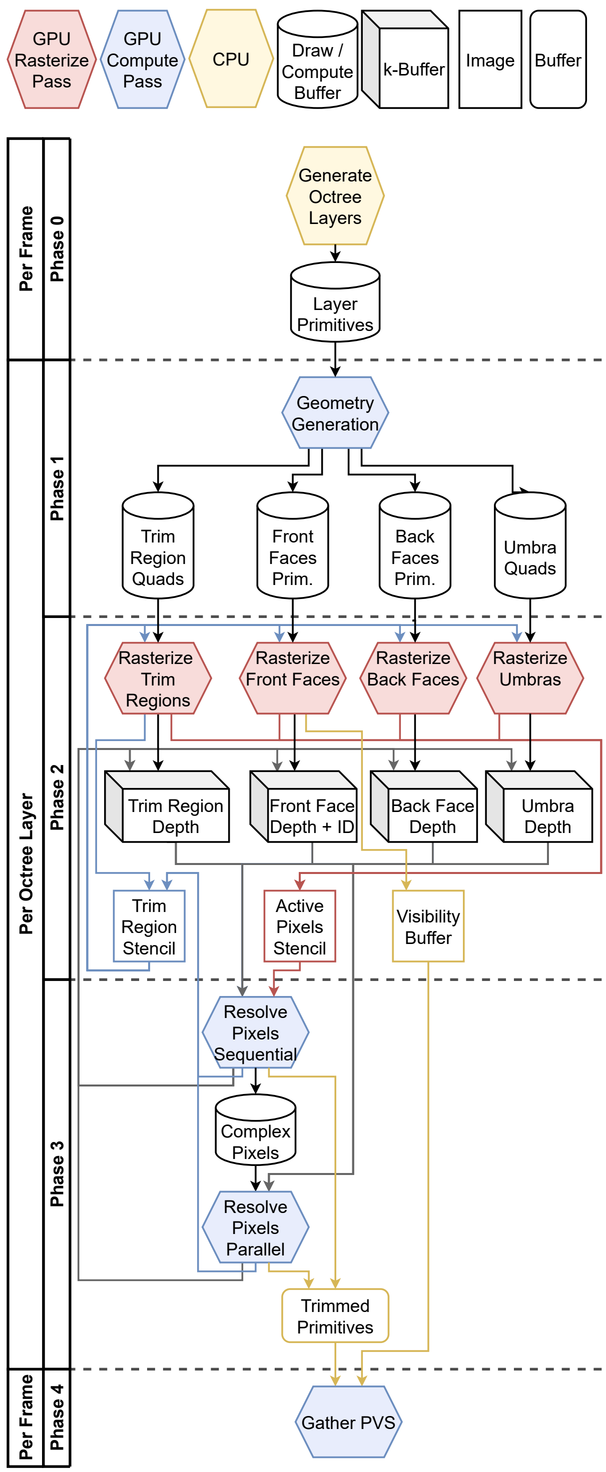

Fig. 8. The trim region algorithm commences in five phases: P1 determines the octree layers on the GPU. P1 generates the draw buffers for each layer, and P2 renders them. P3 resolves the resulting trim sequences. P4 gathers the results into the final PVS.

