Complex VEs on All-In-One VR Headsets Through Continuous From-Segment Visibility Computation

Voicu Popescu* Purdue University
Elisha Sacks† Purdue University
Zirui Zhang‡ Purdue University
Jorge Vazquez§ Purdue University

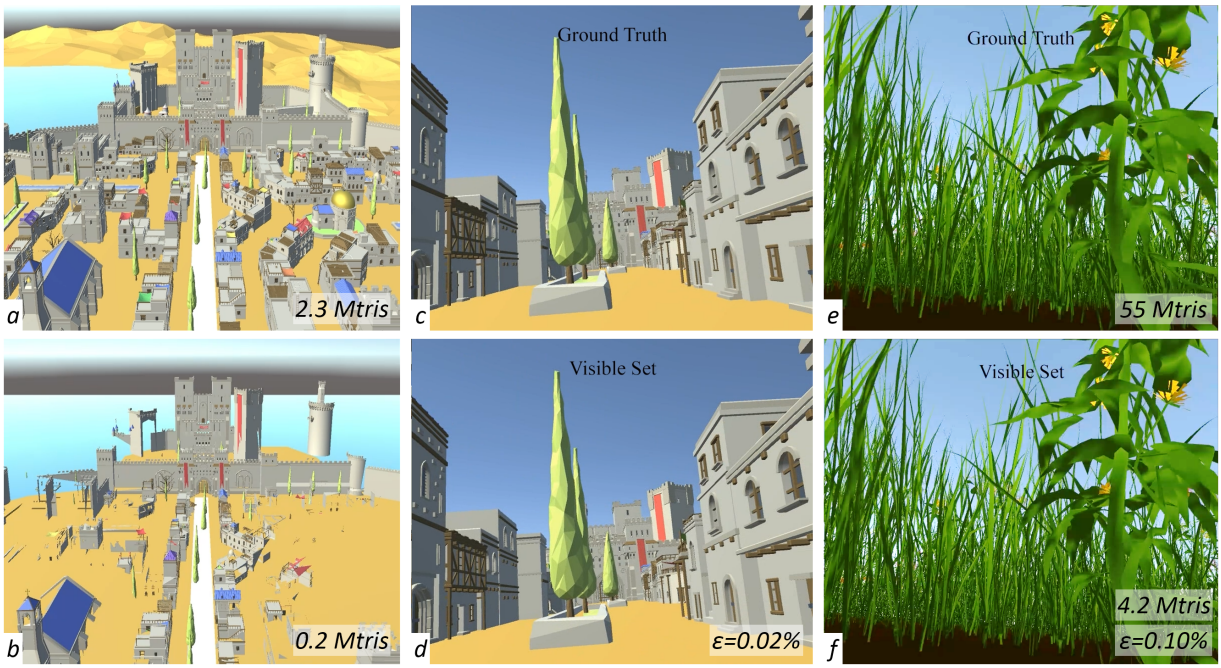

Figure 1: The *Medieval* virtual environment (*a*) with 2.3 million triangles (Mtris) is reduced to 0.2 Mtris (*b*) visible from the central street (white rectangle), using our algorithm. A frame (*d*) rendered from the visible set has a low pixel error ε of 0.02% and is virtually indistinguishable from the ground truth frame (*c*) rendered from the original virtual environment. The *Grass* virtual environment with 55 Mtris is reduced to a visible set of 4.2 Mtris that yields a frame (*f*) with a low ε of 0.10% over ground truth (*e*).

Abstract

All-in-one VR headsets have limited rendering power which limits the complexity of the virtual environments (VEs) that can be used in VR applications. This paper describes a novel visibility algorithm for making complex VEs tractable on all-in-one VR headsets. Given a view segment, the algorithm finds the set of triangles visible as a camera translates on the view segment. When run on the perimeter of a user view region, the algorithm provides a quality approximation of the visible set from inside the view region. The visibility algorithm supports static and dynamic VEs, and it solves visibility with either triangle, particle, or object granularity. The visible sets yield output frames that are virtually indistinguishable from ground truth frames rendered from the original VEs.

Index Terms: VE complexity reduction, all-in-one virtual reality headsets, visibility computation.

*e-mail: popescu@purdue.edu
†e-mail: eps@purdue.edu
‡e-mail: zhan4192@purdue.edu
§e-mail: vazque81@purdue.edu

1 Introduction

Recent hardware and software advances have brought all-in-one virtual reality (VR) headsets with on-board inside-looking-out tracking, rendering, and power, which provide a completely untethered VR experience, free of limited tracked area, video cable, and power cord constraints. This freedom comes at the cost of graphics processing units (GPUs) that can only render a fraction of the triangles that their power-hungry desktop counterparts can. In turn, the reduced rendering power limits the complexity of virtual environments (VEs) that such thin VR clients can render. The problem of reducing the complexity of 3D datasets without reducing the richness of the visual experience is as old as interactive computer graphics, owing to the limited rendering capability of early graphics systems. The problem has received renewed interest in the context of VR headsets, not just because their GPUs are less powerful, but also because this is unlikely to change in the near future. Indeed, it is unlikely that GPU energy efficiency, battery energy density, or heat dissipation strategies will quickly improve sufficiently to allow all-in-one headsets to incorporate desktop-class GPUs.

There are three complementary fundamental approaches for making complex VEs tractable on all-in-one VR headsets. One approach is to replace the original VE with a VE of lesser geometric complexity but that yields frames indistinguishable from those rendered from the original VE. This problem of geometric simplification remains open, with challenges that include maintaining visual quality while meeting a prescribed rendering budget, and

achieving a gradual change in level of detail as the user's viewpoint changes. Developers of VR applications often resort to a drastic simplification of the VE. A second approach is for the VR headset to get help from a server. If the server is to perform all rendering, the network has to be traversed twice for each frame, once for the headset to inform the server of the desired view, and once for the server to send the rendered frame to the headset. This results in a latency unacceptable in the context of VR, where it causes cybersickness. Therefore, one has to partition the rendering load between the server and the headset, in a way that shields the headset from the full complexity of the VE, while hiding the network latency.

A third approach is visibility computation, which determines the parts of the VE that the user can see from a given region. For many complex VEs, the resulting visible set is a small fraction of the original VE, so it can be handled by the VR headset. However, visibility computation is also an open problem, as there are no practical algorithms that can find the set of all VE parts visible from the given view region. Visibility algorithms are categorized based on how they analyze the space of visibility rays emanating from the view region. Sample-based visibility algorithms probe the VE with a discrete set of visibility rays. Such algorithms are fast, but the visibility rays considered are selected heuristically, which could miss visible parts of the environment. A second category of visibility algorithms analyze the space of visibility rays continuously, by considering a continuum–and not just a discrete set–of visibility rays. The goal is an accurate visible set, but, due to the complexity of such algorithms, this goal has only been reached for the simplest of view regions, i.e., a single viewpoint.

In this paper we propose to reduce the complexity of VEs to make them tractable on all-in-one VR headsets with limited rendering capabilities. To do so we introduce a novel continuous visibility computation algorithm that finds all VE triangles that are visible along a linear camera path. In other words, the algorithm finds all and only the triangles that are visible at pixel centers as the camera translates. Figure 1 shows that our approach achieves an 11.5× complexity reduction for the *Medieval* VE (from *a* to *b*), and of 13× for *Grass* (*e-f*). An all-in-one VR headset (i.e., a Quest 3 [1]) renders the original *Medieval* VE at 23 fps, and our visible set at 71 fps, a speedup of 3×, approaching the Quest 3 display frame rate of 72 fps. The Quest 3 cannot render the original *Grass* VE, and can render our visible set at 39 fps. Our approach reduces complexity while maintaining frame quality. For *Medieval*, compared to the ground truth frame *c* rendered from the original VE, our frame *d* has a pixel error ε of 0.02%, a peak signal to noise ratio [19] *PSNR* of 50.9 dB, and a structural similarity index measure [46] *SSIM* of 0.999999. For *Grass*, comparing our frame *e* to ground truth *f* yields an ε of 0.10%, a *PSNR* of 35.5 dB, and an *SSIM* of 0.999994 (see Fig. 6 for a magnified comparison).

Our algorithm generalizes the conventional z-buffer from a single value per pixel to a list of triangles visible at the pixel center as the camera translates. The visible triangles and their visibility intervals are computed based on visibility events, i.e., the moments along the linear camera path when a pixel's center switches sidedness with respect to a projected triangle edge. Our algorithm also handles dynamic VEs, with per-vertex piecewise linear translation vectors. For *Swaying Grass*, an animated version of *Grass* with 10 key frames, our approach reduces complexity by a factor of 9.9×, and achieves high quality frames with $\varepsilon = 0.12\%$, $PSNR = 75.8$, and $SSIM = 0.999996$. Our approach makes *Swaying Grass* tractable on the Quest 3, achieving a frame rate of 28 fps. Our algorithm can solve visibility not just with triangle granularity, but also with coarser, object granularity. We have also extended our algorithm to handle spherical particles directly, without having to convert them first to triangle meshes. Even for a coarse particle tessellation with 10 triangles, working directly with particles reduces the visibility computational load by an order of magnitude.

