The Camera Offset Space: Real-time Potentially Visible Set Computations for Streaming Rendering

JOZEF HLADKY and HANS-PETER SEIDEL, Max-Planck-Institut für Informatik
MARKUS STEINBERGER, Graz University of Technology

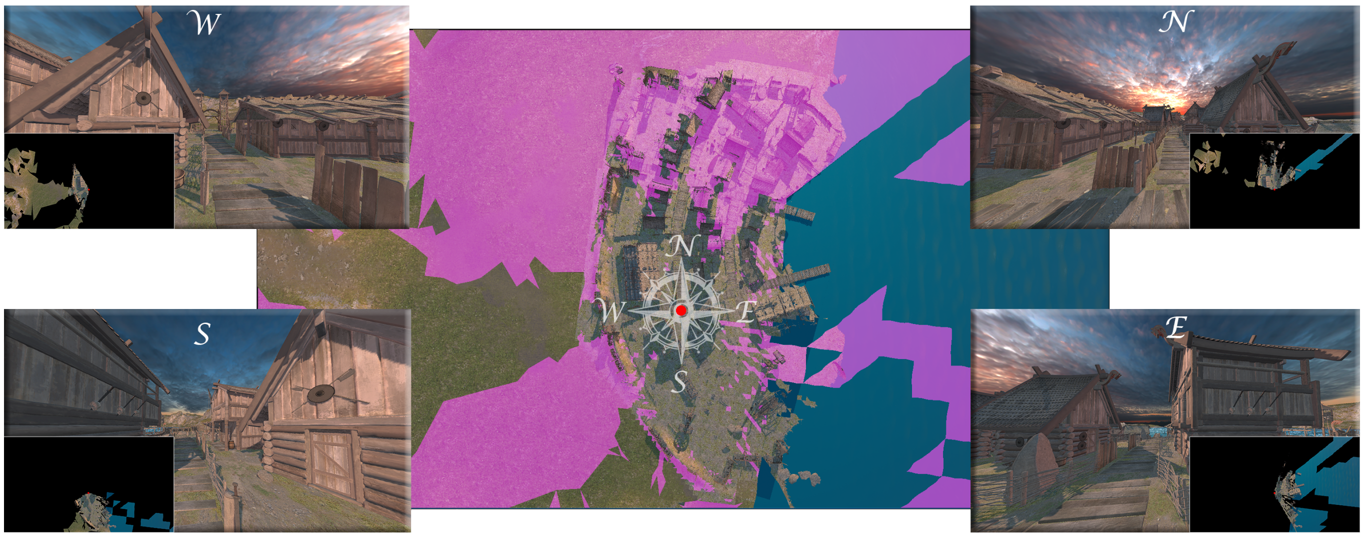


Fig. 1. A birds-eye view on a full 360 degree potentially visible set (PVS) computed for a region around the camera from four reference views (in the corners) in an outdoor scene. Geometry labeled as invisible shown in magenta color, the red dot shows camera position in the scene. To construct the PVS, we collect all triangles that can cover a fragment in a per-fragment list and resolve the visibility in the proposed camera offset space. Each reference view contains a corner inset depicting its corresponding PVS—starting from top right, clockwise: North, East, South and West. The final PVS is constructed by joining these 4 sets. Novel views rendered with the PVS for view points around the reference location are complete and hole free.

Potential visibility has historically always been of importance when rendering performance was insufficient. With the rise of virtual reality, rendering power may once again be insufficient, e.g., for integrated graphics of head-mounted displays. To tackle the issue of efficient potential visibility computations on modern graphics hardware, we introduce the camera offset space (COS). Opposite to how traditional visibility computations work—where one determines which pixels are covered by an object under all potential viewpoints—the COS describes under which camera movement a sample location is covered by a triangle. In this way, the COS opens up a new set of possibilities for visibility computations. By evaluating the pairwise relations of triangles in the COS, we show how to efficiently determine occluded triangles. Constructing the COS for all pixels of a rendered view leads to a complete potentially visible set (PVS) for complex scenes. By fusing triangles to larger occluders, including locations between pixel centers, and considering camera rotations, we describe an exact PVS algorithm that includes all viewing directions inside a view cell. Implementing the COS is a combination of real-time rendering and compute steps. We provide the first GPU PVS implementation that works without preprocessing, on-the-fly, on unconnected triangles. This opens the door to a new approach of rendering for virtual reality head-mounted displays and server-client settings for streaming 3D applications such as video games.

CCS Concepts: • **Computing methodologies** → **Visibility**; *Rasterization*; • **Theory of computation** → *Massively parallel algorithms.*

Additional Key Words and Phrases: Visibility, Potentially Visible Set, Streaming Rendering, Virtual Reality, Real-time Rendering, GPU

Authors' addresses: Jozef Hladky, jhladky@mpi-inf.mpg.de; Hans-Peter Seidel, hpseidel@mpi-sb.mpg.de, Max-Planck-Institut für Informatik, Saarland Informatics Campus, Campus E 1 4, 66123, Saarbrücken, Germany; Markus Steinberger, Graz University of Technology, Inffeldgasse 16/II, 8010, Graz, Austria, steinberger@icg.tugraz.at.

0730-0301/2019/11-ART231 $15.00
https://doi.org/10.1145/3355089.3356530

1 INTRODUCTION

In real-time graphics, we face an ever-increasing demand for more processing power, as the video game industry aims to provide photo-realistic real-time 3D visuals, pushing current graphics processing units (GPUs) to the limit. The rising popularity of thin, lightweight, untethered devices, like head-mounted displays (HMDs) for virtual reality (VR) applications or smartphones and tablets for gaming adds further constraints. These devices operate with limited resources (voltage/on-device memory/processing power) in comparison to workstation and server-grade hardware.

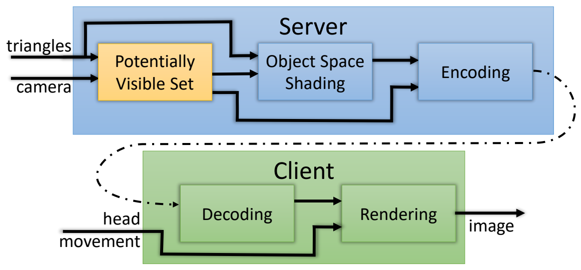


Fig. 2. Our streaming rendering pipeline is split between a server and a client device. The server performs a real-time PVS computation, object space shading and encoding of the data. The client takes the preshaded PVS triangles and synthesizes novel views with high frame rates. In this paper we focus on the PVS computation (shown in yellow).

Especially in VR, compute power is crucial. In order to provide an immersive VR experience while using an HMD, an application must provide high visual fidelity at very high framerates in high resolutions for both eyes. Should the application lack responsiveness, immersiveness degrades and motion sickness can occur. As such, VR applications on untethered HMDs are among the most demanding and most constrained use cases for 3D rendering.

To compensate for the low compute power of untethered HMDs, the HMDs can be combined with a remote server, *e.g.*, in a cloud or a Wi-Fi connected desktop computer. However, Wi-Fi connections increase the rendering latency by up to 100 ms, and wide-area connections up to 200 ms [Huang et al. 2012]. In combination with the previously outlined challenges, untethered HMDs form a very challenging streaming rendering problem, in which a powerful machine (*server*) streams content to a device (*client*).

While image-based rendering techniques are usually used for streaming rendering [Shi and Hsu 2015], they cannot faithfully handle disocclusion, high-frequency information, and large depth discontinuities—especially when latency increases. To overcome these issues, we opt for a split rendering pipeline, illustrated in Fig. 2: A server preprocesses and shades those triangles *potentially needed* for rendering in the near future on the client. Using these data, the client independently creates renderings for the current HMD view. This approach entails a series of new challenges:

(1) The server needs to identify which triangles may potentially be needed by the client considering the system's current state, the current view of the user, and potential user movement.
(2) The server needs to provide shading information for all triangles that may become visible under any user movement before the next shading information is received.
(3) The transmitted data, which may bear little resemblance with a video stream, must be encoded efficiently for transmission.
(4) The client must be able to generate high quality views from the available data, considering moving objects and appearance changes due to different viewing angles.

While solutions to all challenges are necessary, in this paper we focus on challenge (1), which describes a variant of the classical potentially visible set (PVS) problem: identifying all objects (or triangles) that may become visible under a given set of user movements. Working on individual triangles instead of objects reduces the bandwidth required for transmission. The traditional approach to this problem has been preprocessing, for which the entire scene is subdivided into viewing cells and the visible objects for each cell are determined and stored [Cohen-Or et al. 2003]. Such an approach is only feasible for static scenes and usually works only on entire objects with relatively large viewing cells. With the goal of minimizing the number of shaded and transmitted triangles while considering dynamic environments, we propose an approach to real-time potentially visible set creation from a 3D region that runs *entirely on the GPU*, does *not require any preprocessing* and only requires the *triangle data as used for rendering*.

To tackle this task, we view visibility computations from a different angle. PVS approaches are usually concerned with the question "*Which screen area* is covered by a single object (occluder) under all view points inside the view cell?" Then, they determine whether there are other objects (occludees) that always stay within that region and thus are never visible. Instead of trying to identify entire regions covered by an occluder, we reverse the question and ask "*Under which camera offsets* is a single screen location covered by an object?". To this end, we make the following contributions:

- We introduce the *camera offset space*, which, for a given point on the image plane, provides information under which camera offset the point is covered by a given triangle.
- By constructing the camera offset space for every pixel location, we show how the entire PVS for a rasterized image can be computed for translational camera movement.
- By elevating the camera offset space by two additional dimensions, we show how it can be used to compute the PVS for arbitrary sample locations and rotational camera movement.
- We provide a parallel formulation of the complete PVS computation suitable for current GPUs using a combination of rendering pipeline stages and compute mode execution.

We derive the mathematics behind the *camera offset space* (COS) by describing a single triangle in the COS (Section 3.1). Using this description, we consider the visibility between *pairs of triangles* (Section 3.4), which we extend to the visibility of *all objects at a single pixel* (Section 4). Performing this progressive visibility resolution in parallel for *all pixels* already yields a reasonable PVS algorithm that we effectively integrate into the rendering pipeline. However, camera rotations that move sample locations may uncover triangles not contained in this PVS. To include those and allow visibility computations at a lower resolution, we include in-between sample locations in the COS (Section 4.5). To evaluate our method, we show that our approach can compute the PVS for various scenes with different characteristics (Section 6).

Note that a similar streaming pipeline has recently been introduced in Shading Atlas Streaming [Mueller et al. 2018], which focuses on collecting and transmitting shading data in an atlas to a thin-client HMD. Their system lacks a sophisticated PVS algorithm and simply uses a reference view to sample visibility. Thus, their approach misses small triangles, creating holes and flickering artifacts in the final client rendering. Our approach solves these issues and could be used as a drop-in replacement in their system.

2 RELATED WORK

The outlined system relates to many topics, including PVS generation, cloud rendering, encoding and transmission, alternative shading approaches and image-based rendering.

2.1 Visibility Computations

Potentially visible set computation and occlusion culling have been very active research topics decades ago [Cohen-Or et al. 2003], when reducing the number of objects drawn was essential to achieve high refresh rates. Facing a situation where the available rendering power is again insufficient, PVS algorithms may resurface.

Exact Visible Set. A multitude of algorithms for *exact visible set* (EVS) computations for a single view point exist. Among the most well known are cells and portal approaches [Airey et al. 1990; Jiménez et al. 2000; Teller and Hanrahan 1993], where the visible set is computed for a position in a room with limited view to other rooms. The technique can be run online by culling portals against each other [Hong et al. 1997; Luebke and Georges 1995]. 3D planes depicting the change of occlusion status can be used for large convex occluders [Coorg and Teller 1999]. Similarly, one can view the occlusion problem as a shadow frustra problem [Hudson et al. 1997], use binary space partitioning (BSP) trees [Bittner et al. 1998], use a hierarchical representation of occluders [Zhang et al. 1997], or use hardware occlusion queries [Bittner et al. 2004; Govindaraju et al. 2003]. Motion prediction can also improve the results [Correa et al. 2003]. However, an EVS is not sufficient for streaming rendering and a complete PVS is required.