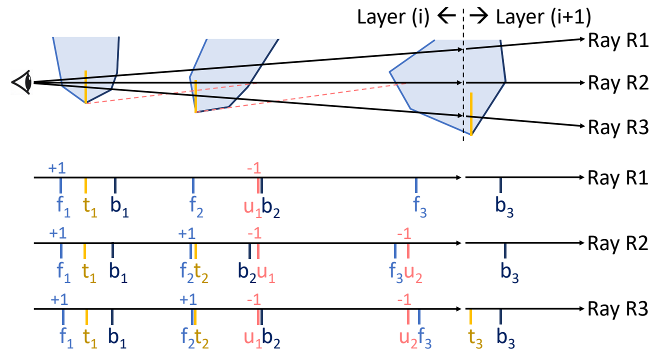

Fig. 9. Example trim sequences for three view rays. The top part of the figure shows the geometric arrangement, while the bottom part shows the trim sequences according to the depth values in clip space (the optical axis in the example is assumed to be aligned with R2). **R1**: A validated interval $f_1/t_1/b_1/u_1$ is found $\longrightarrow f_1$ is added to the PVS; f_2 is ignored; b_2 is classified as a terminator face $\longrightarrow f_3$ is ignored. **R2**: Validated intervals $f_1/t_1/b_1/u_1$ and $f_2/t_2/b_2/u_2$ are found $\longrightarrow f_1$ is added to the PVS; f_2 and f_3 are ignored; no terminator face is found in layer (i), but b_3 is classified as terminator face in layer $(i+1)$. **R3**: Like R2, but f_3 behind u_2 at the sequence trail is classified as a possible frontface of a validated interval and carried over to layer $(i+1)$, where a validated interval $f_3/t_3/b_3$ is found.

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_*. In our experiments, including all edges that could potentially become silhouettes within the viewcell 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 frontfacing or backfacing, 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 frontface 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 targets 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

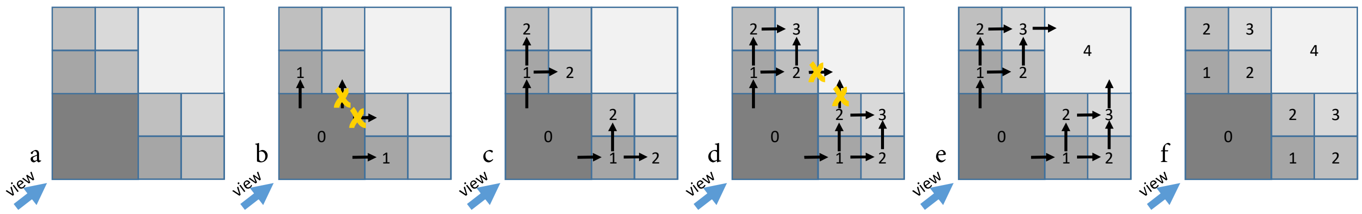

Fig. 10. An example of more complex traversal in the octree. (a) We select the start node based on the view direction. (b) From the start node, we move horizontally and vertically. We need to determine the appropriate neighbor nodes that would block other direct neighbors and process those first. (c) We proceed with traversal, until we arrive at another ambiguous location. (d) We need to ensure not to enter nodes prematurely, *i.e*, nodes that must wait for additional dependencies to be processed first. (e) All required neighbors have been processed. (f) We can move on and arrive at a final layer ordering.

rasterization of frontfacing primitives, we store not only depth, but also primitive id in the k-buffer. Moreover, the frontfaces 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 frontfaces or backfaces. 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 backfaces 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 frontface 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 frontface or backface that is not contained in a validated interval (prefix sum is zero). Since such a terminator face represents untrimmed geometry; it occludes 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 frontface 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 frontface/backface 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 termination candidate, the residual trim sequence is simply left in place to be resolved in subsequent layers. In particular, an unfinished interval (a frontface 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

Fig. 11. We used five test scenes in our evaluation, which are shown here with primitive counts

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 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 $2\alpha = 90°$. Hence, for a forward motion of 5 cm per frame, a PVS with a viewcell size of Δ is valid for a segment of $|\Delta|/5$ frames.

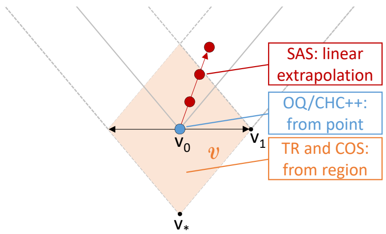

Fig. 12. The viewcells of all methods in our evaluation. While TR and COS cover volumetric viewcells, SAS (red) linearly extrapolates the viewpoint based on movement trajectories. Occlusion query methods like CHC++ are from-point methods and compute a visible set from a single point v_0.

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.