We have evaluated our approach on multiple virtual environments, where it has consistently achieved a significant complexity reduction, high frame rates, and high frame quality as confirmed in comparisons to ground truth. We have also compared our approach to the prior art approach of accumulating visible triangles from intermediate positions along the camera segment; results show that our approach is more efficient, finding more complete visible sets faster. Our approach enables deploying on all-in-one VR headsets virtual environments that were previously reserved to VR systems tethered to a rendering workstation. We also refer the reader to the accompanying video.

2 Prior Work

Our goal is to reduce the complexity of virtual environments to make them tractable on thin VR clients. We review prior art approaches for VE complexity reduction, with a more detailed discussion of the visibility approach.

2.1 Level-of-Detail (LoD) Adaptation

The goal is to replace detailed objects that have a small screen footprint with simpler versions that produce similar images at a lower cost. The challenges are computing simplified alternative representations for complex geometries, avoiding image quality loss, and transitioning continuously from one LoD to the next as the footprint of the simplified object increases making it warrant a more detailed representation [29]. The simplest, oldest, and still very frequently used LoD scheme in interactive visualization, including in VR, is environment mapping [5]: partition the VE into a near and far region, keep the original geometry in the near region, and render the far region to a cubemap that provides a low cost and visually rich backdrop [38]. Complexity reduction of up to 2× was achieved by rendering distant geometry only once, instead of once for each eye [13], or by rendering the entire VE only once, from the middle of the interpupillary segment, and then reconstructing the left and right eye images by warping [42]. We rely on visibility computation, which is an orthogonal approach to LoD adaption. The approaches can be used in conjunction when the visible set is too large to be rendered on the client and has to be simplified first.

2.2 Distributed VR Systems

The goal is to provide help to thin VR clients from remote servers. Rendering each frame on the server implies a network round-trip per frame. Even if one makes abstraction of the time needed for the server to render the frame, the approach incurs too much latency [51]. Future cellular network standards do incorporate the requirement for ultra-low latency [34], but the approach is not yet feasible. Network latency can be hidden if rendering is partitioned between the server, which renders distant objects with higher latency, and the client, which renders near objects with low latency (e.g, CloudVR [21]). Latency can also be diffused if the frame received from the server is used by the client to create intermediate frames [23]. The server can also help with the shading computational load of expensive, global illumination effects [44]. The CloVR system uses a near-far partitioning scheme and provides continuous progressive refinement at startup and after teleportation by growing the near region in concentric rings [53].

Distributed VR systems are also being investigated by the network community, wishing to extrapolate the success of video servers to interactive visualization applications [28, 22, 12]. One early focus were 360° videos, which are only partially seen by a VR user hence the opportunity for saving bandwidth [41, 16]. A subsequent focus were 3D videos with per-pixel depth (a.k.a., volumetric or free-viewpoint videos), which, unlike triangle meshes, provide easier LoD control and occlusion culling due to their regular structure [40, 14, 27, 49]. Handling triangle meshes has been

attempted by converting the meshes to an intermediate uniform representation, e.g., a collection of environment maps [26, 45, 37], or of compressed animated light fields [25], but the intermediate representations come at the cost of higher bandwidth requirements and of lower quality output frames. The low latency and specialized computing requirements of distributed VR has led researchers to differentiate between high latency cloud servers and responsive edge servers, investigating optimal resource allocation [32, 52]. Our visibility computation approach is compatible with both edge servers, whose graphics computation capabilities, i.e., GPUs, allow computing the visible sets in real time, and with cloud servers, which store visibility solutions precomputed over a spatial subdivision of the total user view region.

2.3 Visibility Computation

For VEs where only a fraction of the VE is visible from a given view region, visibility computation can be a powerful approach for making complex VEs tractable on thin VR clients. There are no modifications to the VR application, the headset just renders a subset of the original VE, and, if all visible triangles are found, the output frames are identical to those obtained from the original VE.

One classification of visibility algorithms is based on which triangles are included in the visible set. If some triangles are in fact never visible from the view region, the algorithm is called *conservative*. The advantage is a high quality output, and the disadvantage is that the visible set isn't as small as it could be [7, 9]. If the visible set contains only but not all visible triangles, the algorithm is called *aggressive*. The advantage is that the application doesn't render any unnecessary triangles, and the disadvantage is that the output image quality suffers when the viewpoint reveals a triangle omitted from the visible set [36, 48]. If the visible set contains all visible triangles, and it does not erroneously include a hidden triangle, the algorithm is called *exact*. The advantage is that there is no error in the output image with the smallest possible visible set, and the disadvantage is algorithm complexity.

Another classification of visibility algorithms is based on how the space of visibility rays originating from the give view region is investigated. *Sample-based* visibility algorithms consider a discrete set of visibility rays, producing an aggressive visible set since a ray never returns a hidden triangle, but also since some triangles could be missed. Their advantage is simplicity and efficiency, their disadvantage is output image error. *Continuous* visibility algorithms consider a continuum of rays. Their advantage is an exact visible set over the ray continuum, and their disadvantage is complexity.

Conservative visibility algorithms replace occluders with simpler shapes whose occlusion shadow is easier to compute [11, 8], which simplifies occlusion culling, i.e., the process of discarding multiple hidden triangles at once [4, 2]. Since the approximate occluder underestimates the original occluder, some hidden triangles are incorrectly labeled as visible. Overestimating occluder geometry leads to an aggressive visibility solution [50].

Aggressive Visibility algorithms probe for visible triangles with individual [48, 3, 24] or bundles [31, 30, 43] of rays. Individual rays bring the advantage of flexibility to cast only the rays deemed necessary, ray bundles bring the advantage of amortizing the cost of a ray leveraging the coherence of the rays within the bundle. The challenge is to decide which rays to use to probe for visible triangles, decision that is based on heuristics [24]. Furthermore, such a sample-based algorithm can only confirm that a triangle is visible, and it can never confirm that a triangle is hidden, as this would require an infinite number of rays to confirm that the triangle is not visible at any of its points.

Exact Visibility algorithms analyze the space of visibility rays continuously. For the simplest view region, i.e., a single viewpoint, the space of visibility rays is 2D. One approach for computing from-point visibility is to build a polygonal subdivision of this 2D space into regions where a single triangle is visible [39]. The approach analyzes the 2D space of visibility rays continuously, finding all triangles visible from the given viewpoint, regardless of the view direction. Computing from-point visibility repeatedly was shown [39] to approximate from-region visibility more efficiently than state of the art sample-based visibility [24]. By contrast, our approach computes visibility for a view segment, which yields a 3D space of visibility rays, which it analyzes continuously along the translation degree of freedom and discretely along the two rotation degrees of freedom. As shown in the results section, our from-segment visibility computation algorithm allows finding more visible triangles faster than from-point visibility. Another approach for exact from point visibility is based on beam tracing [17], which has the advantage of being differentiable, as needed for example in inverse graphics [54].

For a general, 3D view region the space of visibility rays is 5D, which was explored by theoretical computational geometry [10, 15, 33, 6], but no practical implementations exist for complex VEs. For example, one approach is to represent the visibility rays between two polygons in a 5D Euclidian space derived from Pluecker space [35]. The approach was demonstrated for VEs of one million triangles, for a small view cell, with a large computational overhead, and without support for dynamic scenes. Although the general visibility problem for a 3D view region has been well understood for a long time, visibility computation research has focused on computationally tractable approximate solutions. The *camera offset space* approach to visibility [18] analyzes the camera translations, i.e., "offsets", under which a triangle covers a given image plane point, i.e., a pixel center. The 3D space of offsets, and the 2D space of image plane points amount to the same 5D complexity as the space of visibility rays of a 3D view region. To make the implementation practical, the intersection of the offset spaces of two triangles is approximated conservatively, which avoids missing visible triangles, but includes triangles that are not visible, making the algorithm conservative, and not exact.

Our algorithm computes visibility for a 1D view region, i.e., a view segment, finding all and only the triangles visible as the camera translates along the view segment. Our algorithm analyzes visibility continuously at each pixel center in the 1D space of the camera translation, and it takes a sample-based approach in the 2D space of the rays at a viewpoint along the view segment. Our algorithm can be used as an efficient visibility computation primitive to approximate visibility for higher dimensional view regions.