Potentially Visible Set. A PVS for a region is significantly more complex than from-point visibility. For indoor scenes, an exact PVS is possible using preprocessing [Funkhouser 1996; Teller and Séquin 1991]. For general scenes, approximative sampling can be employed [Gotsman et al. 1999] and conservative methods can employ occluder fusion [Schaufler et al. 2000]. Considering the projected area an occluder covers under all points within the region, other objects can be classified as invisible if they stay within the fused projection [Durand et al. 2000]. Similarly, the PVS can conservatively be estimated when taking point samples and shrinking occluders by an ϵ value which corresponds to the distance between the point sample locations [Wonka et al. 2000, 2001]. Alternatively, sampling can be employed to approximate the PVS [Bittner et al. 2009; Koltun et al. 2001; Leyvand et al. 2003]. While sampling may become attractive with GPU raytracing, computation speed is still an issue, as the synergy in global sampling approaches [Bittner et al. 2009] becomes a non-factor in dynamic environments. Furthermore, sampling errors may lead to significant flickering in virtual reality scenarios.

Unfortunately, all previously mentioned PVS algorithms are not suitable for dynamic scenes, do not run online, require preprocessing, or work on entire objects only. Thus, none are applicable for the use in a streaming virtual reality scenario. Ideally, we want a conservative PVS that is real-time and provides visibility information on the level of individual triangles. Only then, the number of shaded and transmitted triangles is reduced to a minimum while holes are still avoided during rendering on the client. To the best of our knowledge, there is no technique that meets these requirements.

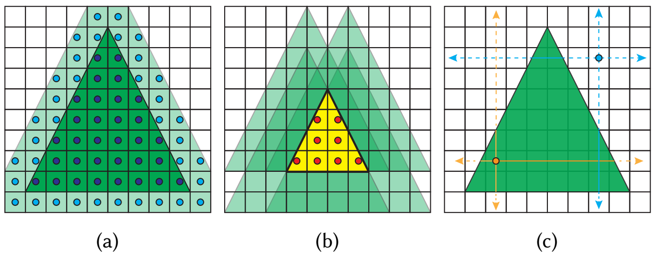


Fig. 3. (a) A green triangle and ±1 px supported camera movement (light green). (b) Traditional visibility computations consider which samples are covered under *all* camera offsets (red) via intersection (yellow) of all supported offsets. (c) Our approach reverses the question and explores under which camera offsets a sample stays within an occluder (full line) or moves outside (dashed line) .

2.2 Other Components

While we focus on visibility computations, we would like to point towards methods that could be used for other components of the system. Shading information sent to the client could be generated using object-space shading [Burns et al. 2010; Hillesland and Yang 2016; Hladky et al. 2019; Mueller et al. 2018]; view-dependent shading information could be updated from previous shading results [Sitthi-amorn et al. 2008] or integrated similarly to image-based rendering techniques [Kopf et al. 2013; Lochmann et al. 2014; Sinha et al. 2012; Zimmer et al. 2015].

Frame rate upsampling [Didyk et al. 2010] and remote rendering [Scherzer et al. 2011] are both related to our complete pipeline. In most cases, image-based rendering [Chen and Williams 1993] (IBR) is used to hide latency and create new views. These techniques include *forward warping* [Chang and Ger 2002; Chen and Williams 1993; Wolberg 1998], *mesh-based warping* [Lee et al. 2015; Mark et al. 1997], *backward warping* [Nehab et al. 2007; Yang et al. 2011], and using *approximate proxy geometry* [Buehler et al. 2001; Debevec et al. 1996; Reinert et al. 2016]. All of these approaches have issues with disocclusions. Our approach, on the other hand, does not, as the PVS is transmitted to the client. Furthermore, our client-side rendering is very efficient, as it neither involves complex geometry, nor depth buffer reconstruction, nor stepping through depth images. We simply render the PVS triangles.

3 CAMERA OFFSET SPACE

Our way of approaching visibility by asking the question "*Under which camera offsets* is a single screen location covered by an object?" is illustrated in Fig. 3. Using this information we can determine for any pair of objects whether the camera offsets under which they cover a sample overlap and thus determine visibility between them.

3.1 Offset Space for a Single Triangle

To introduce the COS, we start by considering a single triangle and camera movement within a three dimensional cuboid region. Starting from a triangle is intuitive, as our target is to construct the PVS from the rendering stream of triangles without connectivity or object information as input. While the following derivations start simple and are related to well known concepts of standard real-time rendering, we quickly depart from these notions and construct a precise description of the COS.

Let the supported camera movements be defined around the current camera location as a cuboid in camera space: $\mathbf{0} \pm \mathbf{c}, \forall \mathbf{c} \in (0, \mathbf{C}]$, where $\mathbf{C}$ is the camera offset for which the PVS is computed. Furthermore, we assume a triangle $T = (\mathbf{p}_{0_c}, \mathbf{p}_{1_c}, \mathbf{p}_{2_c})$ in clip space, with $\mathbf{p}_{i_c} = \begin{bmatrix} x_{i_c} & y_{i_c} & z_{i_c} & w_{i_c} \end{bmatrix}^T$. The projection of a triangle is described as the projection of the triangle's edges. Each edge's projection is captured by a plane that contains the edge and the origin in the (x, y, w)-subspace of the clip space. The plane for an edge $\mathbf{e}_{i_c}$ of T is defined by its normal given by the pairwise cross product of the vertices' x, y, and w coordinates:

$$\mathbf{e}_{01_c} = \hat{\mathbf{p}}_{0_c} \times \hat{\mathbf{p}}_{1_c} \qquad \mathbf{e}_{12_c} = \hat{\mathbf{p}}_{1_c} \times \hat{\mathbf{p}}_{2_c} \qquad \mathbf{e}_{20_c} = \hat{\mathbf{p}}_{2_c} \times \hat{\mathbf{p}}_{0_c}, \tag{1}$$

where the *hat* notation depicts the (x, y, w) coordinates of clip-space entities, *i.e.* $\hat{\mathbf{p}}_{i_c} = \begin{bmatrix} x_{i_c} & y_{i_c} & w_{i_c} \end{bmatrix}^T$. For a sample location $\begin{bmatrix} x_{s_d} & y_{s_d} \end{bmatrix}^T$ in normalized device coordinates, we can test whether the sample location is covered by the triangle, determining its location with respect to all three edge planes. If $\mathbf{e}_{i_c} \cdot \begin{bmatrix} x_{s_d} & y_{s_d} & 1 \end{bmatrix}^T \leq 0 \; \forall i$, the triangle covers the sample. Note that $\cdot$ indicates the dot product.

Considering a potential camera movement $\mathbf{C}' = \begin{bmatrix} C'_x & C'_y & C'_z \end{bmatrix}^T$, the camera (view) matrix $\mathbf{V}$ is altered: $\mathbf{V}' = \mathbf{V} + \mathbf{T}_C$, where $\mathbf{T}_C$ is a zero matrix with $-\mathbf{C}'$ in the last column. A point in object space $\mathbf{p}_o$ is transferred to its clip space location $\mathbf{p}_c = \begin{bmatrix} x_c & y_c & z_c & w_c \end{bmatrix}^T$ using the model ($\mathbf{M}$), view and projection matrix ($\mathbf{P}$): $\mathbf{p}_c = \mathbf{P} \cdot \mathbf{V} \cdot \mathbf{M} \cdot \mathbf{p}_o$.

For the potential camera movement and a standard perspective projection with near plane distance n, far plane distance f, and near plane size $(2r, 2t)$, the point's location in clip space is given by

$$\mathbf{p}'_c = \mathbf{P} \cdot (\mathbf{V} + \mathbf{T}_C) \cdot \mathbf{M} \cdot \mathbf{p}_o \tag{2}$$

$$= \mathbf{p}_c + \mathbf{P} \cdot \begin{bmatrix} -C'_x \\ -C'_y \\ -C'_z \\ 1 \end{bmatrix} = \mathbf{p}_c + \begin{bmatrix} n/r & 0 & 0 & 0 \\ 0 & n/t & 0 & 0 \\ 0 & 0 & zz & zw \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} -C'_x \\ -C'_y \\ -C'_z \\ 1 \end{bmatrix}.$$

Thus, $\mathbf{p}'_c$ describes $\mathbf{p}_c$ under offset $\mathbf{C}'$. Note that $\mathbf{T}_C \cdot \mathbf{M} \cdot \mathbf{p}_o$ yields -$\mathbf{C}'$ as most of the matrix elements are zero.

Considering the previous discussion, we are interested in the (x, y, w)-subspace and thus equation (2) simplifies to:

$$\hat{\mathbf{p}}'_c = \hat{\mathbf{p}}_c + \begin{bmatrix} -C'_x \frac{n}{r} & -C'_y \frac{n}{t} & C'_z \end{bmatrix}^T \tag{3}$$

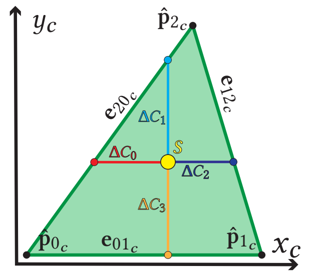
For any camera offset within $\mathbf{C}$, let

$$\Delta_x = -C'_x \frac{n}{r} \qquad \Delta_y = -C'_y \frac{n}{t} \qquad \Delta_w = C'_z \tag{4}$$

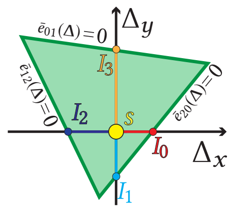
Without loss of generality, consider the influence of the camera movement on $\mathbf{e}_{12_c}$. The plane equation of edge $\mathbf{e}_{12_c}(\mathbf{\Delta})'$ going through two points $\mathbf{p}'_{1_c}$ and $\mathbf{p}'_{2_c}$ evolves as follows

$$\mathbf{e}'_{12_c}(\mathbf{\Delta}) = \begin{bmatrix} x_{1_c} + \Delta_x \\ y_{1_c} + \Delta_y \\ w_{1_c} + \Delta_w \end{bmatrix} \times \begin{bmatrix} x_{2_c} + \Delta_x \\ y_{2_c} + \Delta_y \\ w_{2_c} + \Delta_w \end{bmatrix} \tag{5}$$

$$= \begin{bmatrix} \Delta_y(w_{2_c} - w_{1_c}) + \Delta_w(y_{1_c} - y_{2_c}) + y_{1_c}w_{2_c} - y_{2_c}w_{1_c} \\ \Delta_x(w_{1_c} - w_{2_c}) + \Delta_w(x_{2_c} - x_{1_c}) + w_{1_c}x_{2_c} - x_{1_c}w_{2_c} \\ \Delta_x(y_{2_c} - y_{1_c}) + \Delta_y(x_{1_c} - x_{2_c}) + x_{1_c}y_{2_c} - y_{1_c}x_{2_c} \end{bmatrix}$$



(a) (x, y) clip space



(b) 2D COS

Fig. 4. (a) A triangle in clip space covering a sample s (yellow) and 4 different camera offsets (ΔC_i) where the sample s exactly coincides with a triangle edge. (b) The same triangle in a 2D slice of the COS for sample s. The points I_i denote the intersections of the triangle's COS representation with the COS axes, which correspond to the camera offsets indicated in (a).