OQ is a from-viewpoint method which relies on testing bounding volumes of scene portions against the depth buffer by submitting occlusion queries to the GPU. The CHC++ method [Mattausch et al. 2008] is an improved OQ method that heuristically aggregates these calls to reduce the number of draw calls that require CPU-GPU synchronization. We have implemented a version of OQ that uses our octree peeling instead of CHC++-like heuristics for scheduling the queries. Like CHC++, occlusion queries are submitted for boxes enclosing the octree nodes. However, in our version, only a minimal number of queries is issued, since a whole octree layer can be aggregated into one query. The octree depth was empirically set to eight to obtain a PVS tightness comparable to that delivered by TR. Since OQ is a from-viewpoint method, it does not create a from-region PVS for comparison. Therefore, we report the runtime of its from-viewpoint computation.

Table 2. The k-buffer memory requirements for Trim Regions at a resolution of 1920x1080. 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 managable even for larger viewcell sizes due to our early stopping and divide-and-conquer strategies.

Scene	$\|\Delta\|$	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 temple	5	551648.7	16.2	336.7/ 55.8
	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

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.

Figure 14 reports the overall runtime of TR compared to its competitors. For SAS, we chose the sample rate at one EVS sample for every 5 cm of viewcell size. As can be expected, the runtime is roughly linear to the number of EVS samples and the scene size, and, overall, very fast. However, as will be discussed below, the simplistic predict-and-sample strategy of SAS is unable to support large viewcell sizes without severe loss of quality.

Our estimated comparison with the runtimes reported for COS on Viking village and Robot lab indicates a significant advantage of TR over COS in terms of speed. COS can solve 4D visibility globally with

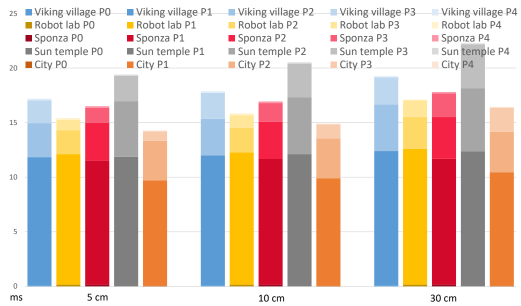

Fig. 13. Runtime of TR for various scenes and viewcell sizes $|\Delta|$, reported in ms, broken down by algorithm phase, where P0 is octree layer computation, P1 is geometry generation (which includes the layer-to-layer synchronization overhead), P2 is geometry rasterization, P3 is occlusion interval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

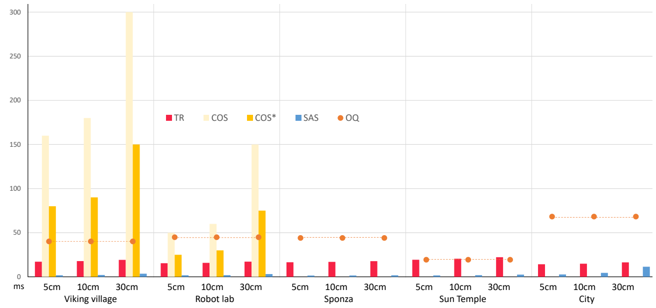

Fig. 14. Runtime comparison for various methods, scenes and viewcell sizes $|\Delta|$, reported in ms. "COS*" (yellow) is the time of "COS" (light yellow) scaled by 0.5× to estimate runtime on the modern RTX 4090 GPU used in our experiments (data only available for Viking village and Robot lab). "OQ"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

virtually no errors. Alas, its expensive linked-list representation lets memory consumption grow rapidly, and performance deteriorates rapidly as the viewcell 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 viewcell 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 viewcell 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 viewcell 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 viewcell by dense sampling, i.e., we compute

Table 3. "FN" and "FN*" indicate the rate of false negative primitives without and with suppression of unreliable primitives, respectively.