3 Continuous From-Segment Visibility

Given a virtual environment modeled with triangle meshes, a user view modeled with a camera, and a camera translation segment, we have developed an algorithm that finds all and only VE triangles that are visible at the camera's pixel centers as the camera moves along the translation segment. We first describe the algorithm for a static VE (Sec. 3.1), then we describe its extension to dynamic VEs (Sec. 3.2), and finally we describe its extension to using spherical particles as first-order visibility computation primitives (Sec. 3.3).

3.1 Static Virtual Environments

Given a static VE, a view segment $\mathbf{v_0v_1}$, and a camera c, we want to find all triangles τ in VE for which there exists a viewpoint $\mathbf{v}$ in $\mathbf{v_0v_1}$ and a pixel p in c such that τ is visible from $\mathbf{v}$ at $\mathbf{p}$. A triangle τ is visible from $\mathbf{v}$ at $\mathbf{p}$ if the line segment $\mathbf{vp}$ does not intersect any other triangle. What is needed is an algorithm that finds all triangles visible at each pixel center as the camera translates. Visibility changes at a pixel center when the pixel center changes sidedness with respect to the projection of a triangle edge, as the camera translates as shown in Fig. 2, left. Our algorithm computes all such visibility events for each pixel and builds a list of visibility intervals where a single triangle is visible (Fig. 2, right).

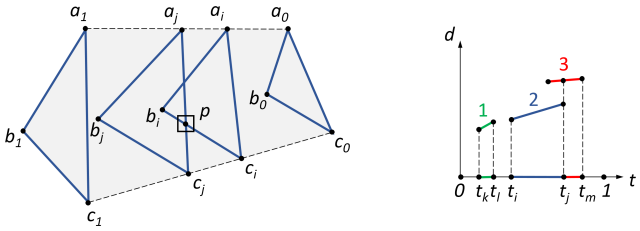

Figure 2: *Left:* Image footprint of a triangle as the camera translates; the triangle projection moves from $a_0b_0c_0$ to $a_1b_1c_1$, and it covers pixel center from when b_ic_i crosses p to when a_jc_j crosses p. *Right:* illustration of the list of visibility intervals at a pixel center, with camera translation parameter t on the x axis, and the distance d from the camera eye to the triangle on the y axis; triangle 1 is visible when $t \in [t_k,t_l]$, 2 when $t \in [t_i,t_j]$, and 3 when $t \in [t_j,t_m]$.

Our algorithm is akin to rendering the VE, but from a view segment as opposed to a view point. The pixels touched by a triangle as the camera translates on the view segment are computed as the 2D convex hull of the projections of the vertices from the segment endpoints. For each pixel, the translation interval when the pixel center is covered by the current triangle is computed using visibility event equations that find the time when the pixel center changes sidedness with respect to a triangle edge. The interval is depth composited with the current list of visibility intervals at the pixel. Once all triangles are processed, the visible set is collected from the final visibility interval lists at the pixels. A pseudocode description of our algorithm is given in Alg. 1. The algorithm initializes the pixel visibility interval lists to empty (lines 1-2), then it computes the visibility of each triangle (lines 3-9), to finally collect the visible triangles from the pixel visibility interval lists (lines 10-13).

(lines 3-9) The visibility of a triangle is computed in two steps: (1) compute the pixels whose centers are covered by the triangle projection (lines 4-6), and (2) update the visibility interval list of each covered pixel (lines 7-9).

(lines 4-6) The pixels covered by a triangle as the camera translates are found by projecting the triangle from both ends of the view segment (lines 4 and 5), and by computing the 2D convex hull of the two triangle projections (line 6). In Fig. 2, left, the convex hull corresponds to the inside of pentagon $\mathbf{a_0a_1b_1c_1c_0}$ (gray). The convex hull is a conservative estimate of the image area touched by the triangle as the camera translates. We prove this statement leveraging epipolar geometry. When the camera translates from $\mathbf{v_0}$ to $\mathbf{v_1}$, each projected vertex moves on a straight line, i.e., on its epipolar line. Since a convex hull contains any segment defined by two of its points, the convex hull will contain the projected vertex trajectories, i.e., segments $\mathbf{a_0a_1}$, $\mathbf{b_0b_1}$, and $\mathbf{c_0c_1}$ in Fig. 2, left. This means that for any intermediate viewpoint v the convex hull will contain the three projected vertices, and therefore the entire triangle projection.

Algorithm 1 Continuous from-segment visibility for static VEs

Input: VE set of triangles S, camera C, view segment $\mathbf{v_0v_1}$
Output: Visible set S_v (triangles from S seen by C from $\mathbf{v_0v_1}$)

1: **for** each pixel $p \in C$ **do**
2: $p.L = \emptyset$
3: **for** each triangle $T_i \in S$ with 3D vertices **abc do**
4: $\mathbf{a_0b_0c_0}$ = Project(**abc**, C, $\mathbf{v_0}$)
5: $\mathbf{a_1b_1c_1}$ = Project(**abc**, C, $\mathbf{v_1}$)
6: H = ConvexHull($\mathbf{a_0}$, $\mathbf{b_0}$, $\mathbf{c_0}$, $\mathbf{a_1}$, $\mathbf{b_1}$, $\mathbf{c_1}$)
7: **for** each pixel center $\mathbf{p} \in H$ **do**
8: $[(t_s,d_s),(t_f,d_f)]$ = CoverageInterval(**p**, **abc**, $\mathbf{v_0}$, $\mathbf{v_1}$)
9: $p.L$ = DepthComposite($p.L$, $[(t_s,d_s),(t_f,d_f)]$, i)
10: $S_v = \emptyset$
11: **for** each pixel $p \in C$ **do**
12: **for** each visibility interval $q \in p.L$ **do**
13: $S_v = S_v \cup q.triID$
14: **return** S_v

(lines 7-9) The algorithm iterates over the pixels inside the convex hull. We denote the viewpoint translation between $\mathbf{v_0}$ and $\mathbf{v_1}$ using a parameter t, i.e., $\mathbf{v} = \mathbf{v_0} + (\mathbf{v_1} - \mathbf{v_0})t$, with $t \in [0,1]$. For each pixel center $\mathbf{p}$, the coverage interval is computed as the translation interval $[t_s,t_f] \subseteq [0,1]$ when triangle **abc** covers $\mathbf{p}$ (line 8).

(line 8) The coverage interval is computed by investigating any visibility event generated by the three triangle edges. An edge generates a visibility event if its projection passes over the pixel center. Given a triangle edge **ab**, where **a** and **b** are 3D vertices, given a view segment $\mathbf{v_0v_1}$, and given a pixel center $\mathbf{p}$, the moment t_e when $\mathbf{p}$ is on **ab**'s projection from $\mathbf{v}$ is found as shown in Eqs. 1.

$$\begin{aligned} (\mathbf{p}-\mathbf{v})n &= 0 \\ n &= (\mathbf{b}-\mathbf{a}) \times (\mathbf{v}-\mathbf{a}) \\ \mathbf{v} &= \mathbf{v_0} + (\mathbf{v_1}-\mathbf{v_0})t_e \\ \mathbf{p} &= \mathbf{p_0} + (\mathbf{v_1}-\mathbf{v_0})t_e \end{aligned} \tag{1}$$

The first two equations place $\mathbf{p}$ on the plane **vab** defined by the edge and the viewpoint. The third equation defines the viewpoint position $\mathbf{v}$ on the view segment when the visibility event occurs, i.e., at t_e. The fourth equation defines the pixel center position at t_e, which has translated from $\mathbf{p_0}$ as much as $\mathbf{v}$ has translated from $\mathbf{v_0}$. Solving for t_e results in a linear equation in t_e, as shown in Eq. 2. If $t_e \notin [0,1]$, edge ab does not generate a visibility event, and p is either on the correct or on the wrong side of **ab** for the entire translation interval $[0,1]$. The correct side of an edge is that given by the third vertex of the triangle, i.e., **c** for edge **ab**. If $t_e \in [0,1]$, edge **ab** does generate a visibility event, and $\mathbf{p}$ is on the correct side of the edge either for $[0,t_e]$ or for $[t_e,1]$.

$$(\mathbf{p_0}-\mathbf{v_0})((\mathbf{b}-\mathbf{a}) \times (\mathbf{v_0} + (\mathbf{v_1}-\mathbf{v_0})t_e - \mathbf{a})) = 0 \tag{2}$$

The visibility event analysis for edge ab produces a translation sub-interval $[t_s,t_f]^{\mathbf{ab}} \subseteq [0,1]$, possibly the empty interval, in which pixel center $\mathbf{p}$ is on the correct side of **ab**. The coverage interval $[t_s,t_f]$ for triangle **abc** is computed by intersecting the translation sub-intervals of the triangle's three edges, i.e., $[t_s,t_f] = [t_s,t_f]^{\mathbf{ab}} \cap [t_s,t_f]^{\mathbf{bc}} \cap [t_s,t_f]^{\mathbf{ca}}$. In addition to needing to know the interval $[t_s,t_f]$ when a triangle covers a pixel center, the visibility algorithm also needs to know the distance $[d_s,d_f]$ to that triangle over the interval in order to enforce correct visibility when multiple triangles cover the pixel center at the same time, much like in conventional z-buffering. Given a value of the translation parameter t, computing the distance d from viewpoint $\mathbf{v} = \mathbf{v_0} + (\mathbf{v_1} - \mathbf{v_0})t$ to triangle **abc** along the ray defined by pixel center $\mathbf{p}$ is trivial, and the complete coverage interval $[(t_s,d_s),(t_f,d_f)]$ is ready to be depth composited with the previously found visibility intervals at p.

