Hierarchical Visibility for Virtual Reality

WARREN HUNT, Oculus Research
MICHAEL MARA, Stanford University and Oculus Research
ALEX NANKERVIS, Oculus Research

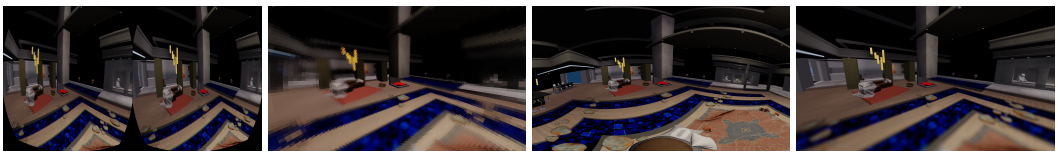

Fig. 1. Images of the Home scene generated with various non-standard sampling patterns that our raycaster can handle natively. From left to right: rendering directly into post-distortion space for the Oculus Rift CV1, rendering a sparse foveated sampling pattern, rendering an extermely wide field of view (210 x 130 degrees), and rendering stochastic depth-of-field.

We introduce a novel primary visibility algorithm based on ray casting that provides real time performance and a feature set well suited for rendering virtual reality. The flexibility provided by our approach allows for a variety of features such as lens distortion, sub-pixel rendering, very wide field of view, foveation and stochastic depth of field blur to be implemented and composed naturally while maintaining real time performance. In contrast, the current rasterization pipelines implemented in hardware require multiple passes and/or post processing to approximate these features and current highly optimized ray tracers, which primarily focus on Monte Carlo path tracing, do not achieve real time performance on current VR displays (1080x1200x2@90hz). Our approach uses a bounding volume hierarchy acceleration and a two level frustum culling/entry point search algorithm to optimize the traversal of coherent primary visibility rays. We introduce an adaptation of MSAA for raycasting that significantly lowers memory bandwidth, we leverage an AVX optimized CPU traversal to perform the majority of culling and an optimized CUDA GPU implementation for triangle intersection, multi-sample antialiasing, and shading. The implementation provides support for animation and physically-based shading and lighting. We believe this approach presents a concrete, viable alternative to rasterization that is significantly better suited to rendering for virtual and augmented reality. In order to engage the community, we have released our implementation under an open-source license.

CCS Concepts: • **Computing methodologies** → **Ray tracing**; **Virtual reality**; *Visibility*; *Graphics processors*;

Additional Key Words and Phrases: Ray Casting, Ray Tracing, Visibility, GPU, Virtual Reality

ACM Reference Format:
Warren Hunt, Michael Mara, and Alex Nankervis. 2018. Hierarchical Visibility for Virtual Reality. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 1, Article 8 (May 2018), 18 pages. https://doi.org/10.1145/3203191

Authors' addresses: Warren Hunt, Oculus Research, Redmond, Washington, Warren.Hunt@oculus.com; Michael Mara, Stanford University, Stanford, California, Oculus Research, Menlo Park, California, mmara@cs.stanford.edu; Alex Nankervis, Oculus Research, Redmond, Washington, Alex.Nankervis@Oculus.com.

2577-6193/2018/5-ART8
https://doi.org/10.1145/3203191

1 INTRODUCTION

One of the most fundamental problems in computer graphics is determination of visibility. At present, the two most commonly used approaches are ray tracing [Whitted 1980], which simulates light transport and is dominant in industries where accuracy is valued over speed such as movies and CAD, and z-buffering, which is implemented by current graphics hardware and is de-facto standard in real-time graphics. We present a visibility algorithm that has performance characteristics close to that of z-buffering, but with additional flexibility that enables a wide variety of effects required for rendering virtual reality. The system presented represents a snapshot of an ongoing research platform. We describe the fundamentals of the system along with some applications, use cases, and a performance evaluation. We also provide the implementation as open source[1].