	Viking Village		Robot Lab		Sponza		Sun Temple		City	
$\lvert\Delta\rvert$	**FN**	**FN***	**FN**	**FN***	**FN**	**FN***	**FN**	**FN***	**FN**	**FN***
5	.031	.013	.048	.046	.029	.023	.015	.013	.044	.024
10	.031	.014	.047	.044	.030	.024	.017	.015	.043	.023
30	.037	.019	.050	.047	.048	.038	.032	.028	.040	.022

the union of primitives contained in 200 EVS samples uniformly spaced in a given viewcell. 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 viewcell size is still acceptable: A larger viewcell size means that the PVS remains valid longer, but at the prize 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 viewcell 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

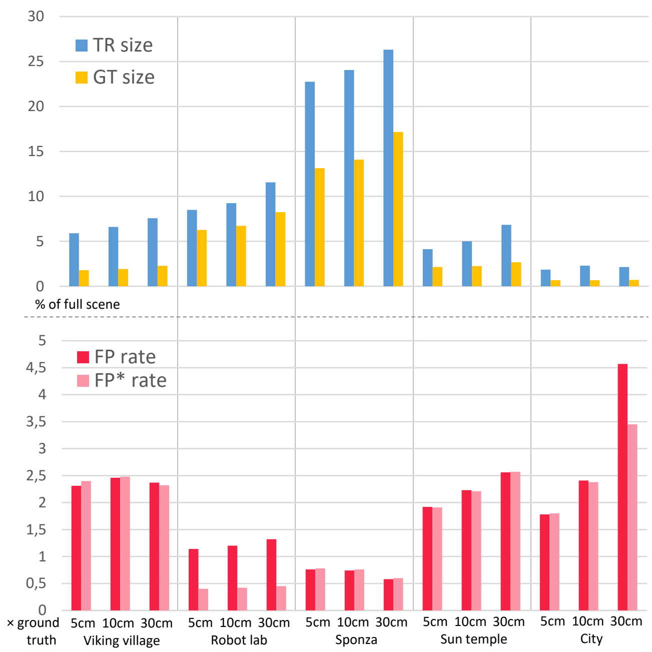

Fig. 15. Average PVS size tightness for various scenes and viewcell sizes $|\Delta|$. In the top chart, "TR size" and "GT size" indicate the percentage of the full scene placed in the PVS of the trim region method and the ground truth, respectively. In the bottom chart, "FP rate" and "FP* rate" indicate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

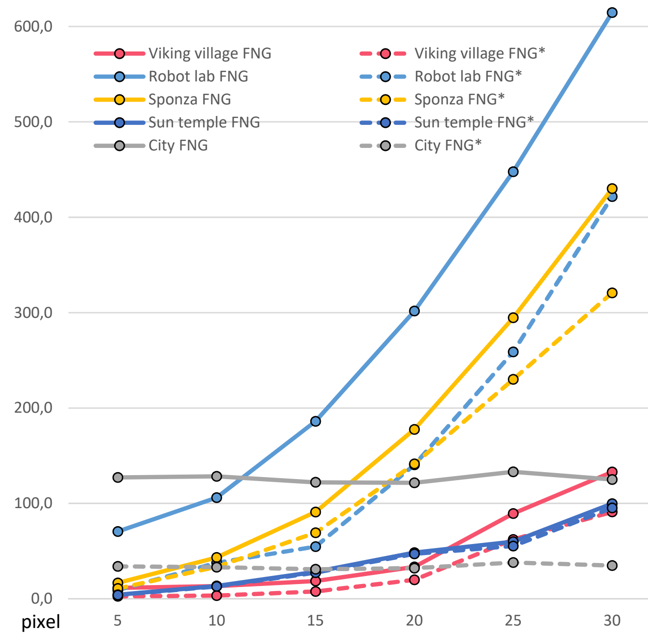

Fig. 16. Average false negative pixel count per frame for all possible views. FNG (solid lines) show results with unreliable triangles; FNG^* (dashed lines) shows results without unreliable triangles. When applying the cut-off suggested by Wonka et al. [2006] of roughly 100 pixels, a viewcell size of 15-20 cm can usually be supported without any noticeable errors.

of the City scene has a similar effect: Triangles of far away buildings project onto few (if any) pixels, making them unreliable.

We see that the ground-truth PVS size depends on the viewcell 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× 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× 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 $\mathcal{U}$ and $\mathcal{P}$ 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 ≤0.005%

Fig. 17. Challenging situations in Robot lab (top row, 210 false negative pixels) and Sun temple (bottom row, 344 false negative pixels). In each row, the left column shows the full scene; the middle column shows the PVS computed by TR, and the right hand side highlights false negative pixels in red. The differences to the full scene are not immediately apparent in the PVS rendering.