(line 9) The newly found coverage interval has to be depth composited with the existing visibility intervals, using conventional interval arithmetic. Consider Fig. 2, right, and let's assume that triangles are processed in the order 1, 2, and then 3. At first, the list of visibility intervals is empty and the entire coverage interval $[t_k,t_l]$ of triangle 1 is kept. The coverage interval $[t_i,t_j]$ of triangle 2 does not overlap with any of the existing visibility intervals, i.e., with $[t_k,t_l]$, so $[t_i,t_j]$ is kept in its entirety as well. The coverage interval of triangle 3 overlaps with that of triangle 2, but 3 is at greater depth than 2, so only the interval $[t_j,t_m]$ is kept. Surfaces modeled with connected triangles yield connected visibility intervals. A coverage interval will not intersect a visibility interval as long as VE triangles do not intersect. If they do, our depth compositing algorithm will correctly detect this second type of visibility event and will create split visibility intervals at the intersection hinge.

Each visibility interval also stores the ID of the triangle that is visible at that interval, i.e., i for T_i, which allows recovering the visible set (line 13). If a coarser granularity visibility solution is desired, the set of visible triangles can be converted to a set of visible objects straightforwardly: an object is visible iff at least one of its triangles is visible. Object-based visibility has the advantage of applying the visibility solution quickly and non-intrusively by simply enabling the visible objects (or disabling the ones that are not visible). The disadvantage is that the total number of triangles rendered by the headset is higher than it should be, as the headset has to render the hidden triangles of partially visible objects. Triangle-based visibility has the advantage of a smaller rendering load, and the disadvantage of having to modify objects to keep only their visible triangles.

3.2 Dynamic Virtual Environments

Many virtual environments of interest to applications are dynamic, with a few or all objects moving. We have extended our continuous visibility computation algorithm to handle dynamic VEs. For each pixel, our algorithm keeps track of which triangle is visible when. For dynamic VEs, computing visibility events becomes more challenging as complex triangle trajectories can result in complex visibility event equations. To make event equation complexity manageable, we decompose the non-linear vertex motion into linear segments. This is already done by numerical simulation codes that save node positions for every one of multiple states. This is also the approach taken to save complex animation with a set of key frames connected smoothly by interpolation. Individual per-vertex piecewise linear translation can indeed approximate any motion, with the error controlled by the number of states. Given a dynamic VE modeled with triangles whose vertices move linearly but independently in a time interval $[0,1]$, we find the triangles seen by a camera as it translates along a view segment with the following modifications to our algorithm from Sec. 3.1.

The visibility event equations for the static VE from Eqs. 1 are extended with the two additional equations shown in Eqs. 3, which model the motions a_0a_1 and b_0b1 of vertices a and b.

$$\begin{aligned} a &= a_0 + (a_1 - a_0)t_e \\ b &= b_0 + (b_1 - b_0)t_e \end{aligned} \tag{3}$$

Solving for t_e now becomes a quadratic equation, as shown in Eq. 4, where t_e now appears in both operands of the cross product.

$$\begin{aligned} (p_0 - v_0)((b_0 + (b_1 - b_0)t_e - a_0 + (a_1 - a_0)t_e)) \times \\ (v_0 + (v_1 - v_0)t_e - a_0 + (a_1 - a_0)t_e)) = 0 \end{aligned} \tag{4}$$

The quadratic visibility event equation can result in 0, 1, or 2 coverage intervals per edge, which can translate to up to four coverage intervals per triangle. This implies that the coverage interval computation in line 8 of Alg. 1 returns an array of coverage intervals, which are then depth composited with the pixel's list of visibility intervals one at the time, in line 9. Depth does not change linearly anymore over a coverage interval, i.e., segments 1, 2, and 3 in Fig. 2, *right*, are now parabola arcs, which makes depth compositing slightly more expensive. Finding the pixels affected by a triangle (lines 3-6 in Alg. 1) does not change.

3.3 Visibility Computation on Spherical Particles

Complex dynamic VEs might include parts modeled with particles. Since our algorithm handles triangles, it also handles particles through tessellation. However, converting particles to triangle meshes leads to a one order of magnitude increase in complexity even for a coarse tessellation that approximates a particle with 10

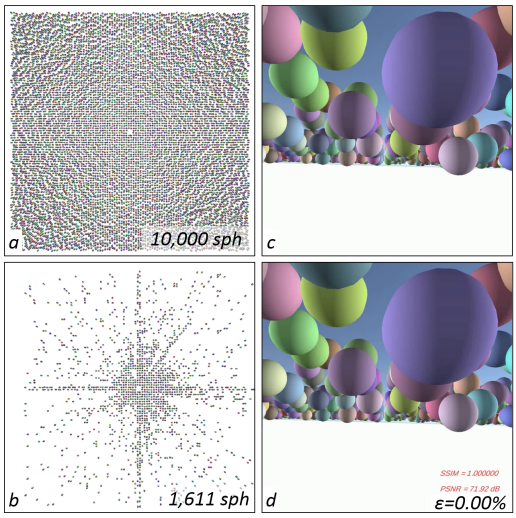

Figure 3: Top view of dataset with 10,000 bouncing spheres and 25 states (*a*), visible set of 1,611 spheres computed directly on spheres, without tessellation, for a view region at the center of the dataset (*b*), ground truth frame rendered from original dataset (*c*), and nearly perfect frame rendered from visible set (*d*).

triangles. For this, we have extended our visibility algorithm to handle spherical particles as a first-order visibility computation primitive. Given a VE modeled with spherical particles, each with its own radius, center, and translation vector, the goal is to find all particles visible as the camera translates on a view segment (Fig. 3).

The first modification of our visibility algorithm (Alg. 1 in Sec. 3.1) is to compute the set of pixels touched by a particle as it translates. We approximate this set conservatively with an axis-aligned bounding box of the extremal projections of the particle. The second concern is to derive the visibility event equation for a spherical particle, as shown in Eqs. 5. o is the center of the sphere as it translates from o_0 to o_1. q is a point on the ray vp, with v and p being defined like before (Eqs. 1). The third equation provides the condition that the ray be perpendicular to the sphere ray through q. The fourth equation places q r away from o, i.e., on the sphere.

$$\begin{aligned} o &= o_0 + (o_1 - o_0)t_e \\ q &= v + (p - v)u \\ (p - v)(q - o) &= 0 \\ (q - o)(q - o) &= r^2 \end{aligned} \tag{5}$$

Plugging the o and q from the first two equations of Eqs. 5, and p and v from the third and fourth equations of Eqs. 1, into the third equation of Eqs. 5 results in Eqs. 6, which shows that u is linear in t_e, with k_1, k_2, and k_3 being scalar constants. Plugging in u from Eqs. 6 into the fourth equation of Eqs. 5 results in a quadratic equation in t_e. Two solutions in $[0,1]$ occur when the ray reaches and leaves the sphere as the viewpoint translates on the view segment. Like for dynamic VEs, depth varies non-linearly along a coverage interval. However, since particles do not intersect, depth compositing of coverage intervals only requires the evaluation of the non-linear depth functions, and not intersecting them.

$$\begin{aligned} u &= t_e(k_1/k_0) + (k_2/k_0) \\ k_0 &= (p_0 - v_0)(p_0 - v_0) \\ k_1 &= (p_0 - v_0)(v_0 - v_1 + o_1 - o_0) \\ k_2 &= (p_0 - v_0)(o_0 - v_0) \end{aligned} \tag{6}$$

Virtual Environment	Illustration	Complexity			Frame rate [fps]	
		Vertices [x10^6]	Triangles [x10^6]	States	Avg	Min
Manhattan	Fig. 5	11.0	3.69	1	42	36
Medieval	Fig. 1 *a-d*	4.35	2.29	1	23	17
Grass	Fig. 1 *e-f*	32.8	54.9	1	-	-
Swaying Grass	video	32.8	54.9	10	-	-
Bouncing Spheres	Fig. 3	10,000 sph.		25	44*	35*

Table 1: VEs used in our experiments. The last column shows the Quest 3 frame rate obtained when rendering the original VE *as-is, without our complexity reduction*. *Grass* cannot be loaded or rendered. **Bouncing Spheres* cannot be loaded at its full complexity, and the frame rate provided is for 20 of the 25 states.