We gain many of the performance advantages of z-buffering while retaining some of the necessary additional flexibility of ray tracing by leveraging data structures and insights from the real-time ray tracing literature, but restricting ourselves to only solving primary visibility (avoiding incoherent secondary effects such as reflection and refraction). This restriction means the algorithm is a version of *ray casting* rather than full ray tracing. Early real-time "ray tracing" systems such as MLRT [Reshetov et al. 2005] were actually ray casters, incapable of efficiently producing incoherent effects, and ours follows in this mold. Work on ray casting has been relatively rare in the prior decade due to a community focus on Monte Carlo ray tracing and the fact that, before the recent repopularization of virtual reality, it didn't provide a compelling enhanced feature set over rasterization.

Our contributions are:

- A breakdown of VR-relevant use cases that are not well-supported by traditional hardware rasterizers (Section 2).
- A straightforward generalization of multi-sample anti-aliasing (MSAA) for ray casting (Section 5.1).
- The HVVR system, which combines previous work, general best practices, and the generalization of MSAA to achieve ray rates in excess of 10 billion rays per second for nontrivial scenes on a modern computer, naturally supporting all of the VR-relevent use cases.

2 APPLICATIONS

The flexibility of our ray casting approach leads it to naturally implement a variety of rendering features relevant to virtual reality. In this section we describe several features we've implemented within the system and describe their applicability to rendering for virtual reality. It's also worth emphasizing that because we are using ray casting, these features also *compose* naturally and are in no way mutually exclusive.

2.1 Direct Distortion + Subpixel Rendering

One of the primary differences between head mounted and traditional displays is the required use of viewing optics. In addition to allowing a user to focus on the display, the viewing optics add a variety of aberrations to the display as viewed. Notably they usually produce a pin-cushion distortion with chromatic dependency, which causes both color separation and non-uniform pixel spacing. This leads to the user effectively seeing three different displays, one for each color, with three different distortion functions. While traditionally these artifacts are corrected during a post-processing image distortion phase, a ray caster can render the distorted image directly.

There are a variety of advantages and one notable disadvantage to the direct rendering approach. The disadvantage is that rays must be independently cast for each subpixel, rather than each pixel.

[1] https://github.com/facebookresearch/HVVR

This is due to the fact that the viewing optics cause suppixels to separate. This increases the sample rate by the ratio of subpixels to pixels (2x on the Oculus Rift and HTC Vive).

The advantages of direct subpixel rendering include an increase in clarity for the same reasons that ClearType improves text clarity: by taking into account the spatial placement of the subpixels, a more accurate intensity can be defined for each subpixel, rather than for the whole pixel simultaneously. Although we do not provide a study of the impact of this improvement in this paper (the details are outside of the intended scope), the improvement appears significant.

Additionally, direct subpixel rendering obviates the need for large pre-distortion render targets. For example, the Vive has an eye-buffer resolution of 3024 x 1680, or 1.96 times that of the output display resolution. Given the image presented to the display is cropped by a barrel distortion mask, only about 85% of the 2160 x 1200 pixels are actually presented. Because of these large guard-bands/over rendering of the eye buffer, fewer than 50% of the total pixels need to be rendered. Notably, this compensates for the 2x subpixel additional shading. Direct subpixel rendering improves image quality further by avoiding the distortion correction resampling pass, which introduces both aliasing and blur, most noticeably for near-Nyquist detail such as text.

Finally, by removing the full-frame distortion correction pass, direct distortion rendering enables the opportunity to render *just in time*. Because blocks can be rendered independently and on demand, a ray caster can render directly to the front buffer, immediately prior to scan-out. This technique is known generally as "beam racing" and has the potential to significantly reduce the head-movement to photon latency. Unfortunately front buffer rendering requires specialized driver extensions and is not implementable in a general manner on current graphics hardware.