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 FNG, 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 (FNG) and once without (FNG^*). Table 3 lists the false negative primitive rates FN/GTC and FN^*/GTC^*.

The data for FNG 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 FN 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.

REFERENCES

M Airey, J Rohlf, and Frederick Brooks Jr. 1990. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments". *ACM SIGGRAPH Computer Graphics* 24 (1990), 41–50. https://doi.org/10.1145/91385.91416

Fabien Benichou and Gershon Elber. 1999. Output Sensitive Extraction of Silhouettes from Polygonal Geometry. In *Proc. Pacific Graphics*.

J. Bittner, V. Havran, and P. Slavik. 1998. Hierarchical visibility culling with occlusion trees. In *Proceedings. Computer Graphics International (Cat. No.98EX149)*. IEEE Comput. Soc, 207–219. https://doi.org/10.1109/CGI.1998.694268

Jiri Bittner, Oliver Mattausch, Peter Wonka, Vlastimil Havran, and Michael Wimmer. 2009. Adaptive global visibility sampling. In *ACM SIGGRAPH '09*, Vol. 28. ACM Press, New York, New York, USA, 1. https://doi.org/10.1145/1576246.1531400

Jiri Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. 2004. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. *Computer Graphics Forum* 23, 3 (9 2004), 615–624.

Jiri Bittner and Peter Wonka. 2003. Visibility in Computer Graphics. *Environment and Planning B* 30, 5 (10 2003), 729–755. https://doi.org/10.1068/b2957

Jiri Bittner, Peter Wonka, and Michael Wimmer. 2005. Fast exact from-region visibility in urban scenes. *Proceedings of the Sixteenth Eurographics conference on Rendering Techniques* (2005), 223–230. https://doi.org/10.2312/egwr/egsr05/223-230

Christopher A Burns and Warren A Hunt. 2013. The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading. *Journal of Computer Graphics Techniques (JCGT)* 2 (8 2013), 55–69. Issue 2. http://jcgt.org/published/0002/02/04/

Anish Chandak, Lakulish Antani, Micah Taylor, and Dinesh Manocha. 2009. FastV: From-point Visibility Culling on Complex Models. In *Proc. of the 20th Eurographics Conference on Rendering (EGSR'09)*. Eurographics Association, 1237–1246. https://doi.org/10.1111/j.1467-8659.2009.01501.x

C. Chandrasekaran, D. McNabb, D. Kuah, M. Fauconneau, and F. Giesen. 2016. Software Occlusion Culling. Published online, last visited 2019-01-15.. https://software.intel.com/en-us/articles/software-occlusion-culling

D. Cohen-Or, Y.L. Chrysanthou, C.T. Silva, and F. Durand. 2003. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics* 9, 3 (7 2003), 412–431. https://doi.org/10.1109/TVCG.2003.1207447

Daniel Cohen-Or, Gadi Fibich, Dan Halperin, and Eyal Zadicario. 1998. Conservative Visibility and Strong Occlusion for Viewspace Partitioning of Densely Occluded Scenes. *Computer Graphics Forum* 17, 3 (1998), 243–253. https://doi.org/10.1111/1467-8659.00271

D. Collin. 2011. Culling the Battlefield. Talk at Game Developer's Conference.

W.T. Correa, J.T. Klosowski, and C.T. Silva. 2003. Visibility-based prefetching for interactive out-of-core rendering. In *IEEE Sensors Journal*. IEEE, 1–8. https://doi.org/10.1109/PVGS.2003.1249035

Xavier Décoret, Gilles Debunne, and François Sillion. 2003. Erosion Based Visibility Preprocessing. In *Proc. of the 14th Eurographics Workshop on Rendering (EGRW '03)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 281–288.

Mark Duchaineau, Murray Wolinsky, David E Sigeti, Mark C Miller, Charles Aldrich, and Mark B Mineev-Weinstein. 1997. ROAMing Terrain: Real-time Optimally Adapting Meshes. In *Proceedings of the 8th Conference on Visualization '97 (VIS '97)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 81–88.