4 Results and Discussion

We have tested our from-segment continuous visibility algorithm on several static and dynamic virtual environments of varying complexity. Our algorithm reduced the VEs to visible sets that can be handled by the headset and that produce high quality frames as compared to ground truth. We provide an overview of our implementation (Sec. 4.1), we describe the VEs used in our experiments (Sec. 4.2), we describe the metrics used to quantify the performance of our algorithm (Sec. 4.3), we provide and discuss the results of our algorithm with respect to ground truth (Sec. 4.4), and we compare our algorithm to the prior art approach of aggregating visibility over multiple viewpoints and timepoints (Sec. 4.5).

4.1 Implementation

Given a rectangular user view region *abcd*, we run our continuous visibility algorithm on segments *ab*, *bc*, *cd*, and *da*, and then union the four visible sets. For each view segment, we find the visible triangles in all view directions, using a cubemap camera, and by running Alg. 1 six times, once for each face of the cubemap. We use a cubemap face resolution of 800×800. The resolution of the cubemap has to be commensurate to the average image footprint of a VE triangle. For datasets with small triangles we run the visibility algorithm multiple times per segment, jittering the orientation of the cubemap. If the first run uses a cubemap aligned to the world coordinate system, each subsequent run will rotate the cubemap around an arbitrary direction. The result is a random sampling of the 2D space of directions, with a resolution controlled by the number of runs. The final visible set is the union of the visible sets found by individual runs.

We have developed a GPU implementation of our visibility algorithm using CUDA [47]. We ran it on a workstation with Intel Xeon E5-2698 v4 2.20GHz 20-core processors, with 512 GB of RAM, and with NVIDIA Tesla V100 SXM2 32GB GPU cards (a single CPU and GPU card were used). The CUDA program proceeds in three phases. Phase 1 counts the number of triangle/pixel pairs (T, p), where the projection of T covers p at least at some point along the segment. This is done with one thread per triangle in thread blocks of size 256. Each thread executes lines 4-7 of Alg. 1 to compute the pixels covered by its triangle. The number of triangle/pixel pairs is used by phase 2 to allocate memory and to store the pairs. Phase 3 computes the list of visibility intervals at each pixel, using one thread per pixel, in thread blocks of size 256. The thread for pixel p executes lines 8 and 9 of Alg. 1. We handle spherical particles (Sec. 3.3) in a CUDA program with a similar structure to the one working with triangles.

4.2 Virtual Environments

The VEs used in our experiments are described in Tab. 1. The VEs are illustrated in the figures listed in column 2, and also in the video accompanying our paper. The first three VEs are static (a single state) and the last two are dynamic (multiple states). *Manhattan* is an actual model of the New York City borough. *Medieval* is a fantasy medieval town. *Grass* is an outdoor VE with grass blades and flowers covering a terrain mesh. *Swaying Grass* has the same geometry as *Grass*, but it is now animated to simulate the grass moving due to wind. *Bouncing spheres* is a simulation of 10,000 spheres that bounce on the ground plane from random heights, around the user, losing energy, and coming to a stop after 25 simulation states. The sphere radii are 4 m and the dataset covers a 100 m $\times$ 100 m ground plane squares. A Quest 3 all-in-one VR headset [1] cannot load or render *Grass*, and cannot load the 25 states of *Bouncing Spheres*. The headset struggles to render *Medieval*, achieving an average frame rate of a third of the 72 fps Quest 3 display frame rate. Although *Manhattan* is a low density VE that can be handled by a Quest 3, it is still important to avoid rendering unnecessary triangles to save battery and to make room for adding complex geometries in the proximity of the user, with the city model as a backdrop.

4.3 Performance Metrics

Our visibility algorithm is called upon to reduce the complexity of a given VE for a given user region such that the headset can render quality frames at interactive rates.

Frame quality. Our algorithm finds all triangles visible at pixel centers as the camera translates along a view segment. This means that rendering the visible set with the camera at an intermediate viewpoint *on the view segment* always results in the correct frame, i.e., the frame that would be obtained by rendering the original VE. However, we run our visibility algorithm on the *perimeter* of a rectangular user view region, and then use the visible set to render frames from viewpoints *inside* the view rectangle. Our from-perimeter visible set is an approximation, i.e., a subset of the complete from-rectangle visible set. Given a frame F^* rendered from our approximate visible set VS^*, if the triangle visible at pixel P is missing from VS^*, P will be rendered incorrectly. We compare our frame F^* to the ground truth frame F obtained by rendering the original VE from the same view.

We quantify frame quality with the following three standard image comparison metrics: (1) image error ε, peak signal to noise ratio *PSNR* [19], and (3) structural similarity index measure *SSIM* [46]. ε is defined as the percentage of pixels in F^* that are different from their counterpart in F. A pixel P^* in F^* is different from its counterpart P in F if the final depth at P^* is different (farther) from the final depth at P. The different depth indicates that the visible triangle at P was missed by the approximate set. This definition of image error is more discerning than one based on color that is VE dependent and discounts errors in regions of the VE with uniform color. *PSNR* is defined as the mean squared error over the three color channels, it is measured on the logarithmic decibel scale, and lossy image processing results are considered good in the 30 to 50 dB range [20]. *SSIM* measures the similarity between two images, and, when one is ground truth, to quantify the perceived quality of the other image, with good values above 0.97 [20].

Frame rate. The visible set has to be sufficiently compact for the headset to render it with a high frame rate. The display frame rate for the Quest 3 is 72 fps. We report the average and minimum frame rates over sequences of thousands of frames.

Visibility computation time. Visibility computation is not in the "inner loop" of the VR application–our algorithm is run on a workstation and the visible set is communicated to the headset. One option is to precompute visible sets for a tiling of the VE and to serve the precomputed visible sets to the application at startup and as the user navigates the viewpoint through the VE. Another option, which is tractable when visibility computation times are reasonable, is to compute the desired visible set on the fly, at VR session startup, and as the user teleports from one VE region to another.

Virtual Environment	Visibility computation					Headset trace			Frame quality						Frame rate [fps]	
	View rect. [m]	Time [s]	Granularity	Visible set triangles		Frames	Distance [m]	Rotation [x360°]	ε [%]		PSNR		SSIM			
				Abs. [x10⁶]	Rel. [%]				Avg	Max	Avg	Min	Avg	Min	Avg	Min
Manhattan	10 x 489	28	tri.	0.034	0.9	3,000	485.2	15.8	0.02	0.45	173	39.2	0.99999	0.99999	71	70
Medieval	A 10 x 165	107	tri.	0.205	8.9	1,500	166.3	16.0	0.02	0.56	109	26.8	0.99999	0.99978	69	57
			obj.	0.813	35				0.0	0.0	255	255	1	1	71	70
	B 23 x 17	22	tri.	0.086	3.8	1,500	74.8	12.9	0.12	2.26	115	39.9	0.99999	0.99997	71	71
			obj.	0.523	23				0.04	0.27	210	37.7	0.99999	0.99998	72	71
Grass	A 2 x 2	5.6	tri.	4.15	7.6	2,250	18.7	15.5	0.19	0.63	38.9	26.1	0.99998	0.99991	39	31
	B 2 x 2	5.4	tri.	1.27	2.3	2,250	15.0	11.7	0.13	0.36	53.6	29.8	0.99999	0.99997	72	70
Swaying Grass	A 2 x 2	47	tri.	5.44	9.9	2,250	18.7	15.5	0.12	0.32	75.8	29.3	0.99999	0.99996	28	23
	B 2 x 2	44	Tri.	1.96	3.6	2,250	15.0	11.7	0.07	0.23	81.9	31.4	0.99999	0.99997	68	61
Bouncing S.	20 x 20	1.3	sph.	1,611	16	1,000	11.3	6.3	0.00	0.06	121	43.4	0.99999	0.99999	71	70

Table 2: VE complexity reduction with our visibility algorithm, and frame quality and frame rate achieved.

4.4 Results. Comparison to Ground Truth.

Tab. 2 gives our results. Each row describes an experiment where a visible set was computed from a view rectangle ("visibility computation" group of columns), the visible set was used to render frames along a path captured with the headset ("Headset trace"), the frames were compared to ground truth frames ("Frame quality"), and the visible set was used to render frames freely, on the headset, in standalone mode, from within the view rectangle, measuring the frame rate. The *Manhattan* view rectangle follows Fifth Avenue for almost 500 m (Fig. 4 *a-b*). The *Medieval* view rectangles map to a street (A) (Fig. 1 *a-b*) and to a plaza (B). The *Grass* view rectangles are in the center (A) and at the edge (B) of the VE. The *Bouncing Spheres* view rectangle in the center of the dataset (Fig. 3, *a, b*).