2.2 Wide Field of View

Another noticeable difference between head mounted and traditional displays is field of view (FoV). While a typical desktop display consumes about 30° of the viewer's FoV, current VR headsets start around 90° and can go as high as 210° for the Starbreeze StarVR.

Because of the uniform sample grid assumption made in the z-buffer algorithm, the number of samples required to render an image goes up with the tangent of the half angle of the FoV (assuming a fixed minimum angular resolution). This leads to a precipitous loss in efficiency towards 180° FoV. For example, the difference between 140° and 160° is almost 4 times the number of rendered pixels.

In order to mitigate this inefficiency, the approach of rendering multiple narrow FoV images and stitching them together has been studied [Toth et al. 2016]. Although this improves the efficiency of the z-buffer algorithm, it comes at the cost of repeatedly processing geometry for each additional view, as well as increased and potentially noticeable seam stitching artifacts. On the contrary, adapting the ray caster to wide FoV is trivial. Because it supports arbitrary ray distributions, the only changes required are to the way pixel footprints and tile bounds are computed.

In Section 6 we show that rendering a 210° x130° FoV image costs little more than rendering at the Oculus Rift FoV.

2.3 Foveation

We implemented a simple foveated renderer on top of HVVR . Because the system is not constrained to sampling on a uniform grid, it can closely match the resolution falloff recommended in [Guenter et al. 2012] with far fewer shaded samples. We visualize this sample distribution by performing nearest neighbor reconstruction in Figure 2. The results demonstrate a significant reduction in shading cost by rendering a foveated sample distribution. Foveated intersection performance in HVVR would benefit greatly from adaptive tile and block sizes.

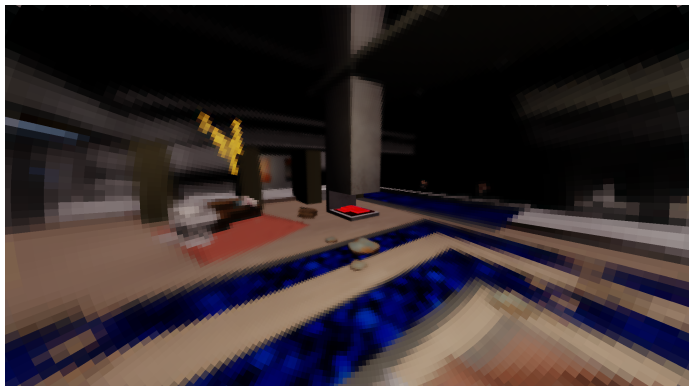

Fig. 2. A foveated image sampled and shaded using our ray casting framework. The final image was visualized using nearest-neighbor reconstruction to emphasize the non-uniform sampling pattern.

2.4 Stochastic Depth of Field/Defocus Blur

Defocus blur is a natural effect in all non-pinhole cameras and a commonly simulated effect in synthetically generated images. In virtual reality, depth of field is also an important effect in variable focus displays [Kramida and Varshney 2016] where content off of the plane of focus needs to be rendered with a synthetic blur to match the expectations of the human visual system. We demonstrate real-time stochastic depth of field using aperture and focal depth parameters similar to those of the human visual system, see Figure 3. A unique feature of our approach is that we use the same machinery for both defocus blur and MSAA, which allows the system to shade at a rate close to the display resolution, despite the significant number of additional rays. Shading differentials can be calculated based on the width of the depth of field packet, which in addition to providing correct pre-filtering, improves performance by requiring coarser mip levels during texture reads.

In addition to efficiently supporting defocus blur, ray casting does so accurately and composibly. In contrast, image space defocus blur algorithms [Demers 2004] don't integrate as naturally with a variety of effects such as foveation (due to the requirement of synthesizing dense, rectilinear color *and depth* maps from the foveated image before computing image space depth of field) or wide field of view (due to the non-uniform, anisotripic differences in angular resolution between pixels across the image plane, the need to simultaneously stitch multiple views, or both).