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

Frédo Durand, George Drettakis, Joëlle Thollot, and Claude Puech. 2000. Conservative visibility preprocessing using extended projections. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH '00*. ACM Press, New York, New York, USA, 239–248. https://doi.org/10.1145/344779.344891

Epic Games. 2022. Precomputed Visibility Volumes. https://docs.unrealengine.com/5.1/en-US/precomputed-visibility-volumes-in-unreal-engine/ Visited 21/12/2022..

Sebastian Freitag, Benjamin Weyers, and Torsten W. Kuhlen. 2017. Efficient approximate computation of scene visibility based on navigation meshes and applications for navigation and scene analysis. In *2017 IEEE Symposium on 3D User Interfaces (3DUI)*. IEEE, 134–143. https://doi.org/10.1109/3DUI.2017.7893330

Ned Greene. 1995. *Hierarchical Rendering of Complex Environments*. Ph. D. Dissertation.

Ned Greene, Michael Kass, and Gavin Miller. 1993. Hierarchical Z-buffer visibility. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques - SIGGRAPH '93*. ACM Press, New York, New York, USA, 231–238. https://doi.org/10.1145/166117.166147

Ulrich Haar and Sebastian Aaltonen. 2015. GPU-Driven Rendering Pipelines. SIGGRAPH Course: Advances in Real-Time Rendering in Games.

Jon Hasselgren, Magnus Andersson, and Tomas Akenine-Möller. 2016. Masked Software Occlusion Culling. In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. https://doi.org/10.2312/hpg.20161189

Francis S. Hill and Stephen M Kelley. 2006. *Computer Graphics Using OpenGL (3rd Edition)*. Prentice-Hall, Inc., USA.

Stephen Hill and Daniel Collin. 2011. Practical, Dynamic Visibility for Games. GPU Pro 2. , 329–347 pages.

Jozef Hladky, Hans-Peter Seidel, and Markus Steinberger. 2019a. The Camera Offset Space: Real-time Potentially Visible Set Computations for Streaming Rendering. *ACM Trans. Graph.* 38, 6, Article 231 (Nov. 2019), 14 pages.

Jozef Hladky, Hans-Peter Seidel, and Markus Steinberger. 2019b. Tessellated Shading Streaming. *Computer Graphics Forum* (2019).

Lichan Hong, Shigeru Muraki, Arie E Kaufman, Dirk Bartz, and Taosong He. 1997. Virtual voyage: interactive navigation in the human colon. In *Proc. of the 24th Annual Conference on Computer Graphics and Interactive Techniques, {SIGGRAPH} 1997, Los Angeles, CA, USA, August 3-8, 1997*. 27–34. https://doi.org/10.1145/258734.258750

T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. 1997. Accelerated occlusion culling using shadow frusta. In *Proceedings of the Annual Symposium on Computational Geometry*. ACM, 1–10. https://doi.org/10.1145/262839.262847

Vladlen Koltun, Yiorgos Chrysanthou, and Daniel Cohen-Or. 2000. Virtual Occluders: An Efficient Intermediate PVS representation. Springer, Vienna, 59–70. https://doi.org/10.1007/978-3-7091-6303-0{_}6

Vladlen Koltun, Yiorgos Chrysanthou, and Daniel Cohen-Or. 2001. Hardware-accelerated from-region visibility using a dual ray space. Springer, Vienna, 205–215. https://doi.org/10.1007/978-3-7091-6242-2{_}19

Samuli Laine. 2005. A general algorithm for output-sensitive visibility preprocessing. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games - SI3D '05*. ACM Press, New York, New York, USA, 31. https://doi.org/10.1145/1053427.1053433

Sungkil Lee, Younguk Kim, and Elmar Eisemann. 2018. Iterative Depth Warping. *ACM Trans. Graph.* 37, 5, Article 177 (Oct. 2018), 13 pages. https://doi.org/10.1145/3190859