Thus, the scaled distance $\bar{e}_{12}$ of a given sample $\mathbf{s} = \begin{bmatrix} s_{x_d} & s_{y_d} & 1 \end{bmatrix}^T$ (relative to the moving camera) to the edge $\mathbf{e}'_{12_c}$ is given by

$$\begin{aligned} \bar{e}_{12}(\mathbf{\Delta}) &= \mathbf{e}'_{12_c}(\mathbf{\Delta}) \cdot \mathbf{s} \\ &= \Delta_x \cdot \bar{e}_{12_c,x} + \Delta_y \cdot \bar{e}_{12_c,y} + \Delta_w \cdot \bar{e}_{12_c,w} + \bar{e}_{12_c,0} \\ &= \mathbf{\Delta} \cdot \begin{bmatrix} \bar{e}_{12_c,x} & \bar{e}_{12_c,y} & \bar{e}_{12_c,w} \end{bmatrix}^T + \bar{e}_{12_c,0}, \end{aligned} \tag{6}$$

with

$$\begin{aligned} \bar{e}_{12_c,x} &= s_{y_d}(w_{1_c} - w_{2_c}) + 1 \cdot (y_{2_c} - y_{1_c}), \\ \bar{e}_{12_c,y} &= s_{x_d}(w_{2_c} - w_{1_c}) + 1 \cdot (x_{1_c} - x_{2_c}), \\ \bar{e}_{12_c,w} &= s_{x_d}(y_{1_c} - y_{2_c}) + s_{y_d}(x_{2_c} - x_{1_c}), \\ \bar{e}_{12_c,0} &= s_{x_d}(y_{1_c}w_{2_c} - y_{2_c}w_{1_c}) + s_{y_d}(w_{1_c}x_{2_c} - x_{1_c}w_{2_c}) \\ &\quad + 1 \cdot (x_{1_c}y_{2_c} - y_{1_c}x_{2_c}). \end{aligned} \tag{7}$$

For $\mathbf{s}$ being a sample on screen, we are interested in $\bar{e}_{12}(\mathbf{\Delta}) = 0$, *i.e.*, offsets $\mathbf{\Delta}$ for which the edge $\mathbf{e}'_{12_c}$ coincides with $\mathbf{s}$. In other words, those camera offsets for which the fragment moves outside the triangle edge. It can be observed that $\bar{e}_{12}(\mathbf{\Delta}) = 0$ corresponds to a plane in a three dimensional space over $\mathbf{\Delta}$. We call this space the *camera offset space.* For an example of a slice of the COS see Fig. 4.

3.2 Offset Space for a Point

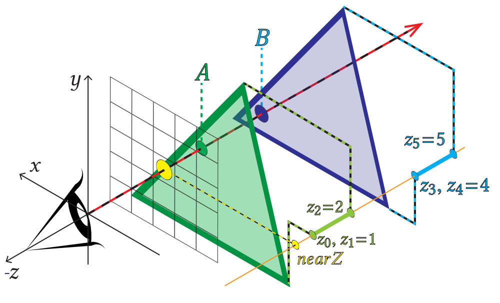
Similarly to transferring a triangle into camera offset space, we can consider points in COS, *i.e.*, compute under which offsets a vertex' projection exactly coincides with the sample location. To compute the normalized device coordinate (NDC) representation of a point in clip space, one simply needs to perform the perspective division:

$$\mathbf{p}_{ndc} = \begin{bmatrix} x_{p_c}/w_{p_c} & y_{p_c}/w_{p_c} \end{bmatrix}^T$$

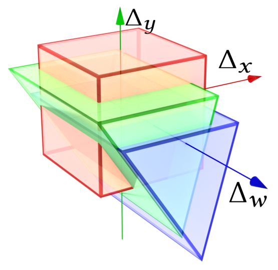
This in combination with equation 3 and $\mathbf{\Delta}$ yields:

$$\begin{aligned} \frac{x_{p_c} + \Delta_x}{w_{p_c} + \Delta_w} &= s_{x_d} & \qquad \frac{y_{p_c} + \Delta_y}{w_{p_c} + \Delta_w} &= s_{y_d} \\ \Delta_x - \Delta_w s_{x_d} &= w_{p_c}s_{x_d} - x_{p_c} & \qquad \Delta_y - \Delta_w s_{y_d} &= w_{p_c}s_{y_d} - y_{p_c} \end{aligned} \tag{8}$$

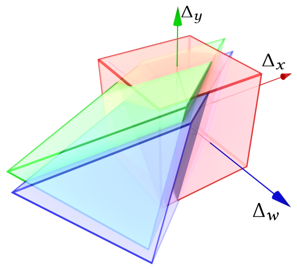
The two equations describe a line. Note that the change of Δ_x while moving along Δ_w is proportional to s_{x_d}. Similarly, Δ_y changes with s_{y_d}. Thus, the direction of the line only depends on the sample



(a) isometric projection of view space

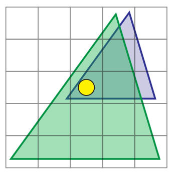


(b) 3D COS; $\mathbf{s} = [0, 0]^T$

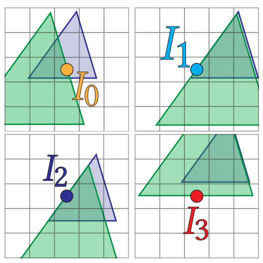


(c) 3D COS; $\mathbf{s} = [1, 1]^T$

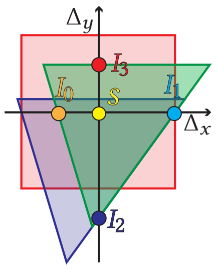
Fig. 5. (a) Two identical but translated triangles with tips leaning away from the camera (z_2 and z_5). The view ray for the sample intersects the triangles at points A and B. (b) 3D COS for sample location at the origin of normalized device coordinates, for which the triangles correspond to triangular prisms orthogonal to the Δ_w axis. (c) for other samples the prisms are more oblique the further from the origin they are.



(a) img space



(b) img space under C



(c) 2D COS at $\Delta_w = 0$

Fig. 6. Visibility consideration for the triangles from Fig. 5. (a) Their image space projection shows the typical perspective effects. (b) Offsetting the camera to positions where the sample coincides with the edges of the green triangle: both triangles move differently due to their depth relations, revealing positions that were not clear from (a). (c) The 2D COS reveals those constellations. Furthermore, the red square denotes the supported camera offset range and immediately reveals under which camera offsets the blue triangle will be visible at sample $\mathbf{s}$.

location, meaning all points correspond to parallel lines in COS for a certain sample location.

3.3 Camera Offset Space Considerations

Analyzing the derived equations, we draw the following conclusions: Considering all three edges of a triangle T, it becomes apparent that the combination of $\bar{e}_{01}(\mathbf{\Delta}) \leq 0$, $\bar{e}_{12}(\mathbf{\Delta}) \leq 0$, $\bar{e}_{20}(\mathbf{\Delta}) \leq 0$ forms a subspace limited by three intersecting planes in the COS. Given that all points result in parallel lines, the subspace is an oblique triangular prism with infinite extent, which we denote as T_Δ. The obliqueness of the prism depends on $\mathbf{s}$, as shown in Fig. 5b and 5c. To have T_Δ describe for which camera offsets the fragment $\mathbf{s}$ will be covered by T, we additionally have to consider under which $\mathbf{C}$ the near plane passes through the triangle, truncating the front of the prism.

If we only allow two dimensional camera offsets, $\Delta_w = 0$, *i.e.*, visibility from a plane, the camera offset space becomes 2D, $\bar{e}_{12}(\mathbf{\Delta}) = 0$ corresponds to a line, and T_Δ becomes a triangle. While the COS provides information under which offsets a sample is covered by a triangle or exactly meets the projection of a vertex, it does not consider the maximum supported camera movement. The camera movement is simply a cuboid/rectangle in 3D/2D COS, defined by the maximum offset $\pm\mathbf{C}$. Thus, visibility considerations in COS can be limited to $\pm\mathbf{C}$.

Analyzing the derived equations, we draw the following conclusions: First, triangles of identical shape, that are only translated relative to each other, but not rotated, cover the exact same area in COS for a single $\mathbf{s}$ (see Fig. 6). Furthermore, this means that unlike perspective projection, where distant objects become smaller, objects in COS keep their size. Thus, visibility considerations in COS are more intuitive than in image space. Second, considering the right side of equation 8, it becomes apparent that the 2D COS can be thought of as an oblique projection, where the view plane is the projection surface and the view-ray of $\mathbf{s}$ defines the projectors. Thus, the projection changes as $\mathbf{s}$ is changed. For $\mathbf{s} = [0, 0]^T$ the projection is orthographic. Note that although the 3D COS results in oblique prism shapes, this effect comes from the $-\Delta_w s_{\cdot d}$ part of equation 8. Third, when stepping along Δ_w the relations on the x-y plane simply shift. Thus, all 3D considerations could also be made in 2D by considering the projection of the $\pm\mathbf{C}$ cuboid to the representative 2D COS plane.

3.4 Visibility for Pairs of Triangles

The camera offset space allows computing visibility between triangles for a single sample location. To evaluate the potential visibility of two triangles T and Q at fragment $\mathbf{s}$ under possible camera offsets $\mathbf{\Delta}$, we compute T_Δ and Q_Δ, *i.e.*, bring them into COS. Intersecting T_Δ with Q_Δ within $\pm\mathbf{C}$, whose volume/area in offset space shall be denoted as $\mathbf{C}_\Delta$, we can distinguish three cases:

(1) $T_\Delta \cap Q_\Delta \cap \mathbf{C}_\Delta = T_\Delta \cap \mathbf{C}_\Delta$
(2) $T_\Delta \cap Q_\Delta \cap \mathbf{C}_\Delta = Q_\Delta \cap \mathbf{C}_\Delta$
(3) neither of the above.

For (1), the part of T_Δ inside $\pm\mathbf{C}$ overlaps with Q_Δ, *i.e.*, every camera offset that samples T also hits Q. Thus, if Q is closer to the camera for all those samples, T *will never be visible* at sample $\mathbf{s}$. In case of (2), the relation of the triangles is reversed and Q will never be visible if T is closer. In case (3), both triangles may generate visible samples at $\mathbf{s}$ under different camera offsets. Clearly, this already shows that visibility in the COS does not need to explicitly distinguish between occluder and occludee up front, as their COS representation brings forth their relative positioning.

So far, we have not considered depth relationships in COS. Determining whether a triangle is in front of another requires the exact depth value a triangle will produce for a *fixed* sample on the image plane under all camera offsets. To compute this depth value we consider the formulas describing homogeneous rasterization [Olano and Greer 1997]: For a triangle at sample location $\mathbf{s}$, the interpolated $1/w_c$ component (which can be interpreted as the inverse of the depth) is given by

$$\frac{1}{w_s} = \begin{bmatrix}1 & 1 & 1\end{bmatrix} \cdot \begin{bmatrix} x_{0_c} & x_{1_c} & x_{2_c} \\ y_{0_c} & y_{1_c} & y_{2_c} \\ w_{0_c} & w_{1_c} & w_{2_c} \end{bmatrix}^{-1} \cdot \begin{bmatrix} s_{x_d} \\ s_{y_d} \\ 1 \end{bmatrix}.$$

Considering camera offsets (adding $\mathbf{\Delta}$) and solving for w_s via the adjugate matrix and determinant to compute the inverse, many terms cancel out and a simple equation results:

$$w_s = \frac{\Delta_x d_x + \Delta_y d_y + \Delta_w d_w + d_0}{d_{adj}} = \frac{\mathbf{\Delta} \cdot \mathbf{d} + d_0}{d_{adj}}. \qquad (9)$$

d_x, d_y, d_w, and d_{adj} are all functions of $(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{s})$ and thus constant for a given triangle and a certain sample location. The structure shows that by adding the depth information and thus extending the 3D COS to 4D (or the 2D COS to 3D), the triangle/pyramid gets embedded on a flat hyperplane in the respective space.