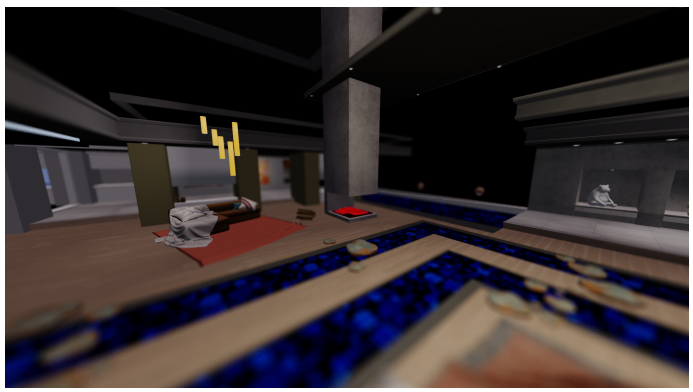

Fig. 3. An image calculated using accurate stochastic depth-of-field leveraging MSAA.

3 BACKGROUND

3.1 Z-buffering

Z-buffering is the de-facto real-time primary visibility algorithm, largely due to its applicability to uniform primary visibility and the availability and proliferation of inexpensive, specialized hardware implementations. The heart of the z-buffer algorithm is the z-buffer, a uniform grid data structure that stores the current closest hit depth for each sample/pixel. Most implementations of z-buffering assume samples/pixels are laid out in a uniform grid, matching precisely to the organization of the data structure. This assumption is broken in irregular z-buffer [Johnson et al. 2005] systems that we describe in more detail later in this section. The uniform nature of the grid structure, combined with the uniform distribution of samples mapped onto this grid, allows for a very efficient algorithm for determining which samples overlap a triangle. The process of mapping the spatial extent of an object onto the grid is known as *rasterization*.

A common optimization employed in z-buffer systems is to perform *tiled* rasterization [Molnar et al. 1994]. In a tiled system, samples/pixels are grouped into rectilinear tiles. Each tile's visibility is solved independently and the results are aggregated into a final image. The tiles themselves are organized into a second, coarse, grid, onto which geometry is first rasterized. After geometry has been distributed into the coarse grid, each tile can use the z-buffer algorithm to independently solve visibility at a tile granularity. Because the per-tile fine grids are small relative to the whole image, this approach improves cache utilization by reducing the working set significantly. The notion of solving visibility separately for many bins is used in other real-time systems such as [Hunt and Mark 2008], and is used in HVVR .

The uniform nature of the grid used in the z-buffer algorithm leads to high efficiency, but makes the algorithm inflexible. The assumed uniform sample distribution is reasonable when computing primary visibility from a virtual pin-hole camera for almost all direct view display technologies such as TVs, monitors or cell phones, leading to broad applicability of the algorithm. However, these assumptions do *not* hold for non-pinhole virtual cameras, secondary effects such as shadows and notably for modern virtual reality devices due to the distortion imposed by head mounted display viewing optics, and currently must be worked around on a case-by-case basis [Guenter et al. 2012; Patney et al. 2016; Popescu et al. 2009; Sun et al. 2017; Toth et al. 2016].

Algorithms such as the irregular z-buffer [Johnson et al. 2005] still use a uniform grid but allow for flexible number and placement of samples within each grid cell. Irregular z-buffering suffers from load-balancing issues related the conflict between non-uniform sample distributions in a uniform data structure, making it significantly more expensive than traditional z-buffering. While such an approach could be used for some VR features such as pre-distortion and foveated rendering, it still supports only a limited field of view, and doesn't support depth of field. Both original and recent [Wyman et al. 2015] work on irregular z-buffering have focused on shadows, and the algorithm's applicability to primary visibility has not been studied.