Visibility computation running time. The asymptotic complexity of Alg. 1 is $O(nmk\log k)$, where n is the number of triangles in the VE over which the algorithm iterates (line 3), m is the average number of pixels of the convex hull of the two extremal projections of a triangle over which the algorithm iterates (line 7), and k is the average number of visibility intervals per pixel (line 9). m depends on the length of the view segment–the longer the segment, the more the triangle moves in the image between the two viewpoints, and the bigger the convex hull. The list of intervals is updated using a balanced binary tree, hence the $k\log k$ time of the depth compositing step (line 9). All other steps of the algorithm are constant time, including the computation of the convex hull of six 2D points (line 6), and of the triangle coverage interval (line 8).

The longer the view rectangle, the larger m, as a triangle's trace extends over a bigger portion of the image, and the larger k, as more triangles pass over a pixel as the viewpoint translates. Consequently, *Medieval A* takes 107 s versus the 5.6 s for *Grass*, even though *Grass* has over twenty times more triangles than *Medieval* (Tab. 1). Fig. 5 shows that for *Manhattan* visibility computation time grows linearly as the view rectangle is scaled up to the 489 m size; for *Medieval A* however, the dependence is super-linear and visibility could be computed more efficiently by splitting the view rectangle into smaller pieces. For *Swaying Grass* visibility is computed for each of the 10 states, with a running time linear in the number of states. Computing visibility for *Bouncing Spheres* takes 1.7 s total, over all 25 states. Scaling up the simulation from 100 × 100 to 1,000 × 1,000 increases the computation time to just 3.2 s, finding 137,763 visible spheres.

Complexity reduction and frame rate. Our visibility algorithm brings a substantial reduction in VE complexity. The visible set of *Manhattan* has only 34,000 triangles, which is 0.9% of the original VE. Solving visibility with triangle granularity yields more compact visible sets than doing so with object granularity, e.g., 8.9% vs 35% for *Medieval A*. In all cases the frame rate is close to the 72 fps display frame rate of the Quest 3, except for *Grass A* where the visible set has over 4 Mtris. Our approach makes *Swaying Grass*, a 54 Mtris animated VE tractable on an all-in-one VR headset, a VE that is unusable at its original complexity. For *Bouncing Spheres*, the 1,611 visible spheres are easily handled by the headset. For visualization, we tessellate a sphere with 15 parallels and 15 meridians, resulting in 450 triangles per sphere, or 725 Ktris for the 1,611

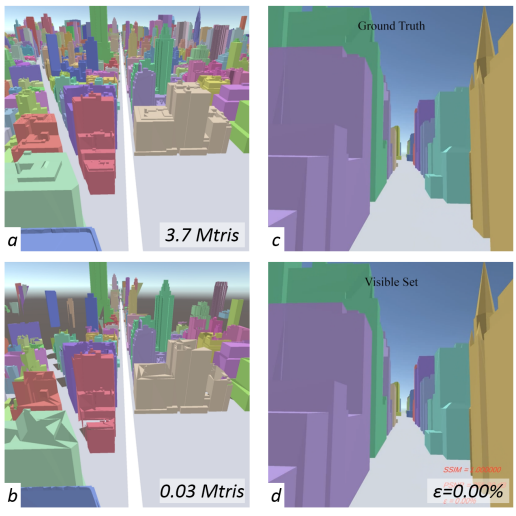

Figure 4: The *Manhattan* VE (*a*) is reduced to the 0.9% of its triangles (*b*) visible from a 489 m stretch of 5th Av. (white rectangle). The visible set produces nearly perfect frames (*c* vs. *d*).

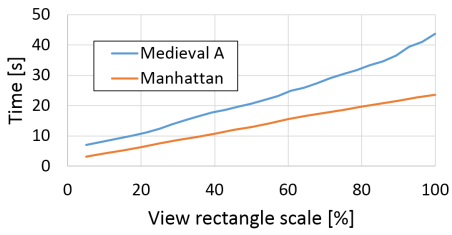

Figure 5: Visibility computation time vs. view rectangle size.

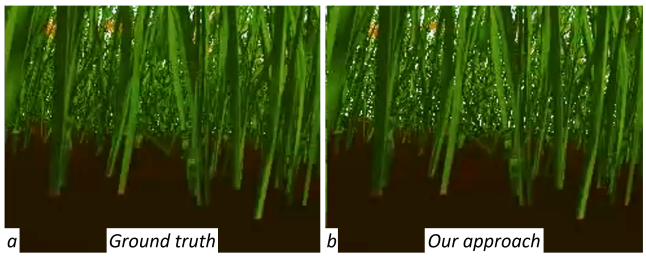

Figure 6: Magnified fragment of frames *e-f* from Fig. 1 that shows the grass is slightly less dense at a distance for our approach.

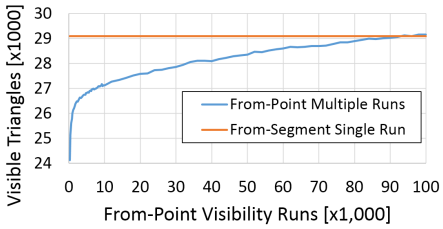

Figure 7: Comparison of a single run of our from-segment algorithm to multiple runs of conventional from-point visibility, in terms of visible triangles found. Close to one 100,000 from-point runs are needed to complete the visible set, which from-segment finds in a single run.

spheres in the visible set, per state. This shows that computing visibility directly on particles is essential, bringing a two orders of magnitude reduction in problem scope. For the scaled-up version of the simulation with 10^6 spheres, the visible spheres yield 62 Mtris per state, and making *rendering* tractable requires geometric simplification or direct sphere rasterization in a custom fragment shader.

Frame quality. The visible sets produce frames that are hard to distinguish from ground truth frames rendered from the original VEs (see Tab. 2, Figs. 1, 3, and 4, and accompanying video). Over all experiments, the average image error ε is at most 0.19%, the average *PSNR* is at least 38.9, and the average *SSIM* is at least 0.99998. The largest ε is 2.26%, the smallest *PSNR* is 26.1, and the smallest *SSIM* is 0.9996. The largest errors are encountered in *Medieval* when the user viewpoint is above the view rectangle, revealing triangles not part of the visible set (e.g., left side of tree bed in Fig. 1 *d*), and in *Grass*, where some of the distant grass blades are missing (Fig. 6). As expected, quality is higher for object granularity than for triangle granularity, and is flawless for *Medieval A*. Quality is higher for *Swaying Grass* than for *Grass*, as the near grass blades move out of the way to reveal grass blades that would otherwise be missed in the static VE.

For many of our experiments the quality of the frames might be unnecessarily high. For example for *Swaying Grass* the average *PSNR* is 75.8 dB and 81.9 dB, and the application might choose to reduce the visible set to save resources while still achieving great frame quality. A simple way to do so is to tune the resolution of the cubemap used in the visibility computation to achieve a desired *PSNR* (e.g., 40 dB), which will result in a smaller visible set, much the same way a video application might choose the most aggressive compression setting that achieves a desired quality level.

4.5 Comparison to From-Point Sample-Based Visibility

We compute visibility continuously along a view segment, without having to *guess* where along the segment new triangles become visible. Instead, our algorithm *computes* these visibility events analytically. The traditional approach to visibility is to sample the given user view region with multiple viewpoints, to compute visibility at each view point, and to aggregate the from-point visible sets to approximate the from-region visible set. Such sample-based visibility is inefficient because visibility events are hard to predict, and because the visible sets of neighboring points are highly redundant. We have conducted two experiments, one that compares multiple runs of from-point visibility to a single run of our from-segment algorithm (Sec. 4.5.1), and one to multiple runs (Sec. 4.5.2).

4.5.1 Single from-segment vs. multiple from-point

We compare our from-segment algorithm to multiple runs of from-point visibility that sample the view segment uniformly. The results are shown in Fig. 7, where *Medieval* was used. From-point visibility converges slowly. Sampling the view segment with 10,000 view points still only finds about 27,000 of the 29,000 visible triangles. Whereas our from-segment algorithm finds the 29,000 visible triangles in a single pass, close to 100,000 from-point runs are needed to complete the visible set. We note that the number of visible triangles found does not strictly increase with the number of even steps along the view segment, i.e., with the number of runs in the graph in Fig. 7. A finer subdivision of the view segment changes the viewpoints along the view segment, and, although more numerous, the new viewpoints might find fewer triangles, which is another testament to the chaotic nature of visibility.