This yields three properties which allow elegant depth considerations in an extended camera offset space: First, for every point in 2D and 3D COS, a depth value can efficiently be computed using equation 9. Second, visibility computations in 2D COS can be seen as an orthographic projection of triangles from a 3D space where the third dimension is given by equation 9. Determining the visibility within $\pm\mathbf{C}$ in the orthographically projected 2D COS yields the visibility of the involved triangles at the given sample location under all camera offsets. Although more complicated to imagine, the same considerations hold for 3D COS, where depth forms a fourth dimension in which an orthographic projection down to 3D yields the visibility.

4 POTENTIALLY VISIBLE SET AND RASTERIZATION

By applying the considerations derived for a single sample to all pixels of a rendered image, we can compute a PVS for the entire scene considering translations around the current camera location. However, there are few problems with such approach. First, computing the intersections of 3D subspaces is not very efficient. Second, considerations on pairs of triangles in practice are not sufficient for a good PVS, as closed meshes will often occlude large parts of the scene, while their individual triangles are too small. Third, the considerations only hold for translational movement and do not consider rotation. In the following, we will address these points to derive a practical PVS algorithm that works in conjunction with rasterization.

4.1 3D Movement and Camera Rotation

Increasing the field of view (FOV) is a starting point for dealing with rotational camera movements. However, in practice, the FOV is limited to stay below 180°, strongly limiting potential rotations. Similar to previous work [Durand et al. 2000], the PVS for a 3D region can be computed from multiple 2D regions, especially when considering camera rotations: For a cuboid cell, the visibility can be considered separately for all six bounding planes by placing the camera in the center of each plane and matching the cuboid's dimension with $\mathbf{\Delta}$. Thus the evaluated camera positions correspond to all side plane positions. With a 180° FOV, all view-rays exiting the side planes are captured and the complete PVS for the 3D region is constructed using 2D COS. While we could rely on the properties of the COS itself to bring computations from 3D to 2D, using the side planes has the same result and we avoid the projection of the COS cuboid.

4.2 Fragment Lists and Sorting

For a practical implementation of the PVS using COS during rasterization, we construct the COS for each pixel and resolve visibility for each pixel. This corresponds to an analytic visibility for all camera offsets sampled at all pixel locations. We will tackle the sampling issue after presenting a visibility resolution algorithm for each pixel. At each pixel, all triangles T_i that fulfill $T_{\Delta_i} \cap \mathbf{C}_\Delta \neq \emptyset$ are needed, *i.e.*, all triangles which may cover the pixel under *any* supported camera offset. To gather those triangles we construct a per-fragment linked-list [Yang et al. 2010]. To generate samples for all triangles that may cover a pixel, we expand them in size before rasterization. As the expansion depends on $\mathbf{C}$ and each triangle's location, a dynamic size increase is needed, which we achieve using an approach similar to workarounds for conservative rasterization presented by [Hasselgren et al. 2005]. Note that we do not need to consider rotations here, as all view-ray directions are covered by moving the camera on the side planes of the view cell.

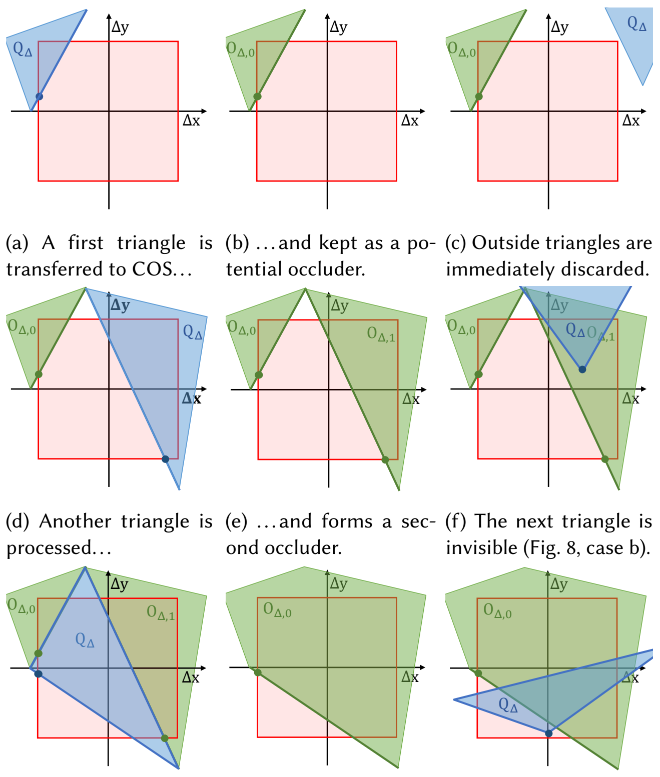
By storing information about the fragment generating triangle in each list entry, we can construct the COS for each pixel. As the visibility considerations unfold around depth relations, it is beneficial to sort those lists. However, as the depth in COS is a function (see equation 9), no single depth value can be associated with each entry and the COS depth ranges might even overlap. Thus, sorting can only be approximative such that the list is processed in a quasi front-to-back manner. We sort the lists according to the minimum depth, which places potential occluders early in each list.

4.3 Occluder Fusion

As mentioned before, resolving visibility between triangle pairs is hardly sufficient, as only the combination of multiple triangles may form a sufficiently large occluder. Thus, we employ occluder fusion during visibility resolution in the COS. Note that occluder fusion has been used for entire objects during PVS construction [Schaufler et al. 2000]. We require an efficient version for individual triangles that works in the COS, *i.e.*, merge triangles into a polygon.

The benefits of occluder fusion are twofold. First, it can be more efficient to consider the overlap of a triangle with a polygon than with many triangles (*cf.* the three cases outlined in Section 3.4). Second, a fused occluder may occlude a triangle as a whole, whereas its individual parts may not. With occluder fusion the process mathematically corresponds to

$$T_{\Delta_{fused}} = T_{\Delta_0} \cup T_{\Delta_1} \cup T_{\Delta_2}$$
$$O = Q_\Delta \cap T_{\Delta_{fused}} \cap \mathbf{C}_\Delta,$$



(a) A first triangle is transferred to COS…

(b) …and kept as a potential occluder.

(c) Outside triangles are immediately discarded.

(d) Another triangle is processed…

(e) …and forms a second occluder.

(f) The next triangle is invisible (Fig. 8, case b).

(g) A triangle sharing two edges…

(h) …fuses the occluders into one.

(i) Triangles intersecting the silhouette are kept.

Fig. 7. Example steps of progressive visibility resolution. Active edges and reference points are indicated. The current triangle is given in blue, occluders in green. Note that the active occluder silhouette always corresponds to the active edges and intersections of active edges outside C_Δ are ignored.

where T_{Δ_i} forms the fused occluder and O yields information about the visibility. To get the same occlusion information without occluder fusion, the occludee's representation in the COS would have to be individually clipped against all occluder triangles instead. Keeping track of which parts of an occludee have not been clipped yet is prohibitively complex. Thus, we employ occluder fusion by merging triangles to form larger occluders if they share an edge in the COS. Note that two triangles share an edge in the COS if they also share an edge in world space. Thus, connected surfaces are the only objects that form larger occluders.

4.4 Progressive Visibility Resolution

To bring all previous considerations together, we describe a serial visibility resolution algorithm that can run for every per-pixel list in parallel. The algorithm is outlined in Algorithm 1 with example steps shown in Fig. 7. The algorithm runs through the list of approximately sorted triangles (ln 2) and determines the visibility for one triangle after the other. Previously checked triangles are kept as potential occluders and merged if they share edges.

For every triangle, we start with an analysis phase (ln 3-14): First, we compute its COS representation (ln 3) and check its overlap

Algorithm 1: Progressive Visibility Resolution

```
1  $Occluders \leftarrow \emptyset$
2  for $Q \in TriangleList$ do
3      $Q_\Delta \leftarrow$ computeCOS $(Q, \mathrm{s})$
4      for $i \leftarrow 0$ to $3$ do
5          if intersect($Q_\Delta.edge(i)$, rectangle($C_\Delta$)) then
6              $Q_\Delta.edge(i) \leftarrow$ CROSSING
7          else if outside($Q_\Delta.point(i), C_\Delta$) then
8              $Q_\Delta.edge(i) \leftarrow$ INACTIVE
9          else
10             $Q_\Delta.edge(i) \leftarrow$ ACTIVE
11     if $Q_\Delta.edge(0\dots2) =$ INACTIVE and $Q_\Delta \cap \mathbf{0} = \emptyset$ then
12         continue
13     $Q_\Delta.\mathrm{d} \leftarrow$ depthEquation($Q_\Delta$, s)
14     $Q_\Delta.refs(0\dots2) \leftarrow$ choosePointsIn($Q_\Delta \cap C_\Delta$)
15     $Qvisible \leftarrow$ true
16     for $Occluder \in Occluders$ and $Qvisible$ do
17         if $\neg Occluder.coverall$ then
18             $Overlap \leftarrow \emptyset$
19             for $T_\Delta \in Occluder$ and $Overlap = \emptyset$ do
20                 for $(e_{Q_\Delta}, e_{T_\Delta}) \in Q_\Delta.edge \times T_\Delta.edge$ do
21                     if $e_{Q_\Delta}$ or $e_{T_\Delta} =$ INACTIVE then
22                         continue
23                     if $e_{Q_\Delta} = e_{T_\Delta}$ then
24                         $Merging \leftarrow Merging \cup (e_{Q_\Delta}, e_{T_\Delta})$
25                         continue
26                     i $\leftarrow$ intersect($e_{Q_\Delta}, e_{T_\Delta}$)
27                     if i $\neq \emptyset$ and i $\cap C_\Delta \neq \emptyset$ then
28                         $Overlap \leftarrow (e_{Q_\Delta}, e_{T_\Delta})$
29                         break 2
30             if ($Overlap \neq \emptyset$) then continue
31             if ($\exists r \in Occluder.refs \mid r \in Q_\Delta$) then continue
32         for $T_\Delta \in Occluder$ do
33             if $\exists r \in Q_\Delta.refs \mid r \in T_\Delta$ then
34                 if depth($T_\Delta$.d, $r$) < depth($Q_\Delta$.d, $r$) then
35                     $Qvisible \leftarrow$ false
36                 break
37     if $Qvisible$ then
38         if $Merging \neq \emptyset$ then
39             fuseOccluder($Merging$)
40             setEdgesInactive($Merging$)
41         else
42             $Occluders \leftarrow Occluders \cup Q_\Delta$
43         $actives \leftarrow (e \mid e \in Occluder.edge \wedge e \neq$ INACTIVE$)$
44         if $actives = \emptyset$ then
45             $Occluder.coverall \leftarrow$ true
46         else
47             $Occluder.refs \leftarrow$ choosePointsIn($actives \cap C_\Delta$)
48 return triangles($Occluders$)
```

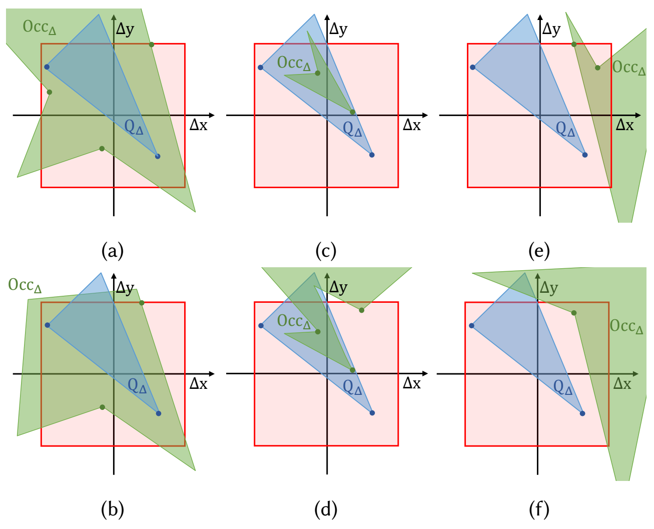


Fig. 8. Six possible cases that do not have silhouette intersections within $\mathbf{C}_\Delta$. Note how (a,b), (c,d), and (e,f) can be distinguished based on their respective reference point locations.