3.2 Ray Tracing

In contrast to z-buffering, ray tracing algorithms take a more general approach to determining visibility by supporting arbitrary point-to-point or ray queries [Whitted 1980]. This flexibility supports distributed effects [Cook et al. 1984] and Monte Carlo simulation of arbitrary light transport. The ability to effectively model physically-based light transport and naturally compose effects such as defocus blur, motion blur, and indirect effects led it to be the dominant rendering algorithm in the movie industry [Keller et al. 2015]. The flexibility ray tracing provides comes at significant cost in performance, which has prevented it from becoming prevalent in consumer real-time applications. Although it is still not competitive with z-buffering for primary visibility,

significant efforts have been made to improve the performance of ray tracing [Wald et al. 2007b]. In the remainder of this section we describe some of the approaches and how they relate to ours.

3.2.1 Acceleration Structures. In order to improve performance, ray tracers typically use an acceleration structure to organize scene geometry [Goldsmith and Salmon 1987]. These structures are either space partitioning (grids, kd-tree, BSP tree, octree) or object partitioning (bounding volume hierarchy). By organizing the geometry into spatial regions or bounding them in enclosing volumes, the structures allow a ray tracer to avoid testing rays with objects if they don't enter the volume bounding the object.

Because they are adaptive to local geometric complexity, are efficient to build [Karras and Aila 2013] and support dynamic geometry [Wald et al. 2007a], axis aligned bounding volumes have become the dominant acceleration structure for ray tracing and are used in a variety of popular ray tracing kernels such as NVIDIA's OptiX [Parker et al. 2010] and Intel's Embree [Wald et al. 2014].

An axis-aligned bounding volume hierarchy (often abbreviated BVH) is a tree data structure that stores scene geometry (usually triangles) at the leaves of the tree and an axis-aligned bounding box at each node. These bounding boxes conservatively enclose all of the geometry for a node's sub-tree. Rays (or other visibility queries such as beams) are traversed recursively through the tree from the root and are tested against nodes' children's bounding volumes. Recursive traversal of a node's children only occurs in the case of intersection, so rays can avoid traversing portions of the tree that they miss. State-of-the-art performance in ray tracing is acheived by modifying the approach taken in OptiX by augmenting the BVH to enable stackless traversal [Binder and Keller 2016].

Traditionally BVHs are binary, but other branching factors have become popular. Particularly, 4-way trees [Dammertz et al. 2008] provide an overall benefit by reducing the number of internal tree nodes and exposing additional parallelism during traversal. HVVR uses a 4-way axis aligned BVH.

Supporting animation in a ray tracer requires rebuilding or refitting the acceleration structure on a per-frame basis in order to account for movement. A significant amount of work has gone toward reducing the amount of time it takes to rebuild BVHs [Karras and Aila 2013; Wald et al. 2007b] and a BVH for 1 million triangles can be built in under 10ms. For small or coherent movements, where the topology of the BVH remains a good match to the geometry after animation, the BVH bounds can simply be updated to match the animated geometry. This refit is almost an order of magnitude faster than rebuild (see Section 6). For games, animated characters can be divided into a few dozen coherent sections that each never need a full rebuild. Since static geometry never needs updates, only dynamic objects and sections ever require a full rebuild.

3.3 Coherent Ray Tracing & Ray Hierarchies

In order to improve the efficiency of ray tracing, many strategies have been developed for exploiting spatial coherence between rays to eliminate redundant computations. Each strategy is distinguished by the type of ray hierarchy and traversal algorithm.

The Ray Engine [Carr et al. 2002] stored rays in a 5D ray tree (using ray position and direction), and simultaneously traversed the scene and ray hierarchies on the CPU, before doing final intersection on the GPU. HVVR borrows the hybrid aspect of this approach, though the actual data structures and traversal algorithms differ. Roger et al also use a hybrid approach, but completely forgo a scene hierarchy to specialize a Whitted raytracer for rendering of fully dynamic scenes, using a deep ray hierarchy [Roger et al. 2007].