A from-point visibility run entails a conventional rendering pass over the VE with a trivial fragment shader that outputs the triangle ID, followed by a collection of the visible triangles with a simple pass over the output image. As such, a from-point visibility run implies a smaller computational cost than a run of our from-segment algorithm. We have compared the running time of our from-segment visibility algorithm to multiple runs of from-point visibility. As expected, the total from-point visibility time is linear in the number of runs. An equal-quality comparison reveals that from-point needs substantially longer than from-segment to complete the visible set, i.e., 30 min vs. 3.84 s. Furthermore, 200 runs of from-point take 4.1 s, so an equal-time comparison reveals that from-point finds only 24,000/29,000 = 83% of the visible triangles in the 3.84 s from-segment needs to find all visible triangles.

4.5.2 Multiple from-segment vs. multiple from-point

A single run of our from-segment continuous visibility computation algorithm is not always sufficient. For a highly detailed VE, visible triangles with a small footprint can land in between the centers of the pixels of the cubemap. Furthermore, we are approximating the triangles visible from inside the view rectangle by computing the triangles visible from the view rectangle perimeter. The user of a VR application is of course free to assume any view direction on the view rectangle perimeter, as well as any viewpoint inside inside the view rectangle, and even slightly above or below it, as the user's head also translates up and down as they walk. This means that from the point of view of a VR application, our from-segment visibility algorithm produces an *approximation* of the visible set needed for a user path. As shown in Sec. 4.4, it is a high quality approximation. In this section we show that the quality of the approximate visible set can be conveniently controlled by running our algorithm multiple times, each time with a different random orientation of the cubemap along the view rectangle perimeter.

Fig. 8 shows the number of visible triangles found by multiple from-segment and multiple from-point runs, for the view rectangle B of *Grass* (see Fig. 2). We computed visibility with our algorithm 1,000 times on the perimeter of the view rectangle, which corresponds to 4,000 runs of the from-segment algorithm. We computed visibility with the conventional from-point algorithm from random viewpoints inside the view rectangle, with a cubemap per viewpoint to cover all directions, and with random cubemap orientations. Our algorithm finds consistently many more visible triangles than from-

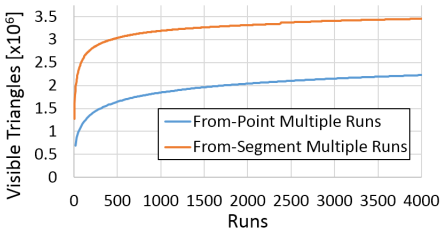

Figure 8: Comparison between our from-segment algorithm and conventional from-point visibility, in terms of visible triangles found as a function of the number of runs. From-segment consistently finds more triangles than from-point for the same number of runs.

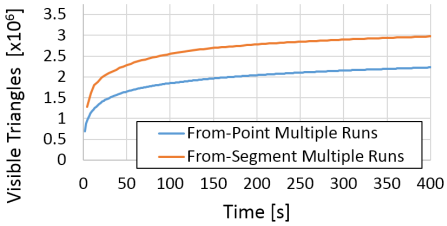

Figure 9: Comparison between our from-segment algorithm and conventional from-point visibility, in terms of visible triangles found as a function of time. From-segment consistently finds more triangles than from-point for the same amount of time.

point for the same number of runs. For example, after 500 runs, we find 3 Mtris while from-point only 1.6 Mtris; after 4,000 runs, we find 3.5 million and from-point only 2.23 million.

The heuristic of using the view rectangle sides for the multiple runs of our algorithm is based on the fact that the perimeter provides the extremal positions of the viewpoint. We have investigated this heuristic by computing visibility along segments connecting opposite sides of the view rectangle. Although each segment is longer than the sides of the rectangle, these random segments crisscrossing the rectangle found fewer triangles for the same number of 4,000 runs, i.e., 3.3 Mtris vs. the 3.5 Mtris found from the perimeter.

The running time of the from-segment and of the from-point approaches is linear with the number of runs, as expected. A run of from-segment takes 1.05 s (per segment, per cubemap face). A run of from-point takes 0.101 s. Fig. 9 shows the number of triangles found by each of the two methods as a function of time. The faster running time is not sufficient for from-point to find more triangles in the same amount of time. We conclude that, although from-segment does not find all triangles visible from a given view rectangle in a single run, from-segment is a more powerful visibility computation primitive than from-point, i.e., multiple from-segment runs find visible triangles more efficiently than multiple from-point.

5 Conclusions. Limitations. Future Work

We have described a visibility algorithm for computing the set of triangles visible at the pixels of a camera that translates along a view segment. We have used this algorithm to reduce the complexity of VEs, making them tractable on an all-in-one VR headset such as the Quest 3. The algorithm supports computing visibility with triangle, sphere, or object granularity, and it supports dynamic VEs, where potentially every vertex moves every frame. The visible sets provide high quality frames and high frame rates.

We use visibility to reduce the complexity of a VE. Therefore, like all visibility based approaches, we rely on the assumption that the visible set constitutes a manageable fraction of the original VE, which can be handled by the headset. Another limitation of visibility that we inherit is that the output frame can be affected not only by parts of the VE to which the user has line of sight, but also by parts not visible from the current user position. For example, such hidden geometry can cast shadows or be reflected in the output frame, so it should be included in the visible set. Future work could investigate the solution of computing the shadow or reflection environment map from the entire geometry, on a workstation, and then to use it for correct shadows and reflections when rendering the visible set on the headset.

Given a 3D view region, there is no exact visibility computation algorithm with a practical implementation. Our algorithm allows sampling the 3D view region with view segments, which we have shown is more efficient than sampling it using view points. Repeated runs of our from-segment visibility algorithm provide a good and controllable approximation of the from 3D region visible set. This makes complex VEs tractable on all-in-one headsets, at the cost of small visual artifacts. The alternative is to resort to drastic simplifications of the VE, such as replacing grass geometry with grass texture, or replacing tree canopy geometry with blobs. Furthermore, even if exact from 3D region visibility were possible, the visible set might contain too many triangles, and the application will not just accept but actually demand a way of reducing the visible set. Multiple runs of our algorithm find visible triangles in decreasing order of importance, as triangles visible for longer over larger image regions are likely to be found sooner, which allows the application to best spend its rendering budget.

VEs with animated geometry are challenging for visibility computation. For many VEs, the dynamic geometry constitutes a small fraction of the total VE geometry, and the conventional approach is to compute visibility without the dynamic geometry, adding it to the resulting visible set. For example, a door the user has the option to open in an indoor VE, or any other object with which the user can interact, is excluded from visibility computation and then added to the visible set. The approach is, of course, compatible with our work. Furthermore, our work makes the contribution of enabling visibility computation in a fully dynamic environment, where every vertex moves at every frame, as demonstrated for *Swaying Grass* and for *Bouncing Spheres*. VEs with dynamic lights require including in the visible set not only the triangles visible to the user, but also the triangles visible to the lights, as needed to compute accurate shadow maps. When the dynamic light trajectory is contained in the user view region, such is the case, for example, for a flashlight handheld by the user or the headlights of a car moving on the street where the user is located, the visible set computed for the user subsumes the light visible set, and no additional work is required. When the lights move in a region disjoint from that of the user, such as the headlights of a car moving on a side street, our visibility algorithm has to be run for the light trajectories to enhance the visible set with triangles not visible to the user.

Our approach is ready to be integrated in distributed VR systems, with a server providing visible sets to individual or clusters of thin VR clients. Our approach allows for large user view cells, which supports the coarse transfer granularity needed to smooth over network performance fluctuations and to prefetch the visible sets of neighboring cells before the user enters them. Our work is inscribed in the longer term effort of standing up online libraries of VR applications that employ full complexity VEs, which users with thin VR clients can browse and run without lengthy download times and without trivializing the VE, as needed for applications beyond entertainment, such as applications in education, healthcare, science, and engineering.