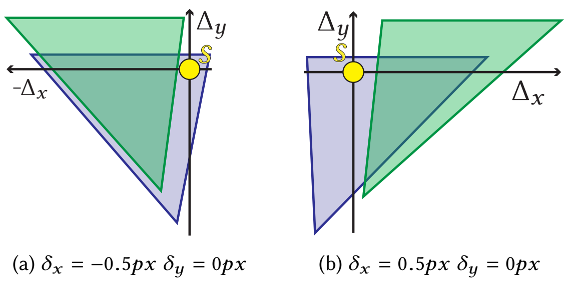

Δ_y
$-\Delta_x$
Δ_x
(a) $\delta_x = -0.5px$ $\delta_y = 0px$
(b) $\delta_x = 0.5px$ $\delta_y = 0px$

Fig. 9. Extending the case from Fig. 5 for sub-sample offsets with the original sample location in yellow. Due to view-space depth differences between the vertices, each vertex moves at a different speed when moving through the sub-sample dimensions. The edges slide and rotate, thus distorting the shape of the triangles.

with $\mathbf{C}_\Delta$ by checking all three edges (ln 4). Determining all edge intersections, we obtain additional information: Edges are classified as either *inside*, *inactive*, or *crossing*. An inside edge is completely contained in $\mathbf{C}_\Delta$; a crossing edge reaches from $\mathbf{C}_\Delta$ outside. Edges which are completely outside of $\mathbf{C}_\Delta$ are marked as inactive. If all edges are inactive (ln 12), the triangle is either completely outside $\mathbf{C}_\Delta$ and invisible, or completely covers $\mathbf{C}_\Delta$. To check the latter, we evaluate the triangle's edge equations at $\mathbf{0}$ to determine whether it covers the origin of the COS. If a triangle is outside $\mathbf{C}_\Delta$, we discard it, as it is not visible under any supported camera offset.

For triangles which have not been discarded, we set up meta information (ln 13-14): We compute the depth equation coefficients and identify up to three points that are inside both Q_Δ and $\mathbf{C}_\Delta$, which we call the triangle's reference points. If the triangle covers $\mathbf{C}_\Delta$ entirely, we choose one reference point at $\mathbf{0}$. Otherwise, for each edge labeled as *inside* or *crossing* we choose one reference point. If either of the edge's endpoints ($\mathbf{q}_0$, $\mathbf{q}_1$, or $\mathbf{q}_2$) lie inside of $\mathbf{C}_\Delta$ we use the endpoint. If both endpoints are outside $\mathbf{C}_\Delta$ (in case of a crossing edge), we compute the intersection with $\mathbf{C}_\Delta$. While this information seems arbitrary at this point, we use it in combination with the edge classification to determine the relationship of triangles efficiently throughout the algorithm. We will limit considerations to *inside* and *crossing* edges and use the reference points when we do not find any edge overlaps to determine whether a triangle is inside another or if they are separate—similarly to how we used $\mathbf{0}$ to determine if a triangle covers $\mathbf{C}_\Delta$ completely.

The main visibility algorithm checks each triangle against the current list of occluders (ln 16-36). We discard a triangle if an occluder covers it completely. In the simplest case, an occluders covers $\mathbf{C}_\Delta$ entirely and we know that there is complete overlap of the triangle and the occluder in $\mathbf{C}_\Delta$. We then skip large parts of the algorithm (ln 17-30) and directly move to considering depth relations.

If the occluder does not cover $\mathbf{C}_\Delta$, we need to determine whether $Occluder_\Delta \cap Q_\Delta \cap \mathbf{C}_\Delta = Q_\Delta \cap \mathbf{C}_\Delta$, *i.e.*, whether Q_Δ is completely inside the fused occluder within $\mathbf{C}_\Delta$. To this end, we determine if there is an intersection between the triangle and the silhouette of the occluder within $\mathbf{C}_\Delta$. If the silhouette intersects any triangle edge within $\mathbf{C}_\Delta$, the occluder certainly does not cover the triangle under any supported offset and we continue with the next occluder. Instead of keeping the silhouette around explicitly, we iterate over all occulder triangles T_Δ (ln 19) and test all occluder edges against all edges of Q_Δ. If either edge is inactive, we skip the test (ln 21). This does not only include edges outside of $\mathbf{C}_\Delta$, but also edges inside the occluder, which we set inactive when fusing triangles (ln 40)).

To compute the relation between two edges, we first test whether they are identical (ln 24). Otherwise, we compute the intersection of the edges (ln 26) and whether it is inside $\mathbf{C}_\Delta$. Note that we do not need the exact intersection $\mathbf{i}$ if either of the edges is marked as inside, as the intersection must also be inside then. Only for *crossing* edges, we compute the exact intersection. In case the intersection is outside, we ignore it, as this does not influence $Occluder_\Delta \cap Q_\Delta \cap \mathbf{C}_\Delta$. If it is inside, we move to the next occluder (ln 28).

If there is no overlap between the occluder's silhouette and the triangle, there are six possible cases to consider , as shown in Fig. 8:

(a) Q_Δ is completely inside $Occluder_\Delta$
(b) Q_Δ is inside $Occluder_\Delta$ only within $\mathbf{C}_\Delta$
(c) $Occluder_\Delta$ is completely inside Q_Δ
(d) $Occluder_\Delta$ is inside Q_Δ only within $\mathbf{C}_\Delta$
(e) $Occluder_\Delta$ and Q_Δ are completely disjoint
(f) $Occluder_\Delta$ and Q_Δ are disjoint only within $\mathbf{C}_\Delta$

To distinguish between the cases, we use Q_Δ's reference points and the reference points we keep for each occluder (*Occluder.refs* in the algorithm), which we choose from the triangle reference points as we add them to occluders. We start by testing for cases (c) and (d) as those are the easiest. The distinguishing fact for these cases is that the silhouette is (partially) inside Q_Δ. To this end, we iterate over *Occluder.refs* and if any of those is inside Q_Δ (which we test using the edge equations), we have confirmed case (c) or (d) (ln 31). Then the triangle is not occluded and we continue with the next occluder.

To distinguish between (a), (b), which may result in Q being occluded and (e), (f), which both show that Q is not occluded by this occluder, we test whether a triangle's reference point is inside $Occluder_\Delta \cap \mathbf{C}_\Delta$. As the occluder may be concave, the test is more involved and we compute it as the final check (ln 32). We iterate over all occluder triangles and check whether a reference point is inside any of them. In case it is, we have verified that $Q_\Delta \cap \mathbf{C}_\Delta$ is inside $Occluder_\Delta$ (as we already have ruled out the other cases). However, for Q to be occluded, Q must be behind the occluder, which we verify at the reference point using the occluder triangle's depth equation (ln 34). Note that we perform the same test if an occluder covers $\mathbf{C}$ to establish the depth relations. Also note that this depth test is only sufficient for non-penetrating triangles. To support penetrating triangles, we would have to iterate over all occluder triangles and check for depth intersections for all occluder triangles. This is certainly possible, but more costly.

If all tests yield that Q is visible, we add Q_Δ to the occluders (ln 37-47). If the triangle shares one or multiple edges with other occluders, we attach it to those occluders, possibly merging up to three occluders. We set the shared edges to inactive and check whether all occluder edges are inactive, which indicates that the occluder covers $\mathbf{C}_\Delta$. Then, we set the $coverall$ flag to speed up testing against that occluder. Finally, we update the occluder's reference points (ln 47). After all triangles have been processed, all potentially visible triangles have been added as occluders and they can directly be output as the PVS for sample $\mathbf{s}$.

4.5 Between Sample Locations

So far we have only considered individual sample locations. When rendering with a wider FOV, sample locations considered by the COS will not coincide with the final image sample locations. Thus, tiny triangles may be missing from the PVS. Also, when considering rotations, sample locations will not coincide with the sample locations considered by our algorithm. To be able to handle all possible sample locations within a pixel, we extend the COS by another two dimensions which represent the sample offsets δ_x and δ_y in x and y direction, respectively. δ_x and δ_y form linear offsets to the sample location $\mathbf{s}$. Thus, the PVS computation becomes resolution independent as a result and can be run at reduced resolution. A maximum offset of $\pm pixelsize/2$ makes sure the algorithm considers all possible samples on the view plane. For a 2D COS, Equation 8 extends to:

$$\begin{aligned} \frac{x_p + \Delta_x}{w_p} &= s_x + \delta_x & \frac{y_p + \Delta_y}{w_p} &= s_y + \delta_y \\ \Delta_x - w_p\delta_x &= w_p s_x - x_p & \Delta_y - w_p\delta_y &= w_p s_y - y_p, \end{aligned} \tag{10}$$

which corresponds to a line in 4D space. Adding these two dimensions essentially includes the change of the oblique projection when moving from one sample location to the next. Considering individual edges, they slide and rotate when moving along the δ_x and δ_y dimension, as shown in Fig. 9.

While this extension makes the COS more complex, the changes to our approach are manageable. For the list construction, we increase the triangle enlargement by half a pixel. For algorithm 1, the edge intersection tests are affected. Instead of determining if a triangle edge intersects with $\mathbf{C}_\Delta$ (ln 5), we determine if it intersects $\mathbf{C}_\Delta$ under any δ. The same is true for edge-edge intersections (ln 27).

All other computations remain the same under the following assumptions: First, depth relations of a triangle and an occluder remain the same under any δ. Second, inside-outside tests for $\delta = 0$ carry over to all sample offsets. Both assumptions hold, if the silhouette-triangle intersection algorithm reports intersection under any sample offset. If there is no intersection, the occluder reference point check (ln 31) and the triangle reference point check (ln 32) can be run with $\delta = 0$ to draw conclusions for all δ.

The issue of edge-$\mathbf{C}_\Delta$ intersection and edge-edge intersection under a sample offset are the same problem; the edge-$\mathbf{C}_\Delta$ case does not apply any offset to the boundary edges of $\mathbf{C}_\Delta$. As we are only interested in whether there is an intersection and not in its exact location (unless both edges are flagged as crossing), a simple test for line segment intersection detection can be used. Ignoring sample offsets, one such test is formed by computing the orientations o of all four groups of point triples [Cormen 2009, Chapter 3.1]. Let

$$o(\mathbf{a}, \mathbf{b}, \mathbf{c}) = sgn((y_c - y_a) \cdot (x_c - x_b) - (x_b - x_a) \cdot (y_c - y_b))$$

be the orientation of points $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, with $[x_i \; y_i]^T$ being the coordinates of point $\mathbf{i}$. If $o(\mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_0) = -o(\mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_1)$ and $o(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2) = -o(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_3)$, the line segments (p_0, p_1) and (p_2, p_3) intersect.

These computations can easily be extended to the sample offset dimensions, by inserting equation 10 for all for edge points. Setting the $o(\cdot, \cdot, \cdot) = 0$ for each group yields a line in δ. These lines describe when the orientation of one group changes while stepping through the sample offset dimension. The combination of all lines partitions the δ-dimension into regions with different orientations. Considering the orientations of each partition inside $\pm pixelsize/2$, we compute whether the line segments intersect under any δ. Note that most often, the lines fall outside $\pm pixelsize/2$ and thus the overhead of adding the sample offset dimension does not significantly affect performance.