Entry point search algorithms, such as MLRTA [Reshetov et al. 2005] and coherent packet tracing such as DYNBVH [Wald et al. 2007a] aggregate rays into packets (usually rectilinear collections)

and conservatively bound them using ray-intervals or beams. By testing these conservative intervals/beams against a bounding volume during traversal, a single test can potentially reject a subtree traversal for an entire collection of rays, resulting in a multiplicative reduction in work. These approaches essentially build a simple two-level hiearchy on rays.

In addition to enabling rejection of sub-trees, conservative tests can also eliminate redundant work in the case the interval/beam intersects a bounding volume. During an entry point search algorithm, an interval/beam continues until a specified termination criterion has been met, and the terminating nodes are collected as a set of *entry points*. Once the set of entry points has been determined by the interval/beam traversal, individual rays can begin traversal from the entry points, rather than the tree root, avoiding unnecessary intersection tests near the root of the tree. Entry point search algorithms typically occur in two phases, with intervals/beams being traced first, followed by individual rays. We extend this to a three stage entry point search algorithm: first traversing coarse beams, then fine beams and finally individual rays. This multi-phase technique is amenable to a pipelined approach and different hardware can be used at each stage, a feature we exploit in HVVR .

CHC+ RT[Mattausch et al. 2015] combines a hierarchy with an entry-point search for raytracing of extremely large scenes for visualization. The authors build a BVH over the scene, and a screen-tile quadtree over pixels, which are simulataneously traversed until the active tile and scene subset is sufficiently small to do final traversal and intersection. The partial traversal is saved between frames and used to seed a "depth" buffer that is combined with hardware occlusion queries to early-out traversal. We leave integrating such a temporal component with HVVR to future work.

Approaches involving ray hierarchies tend to degrade quckly with decreasing coherence of rays, and thus have not had much success outside primary visibility. [Garanzha and Loop 2010] adopt a fast sorting strategy to recover coherence for very wide packet tracing on the GPU, but similar approaches have not been adopted in newer state-of-the art raytracing engines, like OptiX [Parker et al. 2010] and Embree [Wald et al. 2014] which primarily use acceleration structures over geometry (though Embree also supports small ray packets). This contrasts with the previously described z-buffer algorithm, which only uses an acceleration structure for rays.

3.4 Hybrid Approaches

Patidar et al. introduced an early GPU raycasting system designed to work on fully deformable scenes [Patidar and Narayanan 2008]. Their approach is actually a two stage hybrid algorithm that first conservatively rasterizes triangles into a projective 3D grid, and then performs ray triangle intersection for each ray in a tile and each triangle in a given bin.

3D Rasterization [Davidovič et al. 2012] makes explicit the deep connection between rasterization and raycasting, by folding them into a single algorithm, using 3D edge functions to evaluate intersections. The approach can handle a subset of the VR functionality from Section 2 by using a non-planar viewport and case-by-case geometric reasoning. The system can also handle many techniques developed for rasterization, in particular they evaluate MSAA[Akeley 1993] in their framework to reduce shading load. We generalize MSAA for non-pinhole camera models so that we can do the same.

4 DESIGN PRINCIPLES

HVVR is a complete system for determining primary visibility (and shading) for arbitrary camera configurations, designed as an evolving part of a larger research ecosystem. We detail the current state of the system in Section 5, but first we lay out the principals guiding HVVR 's development:

(1) HVVR should transparently handle all of the use cases detailed in Section 2.

(2) Current raytracing systems are bottlenecked by acceleration structure traversal; HVVR should minimize traversal cost by sharing as much traversal work across as many rays as possible.
(3) Both the CPU and GPU should be used for the portion of the system they perform best at.
(4) HVVR should extend to extremely high anti-aliasing rates to ensure high-quality in VR use cases.
(5) Memory bandwidth is increasing at a slower rate than computation with new hardware generations; HVVR should err on the side of recomputation over extra storage.