Acknowledgments

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

REFERENCES

[1] Quest 3. *Meta Platforms, Inc. https://www.meta.com/quest/quest-3/*. 2, 6

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

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

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

[5] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Commun. ACM*, 19(10):542–547, oct 1976. doi: 10.1145/360349.360353 2

[6] S. Charneau, L. Aveneau, and L. Fuchs. Exact, robust and efficient full visibility computation in plücker space. *The Visual Computer*, 23(9-11):773–782, 2007. 3

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

[8] X. Décoret, G. Debunne, and F. Sillion. Erosion based visibility preprocessing. In *Proceedings of the 14th Eurographics workshop on Rendering*, pp. 281–288. Eurographics Association, 2003. 3

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

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

[11] F. Durand, G. Drettakis, J. Thollot, and C. Puech. Conservative visibility preprocessing using extended projections. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 239–248. ACM Press/Addison-Wesley Publishing Co., 2000. 3

[12] T. Feng, H. Sun, Q. Qi, J. Wang, and J. Liao. Vabis: Video adaptation bitrate system for time-critical live streaming. *IEEE Transactions on Multimedia*, 22(11):2963–2976, 2019. 2

[13] L. Fink, N. Hensel, D. Markov-Vetter, C. Weber, O. Staadt, and M. Stamminger. Hybrid mono-stereo rendering in virtual reality. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pp. 88–96, 2019. doi: 10.1109/VR.2019.8798283 2

[14] B. Han, Y. Liu, and F. Qian. Vivo: Visibility-aware mobile volumetric video streaming. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, MobiCom '20. Association for Computing Machinery, New York, NY, USA, 2020. doi: 10.1145/3372224.3380888 2

[15] D. Haumont, O. Mäkinen, and S. Nirenstein. A low dimensional framework for exact polygon-to-polygon occlusion queries. In *Proceedings of the Sixteenth Eurographics conference on Rendering Techniques*, pp. 211–222. Eurographics Association, 2005. 3

[16] J. He, M. A. Qureshi, L. Qiu, J. Li, F. Li, and L. Han. Rubiks: Practical 360-degree streaming for smartphones. In *Proceedings of MobiSys*, 2018. 2

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

[18] J. Hladky, H.-P. Seidel, and M. Steinberger. The camera offset space: Real-time potentially visible set computations for streaming rendering. *ACM Transactions on Graphics (TOG)*, 38(6):1–14, 2019. 3

[19] A. Hore and D. Ziou. Image quality metrics: Psnr vs. ssim. In *2010 20th international conference on pattern recognition*, pp. 2366–2369. IEEE, 2010. 2, 6

[20] A. Hore and D. Ziou. Image quality metrics: Psnr vs. ssim. In *2010 20th international conference on pattern recognition*, pp. 2366–2369. IEEE, 2010. 6

[21] T. Kämäräinen, M. Siekkinen, J. Eerikäinen, and A. Ylä-Jääski. Cloudvr: Cloud accelerated interactive mobile virtual reality. In *Proceedings of the 26th ACM international conference on Multimedia*, pp. 1181–1189, 2018. 2

[22] V. Kelkkanen, M. Fiedler, and D. Lindero. Synchronous remote rendering for vr. *International Journal of Computer Games Technology*, 2021:1–16, 2021. 2

[23] J. Kim, P. Knowles, J. Spjut, B. Boudaoud, and M. Mcguire. Post-render warp with late input sampling improves aiming under high latency conditions. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3(2):1–18, 2020. 2

[24] T. Koch and M. Wimmer. Guided visibility sampling++. *Proc. ACM Comput. Graph. Interact. Tech.*, 4(1), apr 2021. doi: 10.1145/3451266 3

[25] B. Koniaris, M. Kosek, D. Sinclair, and K. Mitchell. Real-time rendering with compressed animated light fields. In *Proceedings of Graphics Interface 2017*, GI 2017, pp. 33 – 40. Canadian Human-Computer Communications Society / Société canadienne du dialogue humain-machine, 2017. doi: 10.20380/GI2017.05 3

[26] Z. Lai, Y. C. Hu, Y. Cui, L. Sun, and N. Dai. Furion: Engineering high-quality immersive virtual reality on today's mobile devices. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, p. 409–421. Association for Computing Machinery, New York, NY, USA, 2017. doi: 10.1145/3117811.3117815 3

[27] K. Lee, J. Yi, Y. Lee, S. Choi, and Y. Kim. GROOT: A Real-time Streaming System of High-Fidelity Volumetric Videos. In *Proc. ACM MobiCom*, Sept. 2020. 2

[28] T. Liu, S. He, S. Huang, D. Tsang, L. Tang, J. Mars, and W. Wang. A benchmarking framework for interactive 3d applications in the cloud. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 881–894. IEEE, 2020. 2

[29] D. Luebke, B. Watson, J. D. Cohen, M. Reddy, and A. Varshney. *Level of Detail for 3D Graphics*. Elsevier Science Inc., USA, 2002. 2

[30] N. Max and K. Ohsaki. Rendering trees from precomputed z-buffer views. In *Rendering Techniques' 95*, pp. 74–81. Springer, 1995. 3

[31] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pp. 39–46. ACM, 1995. 3

[32] A. Mehrabi, M. Siekkinen, T. Kämäräinen, and A. yÏ J¨ślki. Multi-tier cloudvr: Leveraging edge computing in remote rendered virtual reality. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 17(2):1–24, 2021. 3

[33] F. Mora and L. Aveneau. Fast and exact direct illumination. In *Computer Graphics International 2005*, pp. 191–197. IEEE, 2005. 3

[34] Z. Nadir, T. Taleb, H. Flinck, O. Bouachir, and M. Bagaa. Immersive services over 5g and beyond mobile systems. *IEEE Network*, 35(6):299–306, 2021. 2

[35] S. Nirenstein, E. Blake, and J. Gain. Exact from-region visibility culling. In *Proceedings of the 13th Eurographics Workshop on Rendering*, EGRW '02, p. 191–202. Eurographics Association, Goslar, DEU, 2002. 3

[36] S. Nirenstein and E. H. Blake. Hardware accelerated visibility preprocessing using adaptive sampling. *Rendering Techniques*, 2004:15th, 2004. 3

[37] J. Park, I.-B. Jeon, S.-E. Yoon, and W. Woo. Instant panoramic texture mapping with semantic object matching for large-scale urban scene reproduction. *IEEE Transactions on Visualization and Computer Graphics*, 27(5):2746–2756, 2021. doi: 10.1109/TVCG.2021.3067768 3

[38] V. Popescu, S. H. Lee, A. S. Choi, and S. Fahmy. Complex virtual environments on thin vr systems through continuous near-far partitioning. In *2022 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pp. 35–43. IEEE, 2022. 2

[39] V. Popescu, E. Sacks, J. Cui, and R. Ashok. Efficient and robust from-point visibility. *IEEE Transactions on Visualization and Computer Graphics*, 30(8):5313–5327, 2024. doi: 10.1109/TVCG.2023.3291138 3

[40] F. Qian, B. Han, J. Pair, and V. Gopalakrishnan. Toward practical volumetric video streaming on commodity smartphones. In *Proc. of ACM HotMobile*, 2019. 2

[41] F. Qian, B. Han, Q. Xiao, and V. Gopalakrishnan. Flare: Practical viewport-adaptive 360-degree video streaming for mobile devices. In *Proceedings of MOBICOM*, 2018. 2

[42] A. Schollmeyer, S. Schneegans, S. Beck, A. Steed, and B. Froehlich.

Efficient hybrid image warping for high frame-rate stereoscopic rendering. *IEEE Transactions on Visualization and Computer Graphics*, 23(4):1332–1341, 2017. doi: 10.1109/TVCG.2017.2657078 2

[43] J. Shade, S. Gortler, L.-w. He, and R. Szeliski. Layered depth images. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pp. 231–242. ACM, 1998. 3

[44] M. Stengel, Z. Majercik, B. Boudaoud, and M. McGuire. A distributed, decoupled system for losslessly streaming dynamic light probes to thin clients. In *Proceedings of the 12th ACM Multimedia Systems Conference*, pp. 159–172, 2021. 2

[45] P. Stotko, S. Krumpen, M. B. Hullin, M. Weinmann, and R. Klein. Slamcast: Large-scale, real-time 3d reconstruction and streaming for immersive multi-client live telepresence. *CoRR*, abs/1805.03709, 2018. 3

[46] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004. 2, 6

[47] N. Wilt. *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013. 6

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

[49] A. Zhang, C. Wang, B. Han, and F. Qian. YuZu: Neural-Enhanced volumetric video streaming. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 137–154. USENIX Association, Renton, WA, Apr. 2022. 2

[50] H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff III. Visibility culling using hierarchical occlusion maps. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 77–88. ACM Press/Addison-Wesley Publishing Co., 1997. 3

[51] S. Zhao, H. Abou-zeid, R. Atawia, Y. S. K. Manjunath, A. B. Sediq, and X.-P. Zhang. Virtual reality gaming on the cloud: A reality check. In *2021 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6. IEEE, 2021. 2

[52] P. Zhou, Y. Xie, B. Niu, L. Pu, Z. Xu, H. Jiang, and H. Huang. Qoe-aware 3d video streaming via deep reinforcement learning in software defined networking enabled mobile edge computing. *IEEE Transactions on Network Science and Engineering*, 8(1):419–433, 2020. 3

[53] Y. Zhou and V. Popescu. Clovr: Fast-startup low-latency cloud vr. *IEEE Transactions on Visualization and Computer Graphics*, 30(5):2337–2346, 2024. doi: 10.1109/TVCG.2024.3372059 2

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