Determining the exact intersection of two edges marked as *crossing* is not as straightforward. Expressing the intersection by inserting the parameter-free line equation of one edge into the parameterized equation of the other yields a hyperbola in δ. Instead of trying to solve the equation, we opt for a conservative estimate of the intersection location, which is more efficient to compute. We estimate the parameter range under which an intersection can occur: We first insert $\pm pixelsize/2$ into the hyperbolic function to establish the intersections at the extrema of the supported sample offsets. Then, we compute its derivative and set it to zero to get the potential extrema locations inside $\pm pixelsize/2$ and evaluate those. Additionally, we bound the established parameter range by the edge itself $(0, 1)$. This yields a bound under which an intersection can occur. Evaluating the line equation for the extrema of this parameter range and extending the resulting $\mathbf{\Delta}$ range by the maximum endpoint displacement under δ gives a conservative estimate of where the intersection of the edges in the COS can occur. If this conservative estimate does not intersect $\mathbf{C}_\Delta$, the edge intersection is certainly outside $\mathbf{C}_\Delta$. A conservative estimate results in a potential increase of the PVS, but never removes visible triangles.

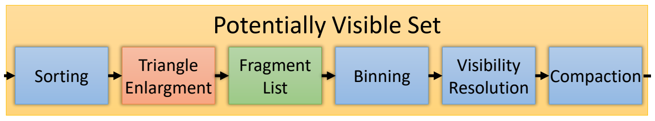


Fig. 10. Our PVS construction in detail: blue blocks correspond to CUDA kernels, red indicates a geometry shader and green a fragment shader.

5 IMPLEMENTATION

To implement the COS PVS algorithm on the GPU, we use a combination of OpenGL and CUDA. The same functionality can also be achieved using any other graphics API and compute mode shader execution. To evaluate our technique in practice, we implement a prototype of the entire pipeline. Starting from simple geometry input, we compute the PVS which yields a list of triangles, which we shade in object space, pack into a texture and transmit to the client. On the client side, we perform simple textured rendering of the transmitted triangles according to the current view. The implemented PVS steps are outlined in Fig. 10.

Our PVS algorithm starts with capturing all triangles rendered for a given view. Next, we create an approximately sorted per-pixel list of triangles that may cover a pixel under any camera offset. To this end, we approximately sort all triangles before creating linked lists, to avoid sorting each list individually. For the linked list creation, we use an OpenGL rendering pass. To determine all pixels that can potentially be covered by a triangle, we enlarge every triangle similarly to conservative rasterization in the geometry shader [Durand et al. 2000]. To increase performance, we perform list creation in a two-pass approach. During sorting (when all triangles are touched for the first time), we determine the number of triangles that will hit a fragment and thus can use fixed size memory for each list. Furthermore, we store an *early z* discard value for each fragment when we determine that a triangle covers a fragment under all camera offsets. These optimizations significantly reduce computations in comparison to capturing a dynamic linked list first and sorting all lists separately.

Working on different list sizes involves vastly different numbers of steps, thus we use specialized implementations of algorithm 1 for different bin sizes. Small lists use a single thread and parallelization happens across threads. Medium-sized lists are handled by small groups of threads (warps), where the warps at first construct occluders in a combined effort and then re-test visibility of all triangles against the preformed occluders. Larger lists start with a cooperative initialization similarly to the warp variant. Then, every thread runs algorithm 1 on one incoming triangle in parallel.

6 RESULTS

To evaluate our approach, we test four 3D scenes with various characteristics (Fig. 11). We use pre-recorded camera paths, which correspond to natural interactions with those scenes. Each scene is tested in two versions: the complete geometry and a low LOD version of the scene (Fig. 12). To yield correct results, the LOD version requires a mapping from simplified to full geometry. Furthermore, the LOD version needs to feature an inscribed and a circumscribed geometry version—one for occludee testing and the other for testing

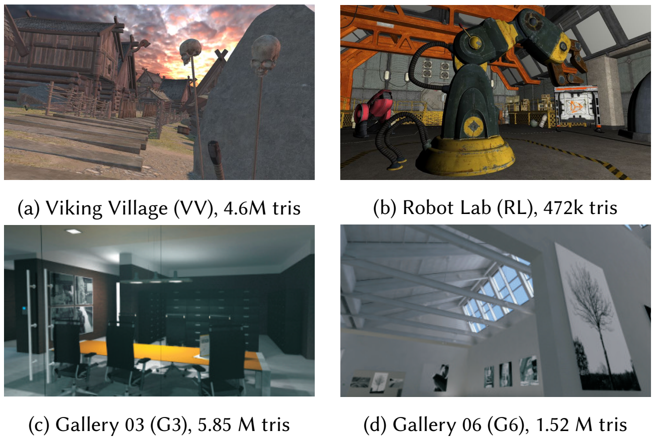

(a) Viking Village (VV), 4.6M tris
(b) Robot Lab (RL), 472k tris
(c) Gallery 03 (G3), 5.85 M tris
(d) Gallery 06 (G6), 1.52 M tris

Fig. 11. Tested scenes. (a) is a large outdoor scene offering many challenges for our algorithm, such as long fragment lists, small slanted triangles, and multiple layers of slightly offset structures. (b) is a smaller indoor scene. (c, d) are massive CAD indoor scenes with high-detail geometry from DoshDesign.

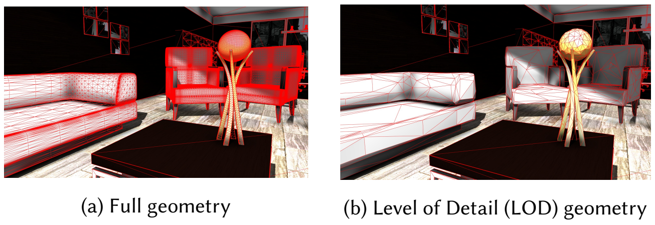


Fig. 12. LoD example used for Gallery 03 scene.

whether a triangle should become an occluder. Tests were run on an Intel Xeon CPU E5-2643 @ 3.4 GHz with 32 GB of RAM and an NVIDIA Titan Xp.

For our tests, we consider a head movement of 5 to 30 cm, a rotation of ±15 to 60 degree, and COS-pass resolutions between 200 × 100 and 1000 × 500. All client renderings are performed in 1920 × 1080 with 50 degree FOV. Note that all COS-pass resolutions yield the full PVS, as we always consider between pixel locations. This parameter set together with eight scene variants yields 960 different configurations, all included in the supplemental material. In the following, we present representative results from the test set.

There is no other PVS algorithm designed to work for our usecase: a dynamic input triangle stream without preprocessing. Nevertheless, we have implemented an efficient version of Instant visibility [Wonka et al. 2001] to work on individual triangles: At first the scene is rendered with occluder shrinking being applied on all triangles. During a second pass, the depth buffer of the first pass is used to determine triangles that may become visible at any location. As resolution, we always use the client resolution (1920 × 1080). To compare to the state-of-the-art for streaming rendering, a comparison to image-based rendering approaches can be found in the supplemental material. These results indicate the advantages of having a PVS available and sending shading information for the entire potentially visible geometry to a client.

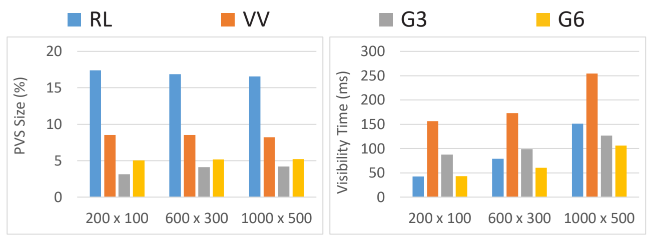


Fig. 13. Average PVS size for walkthroughs of full detailed scenes (in percent of full geometry) and timing (in ms) for various fragment list construction resolutions. The PVS always contains all necessary triangles. Increased sizes correspond to conservative overestimations.

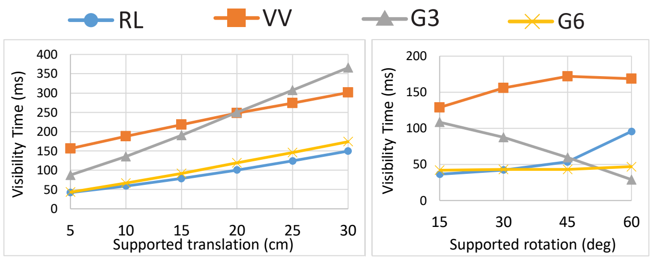


Fig. 14. Average timing (in ms) for walkthroughs of full detailed scenes for camera offsets and supported angles.

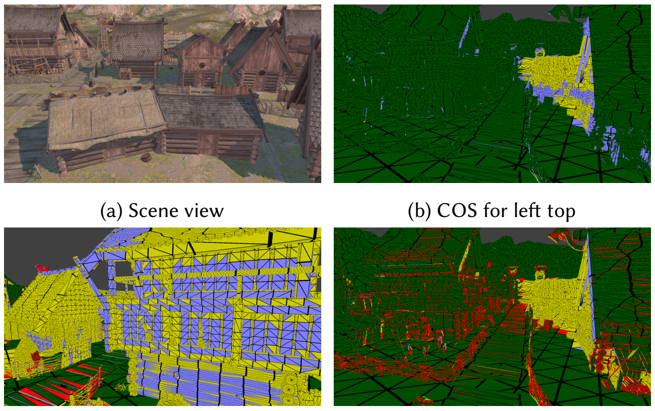


Fig. 15. True positives (green), true negative (yellow), false positives (blue) and false negatives (red) of our COS and the compared Instant Visibility for the view shown in (a). Alternative views for the left top region of (a) are provided in (b-d). Our approach does not show a single false negative, while IV misses many triangles and has similar amounts of false positives.

6.1 Inner workings

Fig. 17 shows challenging example views. Our algorithm discards occluded geometry and novel views are correctly rendered from the PVS. Consider the example in the third column of Fig. 17, the roof occludes large parts of the scene and no geometry behind the roof is classified as visible. A slight movement or rotation to either side opens up the view to the mountain in the back. The respective triangles are clearly classified as potentially visible.

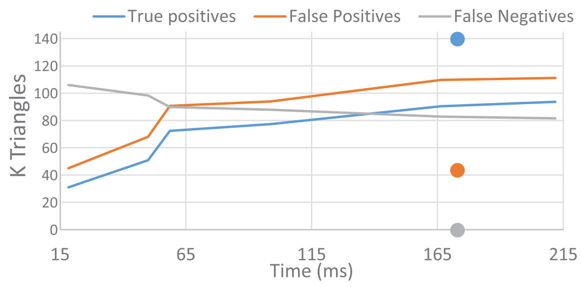


Fig. 16. Running IV on a higher resolution only marginally increases true positives, while increasing time cost and false positive count, as shown for the evaluation on VV in this example. COS (shown in dots) per design has a higher initial cost, but is completely conservative with lower false positives.

The COS works at any resolution when considering between-sample locations. However, different resolutions influence performance and PVS overestimation. The extreme case—a single sample—gathers all triangles in a single list. The results for different COS resolutions (Fig. 13) show that the PVS size stays mostly constant, which indicates that our approach always captures all necessary geometry. For RL and VV, PVS size slightly decreases with increasing resolution, as the approximations and conservative accepts are reduced. G3 and G6 behave differently. Those are CAD scenes and contain many regular triangle grids (Fig. 12a). The denser the gathering resolution, the more triangles fall onto the COS boundary, yielding more conservative accepts. The execution time increases significantly with higher resolution as many more lists are processed. Thus, the overall best resolution seems to be the smallest tested resolution, which we use for all further testing. Further lowering the resolution made the performance significantly worse.