Principle (1) led to the choice of a raycasting algorithm; (2) and (3) led to the choice of a CPU-based entry-point search algorithm, which amortizes traversal cost, and GPU-based ray-triangle intersection, which takes strong advantage of the raw computational power of modern GPUs. Principles (4) and (5) require cheap ray generation, and a work-sharing solution for shading. Our solutions were just-in-time subsample ray generation (Section 5.5) and our MSAA technique (Section 5.1).

5 SYSTEM

In this section, we detail our system. In summary, HVVR is a ray caster that uses a three level entry point search algorithm. The system is implemented in a heterogeneous manner, with beam traversal occurring on the CPU and ray-triangle intersection and shading occurring on the GPU. The system supports some standard features such as animation via linear blend skinning, MSAA, and physically based shading. It also supports a variety of VR specific features described in Section 2.

Primary rays are assigned a footprint for antialiasing and are then aggregated into a two level hierarchy with 4-sided bounding beams. The finer level of the hierarchy aggregates 16x8 tiles of pixels. These tiles are then aggregated into a coarser collection of blocks, each containing 8x8 tiles. Thus, each block represents 8192 pixels and a total ray count of that multiplied by the multi-sampling rate. Our choice of 128:1 between pixels and tiles and 64:1 tiles and blocks is based on coarse tuning for the hardware we use, but we do not claim that it is globally optimal.

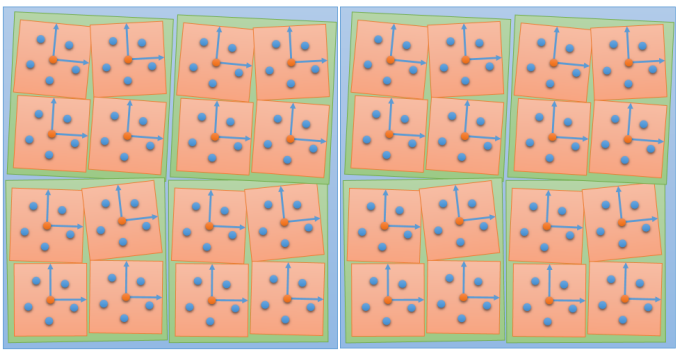

Fig. 4. An example sample hierarchy, with much smaller tiles and blocks than in the actual system. Pixel footprints in orange, with orange pixel center, blue differentials and blue subsample locations. Pixels are aggregated into tiles (green), and tiles are aggregated into blocks (blue).

When any kind of lens distortion is present, the distortion is applied to the rays before bounding beams are computed via the dominant axis technique decribed in [Overbeck et al. 2008]. When subpixel rendering is enabled, red, green and blue channels are considered separate grids and tiled and blocked separately. When depth of field is present, the beams are expanded to accommodate the distribution of ray origins. When using a foveated ray distribution, rays bundles are generated using a top-down divisive algorithm to build tiles containing no more than 128 pixels and blocks

containing no more than 64 tiles. The system supports partially occupied tiles and blocks. For most use cases, these bounds may be computed once, at the beginning of time, based upon the lens parameters of the system. However, in the case that parameters change between frames, such as the point of attention during foveated rendering, they must be recalculated on a per-frame basis. The bounding beams bound the entire footprint of every pixel, rather than just their centers, to support MSAA.

The system uses a 4-way axis-aligned BVH as the geometry acceleration structure. Presently a single, combined BVH is used for all scene geometry but the system is evolving towards a multi-level approach to allow for instancing and to enable more efficient animation by allowing for more granular BVH rebuilds and refits. Rudimentary animation is supported via global BVH refit per frame.

Our implementation lays the BVH out in memory in depth-first preorder and stores triangles in a contiguous array, in the order they would be touched in a depth-first traversal of the BVH. Additionally, any node with a mix of leaf and internal children stores the leaf children first. With these assumptions, iterating in reverse through the list of BVH nodes guarantees that a node's children will always be visited before it will and that all triangles will be visited in a linear, reverse, order. These assumptions enable a linear, non-recursive BVH refit algorithm and improves cache locality during refit, traversal and intersection.