Tommer Leyvand, Olga Sorkine, and Daniel Cohen-Or. 2003. Ray space factorization for from-region visibility. In *ACM SIGGRAPH 2003 Papers on - SIGGRAPH '03*, Vol. 22. ACM Press, New York, New York, USA, 595. https://doi.org/10.1145/1201775.882313

Oliver Mattausch, Jiri Bittner, and Michael Wimmer. 2006. Adaptive visibility-driven view cell construction. *Proceedings of the 17th Eurographics conference on Rendering Techniques* (2006), 195–205. https://doi.org/10.2312/egwr/egsr06/195-205

Oliver Mattausch, Jiri Bittner, and Michael Wimmer. 2008. CHC++: Coherent Hierarchical Culling Revisited. *Computer Graphics Forum* 27, 2 (4 2008), 221–230.

Joerg H Mueller, Philip Voglreiter, Mark Dokter, Thomas Neff, Mina Makar, Markus Steinberger, and Dieter Schmalstieg. 2018. Shading Atlas Streaming. *ACM Transactions on Graphics* 37, 6 (11 2018). https://doi.org/10.1145/3272127.3275087

S. Nirenstein and E. Blake. 2004. Hardware accelerated visibility preprocessing using adaptive sampling. *Proceedings of the Fifteenth Eurographics conference on Rendering Techniques* (2004), 207–216. https://doi.org/10.2312/egwr/egsr04/207-216

E. Persson. 2012. Creating Vast Game Worlds: Experiences from Avalanche Studios. SIGGRAPH Talks.

Gernot Schaufler, Julie Dorsey, Xavier Decoret, and François X. Sillion. 2000. Conservative volumetric visibility with occluder fusion. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH '00*. ACM Press, New York, New York, USA, 229–238. https://doi.org/10.1145/344779.344886

Yvens Rebouças Serpa and Maria Andréia Formico Rodrigues. 2019. A draw call-oriented approach for visibility of static and dynamic scenes with large number of triangles. *Visual Computer* 35, 4 (apr 2019), 549–563.

Shu Shi and Cheng-Hsin Hsu. 2015. A Survey of Interactive Remote Rendering Systems. *ACM Comput. Surv.* 47, 4, Article 57 (May 2015).

Jeremy Shopf, Joshua Barczak, Christopher Oat, and Natalya Tatarchuk. 2008. March of the Froblins: Simulation and rendering massive crowds of intelligent and detailed creatures on GPU. In *ACM SIGGRAPH Courses*. 52–101. https://doi.org/10.1145/1404435.1404439

Seth J. Teller and Carlo H. Séquin. 1991. Visibility preprocessing for interactive walkthroughs. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques - SIGGRAPH '91*, Vol. 25. ACM Press, New York, New York, USA, 61–70. https://doi.org/10.1145/122718.122725

N. Thomos, N.V. Boulgouris, and M.G. Strintzis. 2006. Optimized transmission of JPEG2000 streams over wireless channels. *IEEE Transactions on Image Processing* 15, 1 (2006), 54–67. https://doi.org/10.1109/TIP.2005.860338

Alex Wilson, Ashley Miller, Martin Matthews, and Shirley Stevens. 2023. 2023 GPU Benchmark and Graphics Card Comparison Chart. https://www.gpucheck.com/gpu-benchmark-graphics-card-comparison-chart Visited on January 17, 2023..

Peter Wonka, Michael Wimmer, and Dieter Schmalstieg. 2000. Visibility Preprocessing with Occluder Fusion for Urban Walkthroughs. Springer, Vienna, 71–82. https://doi.org/10.1007/978-3-7091-6303-0{_}7

Peter Wonka, Michael Wimmer, and Francois X. Sillion. 2001. Instant Visibility. *Computer Graphics Forum* 20, 3 (9 2001), 411–421. https://doi.org/10.1111/1467-8659.00534

Peter Wonka, Michael Wimmer, Kaichi Zhou, Stefan Maierhofer, Gerd Hesina, and Alexander Reshetov. 2006. Guided visibility sampling. In *ACM SIGGRAPH 2006*, Vol. 25. ACM Press, New York, USA, 494. https://doi.org/10.1145/1179352.1141914