In addition to the COS-resolution, the ranges of camera offsets and rotations influence performance. Fig. 14 shows that increasing the view cell size also increases the visibility resolution approximately linearly. This is not surprising, as for larger view cells triangles are visible at more sample locations, which increases list lengths and visibility resolution needs to work through more triangles. The supported camera rotation has less influence on the timing, as the number of samples stays the same; they are simply spread out more in the scene. Thus, depending on the scene, the execution time can increase, decrease or even fluctuate for different ranges of rotations.

6.2 Performance and Comparison

Table 1 shows a detailed comparison against Instant Visibility (IV). To determine whether triangles are classified correctly, we use a sampled ground truth PVS, for which we render 216 views in 4K resolution distributed over the view cell and add all triangles that are visible in any view to the PVS. While this approach captures a large amount of geometry, it still might miss small triangles. Thus the actual PVS might be slightly larger than our ground truth. However, computing an exact ground truth PVS is hardly achievable within realistic computation time and suffers from floating point errors.

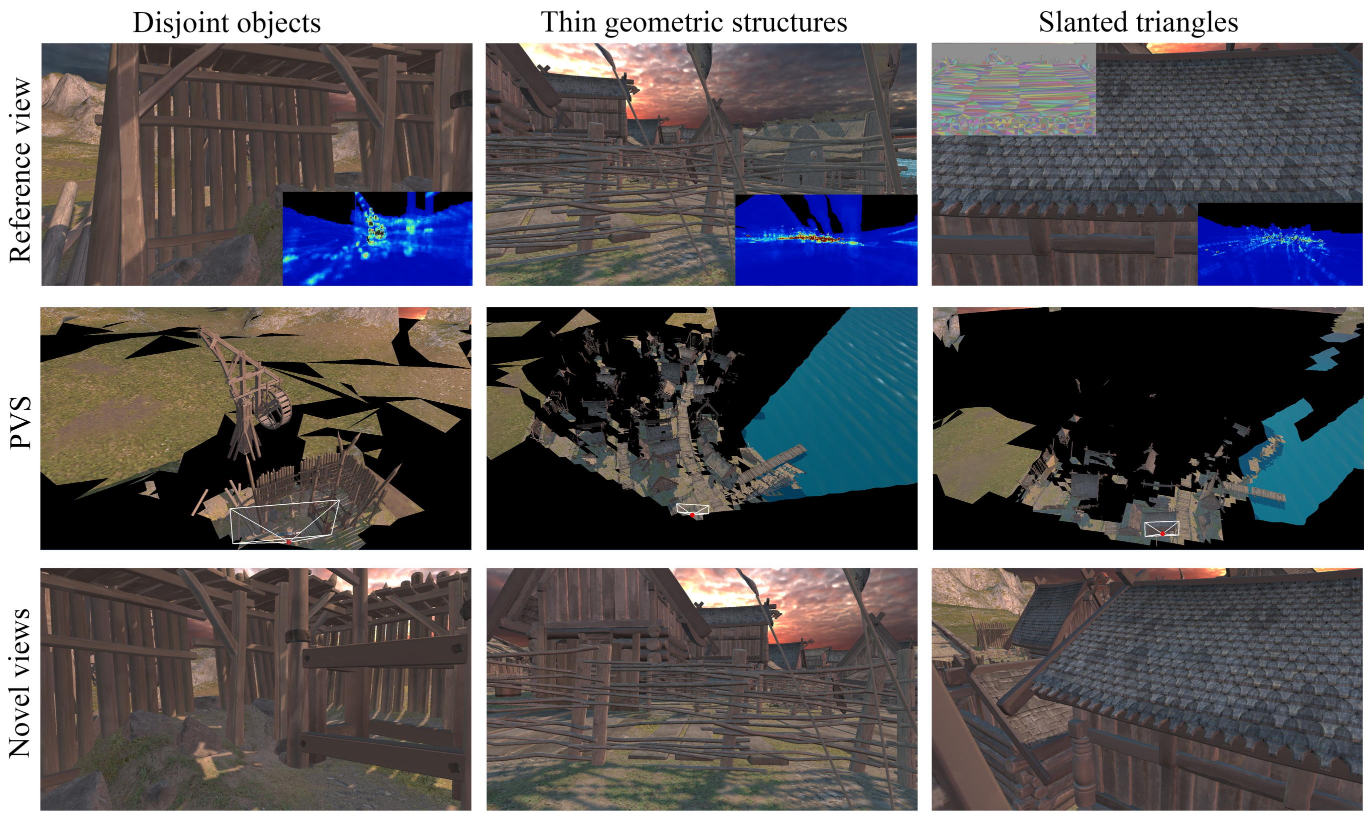


Fig. 17. Three challenging scenarios that our method handles correctly: The first row is the reference view; the second row shows the calculated PVS (red dot shows the camera position in reference view); and the insets show heat-maps of the fragment list length with low (blue) of 1 and high (red) of 256/2048/2048 entries. The slanted triangles case shows an additional inset indicating triangle topology. The novel views show that all geometry is contained in the PVS.

Our PVS algorithm reduces the triangle count from 472k down to 50k to 100k for RL, from 4.7M down to 170k to 270k for VV, from 5.8M down to 80k to 100k for G3, from 1.5M down to 35k to 50k for G6. In all our tests our approach always marks all required triangles as potentially visible, not missing a single triangle. However, our PVS shows an overestimation between 20 to 90%. Note that some of these triangles are likely to be true positives, as we still observed small triangles missing from our sampled ground truth. For the full scenes, our PVS computations takes between 35 to 220ms. On the LOD models, we reduce the 23k, 138k, 428k, and 117k triangles down to 5k, 10k, 9k, and 7k, respectively. For the LOD scenes, we have a false positive rate of up to 100%. Execution times are between 12 to 38ms. While the full scenes may result in a low server frame rate, the LOD times are certainly sufficient for real-time streaming in our use case (note that the client frame rate is independent).

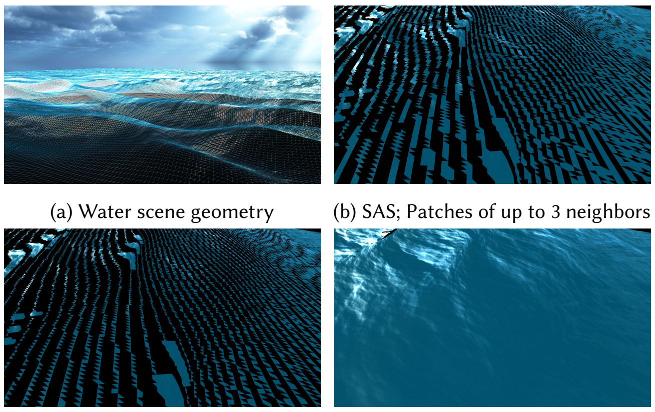
While our approach never misses a single triangle, it potentially overestimates the PVS by a significant margin. However, the sampling approach taken by Instant Visibility [Wonka et al. 2001] applied to a simple triangle stream cannot produce useful results. As can be seen, the number of false negatives is most often significantly larger than the true positives, *i.e.*, missing more triangles than are being captured. This is a typical issue when using sampled visibility to classify small geometry—simply too much geometry is missed (also shown in Fig. 15). Due to shrinking, the number of false positives is also higher than the true positives and most often in a similar order as in our approach. Of course, the run time of IV is rather short with 3 to 23ms for the original scenes and 1.7 to 3.3ms for the LODs. Providing IV with more time by increasing the resolution only marginally improves its performance (see Fig. 16), as it is not suitable for determining visibility of a triangle stream.

To further underline the need and effectiveness of resolving the PVS for in-between sample locations, we show a rendering from a finely tessellated water scene (Fig. 18). As can be seen, holes are present in the PVS when in-between sample locations are ignored. These holes become visible under small camera rotations. However, including the half pixel offset into all tests in the COS leads to a complete PVS for translation and rotation. In comparison, SAS [Mueller et al. 2018] suffers from such artifacts, as their PVS is computed by sampling patches of up to 3 neighboring triangles, missing patches that fall in between the samples.

To analyze the performance of our approach in detail, Fig. 19 shows the timings for each step of our algorithm. Clearly, sorting, fragment list generation, binning and copy out are very efficient. Considering the amount of primitives in the fragment lists our complete visibility resolution implementation is reasonably efficient. However, optimizations in this stage are still possible.

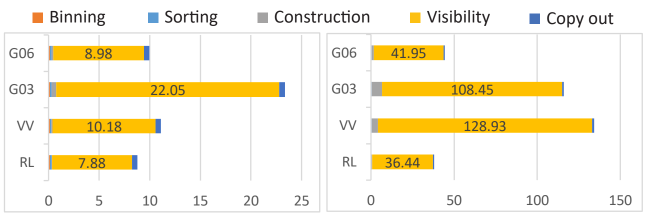
Table 1. Timing and quality breakdown for walkthroughs for all scenes. COS resolution is 200 × 100, FOV is 60°. Timing for the whole pipeline are given in ms, true positives, false positives and false negatives are compared to a ground truth obtained using high-resolution sampling. List entries capture the overall list entries stored for the per-fragment linked lists, list lengths are provided on a per list basis. All measurements are averaged over the whole walkthrough.

						COS				Instant Visibility			
		triangles	C	list entries	list max / avg	time	true pos	false pos	false neg	time	true pos	false pos	false neg
Original	RL	472k	5.00cm	590k	8573 / 28	96.98	60k	36k	0	3.76	16k	18k	44k
			30.00cm	1.44M	10581 / 71	161.76	68k	26k	0	3.72	17k	17k	51k
	VV	4.7M	5.00cm	1.63M	109675 / 131	175.53	137k	42k	0	18.34	31k	45k	106k
			30.00cm	2.7M	117461 / 207	227.34	154k	25k	0	18.32	32k	45k	123k
	G3	5.8M	5.00cm	828k	144616 / 40	38.10	50k	43k	0	22.30	5k	11k	45k
			30.00cm	2.5M	190161 / 125	78.49	56k	49k	0	22.31	6k	10k	50k
	G6	1.5M	5.00cm	612k	24597 / 30	50.11	24k	17k	0	7.51	3k	5k	21k
			30.00cm	1.9M	32311 / 95	104.12	27k	23k	0	7.50	3k	4k	24k
LOD	RL	23k	5.00cm	109k	496 / 4	11.92	2197	2106	0	1.78	1006	1711	1191
			30.00cm	257k	627 / 12	18.87	2506	2863	0	1.70	1086	1743	1420
	VV	138k	5.00cm	81k	3014 / 5	22.21	5336	4475	0	2.21	2559	3822	2777
			30.00cm	178k	3787 / 13	33.47	5939	5883	0	2.22	2665	3966	3274
	G3	428k	5.00cm	248k	11940 / 11	19.90	4198	3474	0	3.31	1165	1822	3033
			30.00cm	680k	16662 / 33	38.06	4865	4669	0	3.33	1278	1741	3587
	G6	117k	5.00cm	118k	1974 / 5	12.52	3192	3236	0	2.15	672	1114	2520
			30.00cm	333k	4636 / 16	21.86	3782	3466	0	2.14	749	1074	3033


(a) Water scene geometry (b) SAS; Patches of up to 3 neighbors

(c) COS; Sample locations only (d) COS; Between sample locations

Fig. 18. (a) A scene with very small triangles highlights the need to consider between sample locations. A different perspective on the regions in the back reveals missed triangles when considering pixel centers only (b,c). After including the half-pixel offset the COS, PVS becomes watertight, adding support for any translation and rotation within the supported view cell.



(a) LOD geometry (b) Full geometry

Fig. 19. Step timings for the different stages of our algorithm in ms.

7 CONCLUSION AND FUTURE WORK

Starting from considerations about under which camera offsets a triangle covers a given pixel, we have introduced the camera offset space (COS). In COS, this coverage is a simple geometric shape. By deriving a function that describes the depth of the triangle under all camera movements, it is possible to determine whether one triangle will cover another for various camera offsets. Extending this approach to all samples of a rendered scene and locations in-between samples, we have shown that a complete potentially visible set (PVS) can be constructed. As with all PVS algorithms, occluder fusion is of high importance, which we integrate for connected surfaces that we extract on the fly.

By implementing our visibility algorithm in a combination of OpenGL and CUDA, we have presented interactive rates for online PVS generation, especially when using LOD geometry for complex scenes. Testing indicates that our approach is very reliable and always captures the complete PVS. Our approach is the first PVS algorithm that works on-the-fly and without preprocessing. All information is computed directly from the triangle input, thus, dynamic objects and connectivity changing are supported.

In the future, we aim to address all other aspects of the proposed streaming rendering pipeline. While we support dynamic objects, their movement from the current PVS construction until the rendering takes place is not considered in the PVS. Movement prediction could be incorporated directly into COS and also transmitted to the client. Our COS is also applicable in a multi-framerate rendering system, where server and client run on the same machine, performing frame-rate upsampling while reducing overall GPU load.

ACKNOWLEDGMENTS

This research was supported by the Max Planck Center for Visual Computing and Communication, by the German Research Foundation (DFG) grant STE 2565/1-1, the Austrian Science Fund (FWF) grant I 3007, and NVIDIA Corporation who donated the GPU used for this research.

REFERENCES

John M. Airey, John H. Rohlf, and Frederick P. Brooks, Jr. 1990. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. *SIGGRAPH Comput. Graph.* 24, 2 (Feb. 1990), 41–50.

Jiri Bittner, Vlastimil Havran, and Pavel Slavik. 1998. Hierarchical visibility culling with occlusion trees. In *Computer Graphics International, 1998. Proc.* 207–219.

Jiří Bittner, Oliver Mattausch, Peter Wonka, Vlastimil Havran, and Michael Wimmer. 2009. Adaptive Global Visibility Sampling. *ACM TOG* 28, 3, Article 94 (July 2009), 10 pages.

Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. 2004. Coherent hierarchical culling: Hardware occlusion queries made useful. In *CGF*, Vol. 23. 615–624.

Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen. 2001. Unstructured Lumigraph Rendering. In *Proc. SIGGRAPH (SIGGRAPH '01).* 425–432.

Christopher A. Burns, Kayvon Fatahalian, and William R. Mark. 2010. A Lazy Object-space Shading Architecture with Decoupled Sampling. In *Proc. High Performance Graphics (HPG '10).* 19–28.

Chun-Fa Chang and Shyh-Haur Ger. 2002. Enhancing 3D Graphics on Mobile Devices by Image-Based Rendering. In *Proc. of the Third IEEE Pacific Rim Conference on Multimedia: Advances in Multimedia Information Processing (PCM '02).* 1105–1111.

Shenchang Eric Chen and Lance Williams. 1993. View Interpolation for Image Synthesis. In *Proc. SIGGRAPH (SIGGRAPH '93).* 279–288.

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

Satyan Coorg and Seth Teller. 1999. Temporally Coherent Conservative Visibility. *Comput. Geom. Theory Appl.* 12, 1-2 (Feb. 1999), 105–124.

Thomas H Cormen. 2009. *Introduction to algorithms.* MIT press.

Wagner T. Correa, James T. Klosowski, and Claudio T. Silva. 2003. Visibility-Based Prefetching for Interactive Out-Of-Core Rendering. In *Proc. PVG (PVG '03).* 2–.

Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. 1996. Modeling and Rendering Architecture from Photographs: A Hybrid Geometry- and Image-based Approach. In *Proc. SIGGRAPH (SIGGRAPH '96).* 11–20.

Piotr Didyk, Elmar Eisemann, Tobias Ritschel, Karol Myszkowski, and Hans-Peter Seidel. 2010. Perceptually-motivated Real-time Temporal Upsampling of 3D Content for High-refresh-rate Displays. *CGF (Proc. Eurographics 2010)* 29, 2 (2010), 713–722.

Frédo Durand, George Drettakis, Joëlle Thollot, and Claude Puech. 2000. Conservative visibility preprocessing using extended projections. In *Proc. Computer graphics and interactive techniques.* 239–248.

Thomas A Funkhouser. 1996. Database management for interactive display of large architectural models. In *Graphics Interface*, Vol. 96. 1–8.

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

Naga K. Govindaraju, Avneesh Sud, Sung-Eui Yoon, and Dinesh Manocha. 2003. Interactive Visibility Culling in Complex Environments Using Occlusion-switches. In *Proc. of the 2003 Symposium on Interactive 3D Graphics (I3D '03).* 103–112.

Jon Hasselgren, Tomas Akenine-Möller, and Lennart Ohlsson. 2005. Conservative rasterization. *GPU Gems* 2 (2005), 677–690.

Karl E. Hillesland and J. C. Yang. 2016. Texel Shading. In *EG 2016 - Short Papers*, T. Bashford-Rogers and L. P. Santos (Eds.). The Eurographics Association.

Jozef Hladky, Hans-Peter Seidel, and Markus Steinberger. 2019. Tessellated Shading Streaming. *Computer Graphics Forum* 38, 4 (2019), 12.

Lichan Hong, Shigeru Muraki, Arie Kaufman, Dirk Bartz, and Taosong He. 1997. Virtual voyage: Interactive navigation in the human colon. In *Proc. Computer graphics and interactive techniques.* 27–34.

Junxian Huang, Feng Qian, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. 2012. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *Proc. of International Conference on Mobile Systems, Applications, and Services (MobiSys '12).* 225–238.

Tom Hudson, Dinesh Manocha, Jonathan Cohen, Ming Lin, Kenneth Hoff, and Hansong Zhang. 1997. Accelerated occlusion culling using shadow frusta. In *Proc. Computational geometry.* 1–10.

WFH Jiménez, Claudio Esperança, and Antonio AF Oliveira. 2000. Efficient algorithms for computing conservative portal visibility information. In *CGF*, Vol. 19. 489–498.

Vladlen Koltun, Yiorgos Chrysanthou, and Daniel Cohen-Or. 2001. Hardware-accelerated from-region visibility using a dual ray space. In *Rendering Techniques 2001.* 205–215.

Johannes Kopf, Fabian Langguth, Daniel Scharstein, Richard Szeliski, and Michael Goesele. 2013. Image-based rendering in the gradient domain. *ACM TOG* 32, 6 (2013), 1–9.

Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Alec Wolman, Yury Degtyarev, Sergey Grizan, and Jason Flinn. 2015. Outatime: Using Speculation to Enable Low-Latency Continuous Interaction for Mobile Cloud Gaming. *GetMobile: Mobile Comp. and Comm.* 19, 3 (Dec. 2015), 14–17.

Tommer Leyvand, Olga Sorkine, and Daniel Cohen-Or. 2003. Ray Space Factorization for From-region Visibility. *ACM TOG* 22, 3 (July 2003), 595–604.

Gerrit Lochmann, Bernhard Reinert, Tobias Ritschel, Stefan Müller, and Hans-Peter Seidel. 2014. Real-time Reflective and Refractive Novel-view Synthesis, Jan Bender, Arjan Kuijper, Tatiana von Landesberger, Holger Theisel, and Philipp Urban (Eds.). Eurographics Association, Darmstadt, Germany, 9–16.

David Luebke and Chris Georges. 1995. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proc. Interactive 3D graphics.* 105–ff.

William R. Mark, Leonard McMillan, and Gary Bishop. 1997. Post-rendering 3D warping. *Symposium on Interactive 3D Graphics* Figure 2 (1997), 7–16.

Joerg H. Mueller, Philip Voglreiter, Mark Dokter, Thomas Neff, Mina Makar, Markus Steinberger, and Dieter Schmalstieg. 2018. Shading Atlas Streaming. *ACM TOG* 37, 6, Article 199 (Dec. 2018), 16 pages.

Diego Nehab, Pedro V. Sander, Jason Lawrence, Natalya Tatarchuk, and John R. Isidoro. 2007. Accelerating Real-time Shading with Reverse Reprojection Caching. In *Proc. Symposium on Graphics Hardware (GH '07).* 25–35.

Marc Olano and Trey Greer. 1997. Triangle scan conversion using 2D homogeneous coordinates. In *Proc. workshop on Graphics hardware.* 89–95.

Bernhard Reinert, Johannes Kopf, Tobias Ritschel, Eduardo Cuervo, David Chu, and Hans-Peter Seidel. 2016. Proxy-guided Image-based Rendering for Mobile Devices. *CGF* 35, 7 (2016), 353–362.

Gernot Schaufler, Julie Dorsey, Xavier Decoret, and François X Sillion. 2000. Conservative volumetric visibility with occluder fusion. In *Proc. Computer graphics and interactive techniques.* 229–238.

Daniel Scherzer, Lei Yang, Oliver Mattausch, Diego Nehab, Pedro V. Sander, Michael Wimmer, and Elmar Eisemann. 2011. A Survey on Temporal Coherence Methods in Real-Time Rendering. In *EUROGRAPHICS 2011 State of the Art Reports.* 101–126.

Shu Shi and Cheng-Hsin Hsu. 2015. A Survey of Interactive Remote Rendering Systems. *ACM Comput. Surv.* 47, 4, Article 57 (May 2015), 29 pages.

Sudipta N. Sinha, Johannes Kopf, Michael Goesele, Daniel Scharstein, and Richard Szeliski. 2012. Image-based rendering for scenes with reflections. *ACM TOG* 31, 4 (2012), 1–10.

Pitchaya Sitthi-amorn, Jason Lawrence, Lei Yang, Pedro V. Sander, Diego Nehab, and Jiahe Xi. 2008. Automated Reprojection-based Pixel Shader Optimization. *ACM TOG* 27, 5, Article 127 (Dec. 2008), 11 pages.

Seth Teller and Pat Hanrahan. 1993. Global Visibility Algorithms for Illumination Computations. In *Proc. SIGGRAPH (SIGGRAPH '93).* 239–246.

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

George Wolberg. 1998. Image morphing: a survey. *The Visual Computer* 14, 8-9 (1998), 360–372.

Peter Wonka, Michael Wimmer, and Dieter Schmalstieg. 2000. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Rendering Techniques 2000.* 71–82.

P. Wonka, M. Wimmer, and F. X. Sillion. 2001. Instant visibility. *CGF* 20, 3 (9 2001).

Jason C. Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. 2010. Real-time Concurrent Linked List Construction on the GPU. In *Proc. EGSR (EGSR'10).* 1297–1304.

Lei Yang, Yu-Chiu Tse, Pedro V. Sander, Jason Lawrence, Diego Nehab, Hugues Hoppe, and Clara L. Wilkins. 2011. Image-based Bidirectional Scene Reprojection. *ACM TOG* 30, 6, Article 150 (Dec. 2011), 10 pages.

Hansong Zhang, Dinesh Manocha, Tom Hudson, and Kenneth E. Hoff. 1997. *Visibility culling using hierarchical occlusion maps.* ACM, 77–88.

Henning Zimmer, Fabrice Rousselle, Wenzel Jakob, Oliver Wang, David Adler, Wojciech Jarosz, Olga Sorkine-hornung, and Alexander Sorkine-hornung. 2015. Path-space Motion Estimation and Decomposition for Robust Animation Filtering. *Egsr 2015* 34, 4 (2015), 12.