Our implementation uses an AVX-optimized frustum traversal for block (and tile) culling and a CUDA implementation for ray generation, ray-triangle intersection, and shading.

In the following subsections we first detail our MSAA generalization, and then describe the stages and implementation of the algorithm in more detail.

5.1 Multi-Sample Anti-Aliasing for Rays

Multi-sample antialiasing (MSAA) is the most popular anti-aliasing algorithm for real-time graphics, and is a recommended technique by many in industry[2,3]. When using traditional MSAA, triangles are tested against many procedurally defined subsample positions within a single pixel, and a depth value is tested and stored for each subsample, but each triangle is only shaded once per triangle.

We extend this concept for rays, using ray differentials [Igehy 1999]. We encode a ray footprint with two vectors mutually perpendicular with themselves and the ray direction. The extent of the ray footprint is defined by the length of these vectors. Subsamples (or subrays) are procedurally generated within this footprint by first transforming a low-discrepancy point set on the unit square using the coordinate frame defined by the ray direction and footprint vectors, adding the ray direction to the transformed points, and defining the subray to be the ray through the original ray's origin and the newly transformed points. For depth-of-field rays, we also choose the ray origin using a separate low-discrepancy point set (without translating along the ray direction).

We can then perform shading one per pixel per triangle, as in regular MSAA, which saves a large amount of shading work (or shade every sample, to get full SSAA). Since the subrays are procedurally generated, we can cut the ray memory bandwidth by the anti-aliasing factor when compared to naively rendering at higher resolution.

5.2 Scene Update and Triangle Precomputation

Before rendering begins, animation is performed and the BVH is refit if required. Bone animation occurs on the CPU, while linear blend skinning and BVH refit are implemented in a series of CUDA kernels in the following stages:

[2] https://developer.oculus.com/blog/introducing-the-oculus-unreal-renderer/
[3] http://www.gdcvault.com/play/1021771/Advanced-VR

(1) Transform vertices (perform linear blend skinning)
(2) Clear BVH node bounds
(3) Precompute triangles
  (a) Gather vertices
  (b) Compute edge equations (for a Moller-Trumbore ray-triangle intersection)
  (c) Compute triangle bounds and atomically update corresponding leaf bounding box
(4) Refit BVH by propagating bounds from leaf nodes up through internal node hierarchy

After refit is performed on the GPU, the BVH is copied back to CPU memory for the block and tile traversal stages. At this point, block and tile bounds are computed and refit, if needed.

5.3 Block Culling

The first stage of visibility computation is the block culling phase. During this phase, the beams defined by the 128x64 pixel blocks are traversed through the hierarchy. The implementation of beam traversal uses an explicit stack AVX implementation, very similar to the ray-traversal kernel of Embree [Wald et al. 2014]. However, there are a few key differences. Because block traversal is a culling/entry point search phase, rather than traversing all the way to the leaves, as a traditional ray tracer would do, block traversal only traverses until it reaches a specific stopping criteria (presently, when 64 entry points have been discovered).

The explicit traversal stack is initialized with the root box and at each step during traversal the thickest box along the primary traversal axis in the traversal stack is tested. This allows us to more efficiently refine the nodes down to individual surface patches. Despite the overhead of sorting, this improved combine tile/block culling performance by 5-10% during development.

Each beam/box test can have one of 3 outcomes:

(1) Miss - the beam misses the box entirely, and that box's subtree can be discarded.
(2) Fully contained - the beam contains the box entirely. In this case, the box is accumulated as an entry point and no further traversal of that subtree is required as it is transitively fully contained.
(3) Partial intersection - the beam and the box intersect. In this case the subtree associated with the box is inserted into the traversal stack for continued refinement.

Traversal continues until one of the following conditions are met:

(1) The traversal stack is empty. The accumulated fully contained boxes are sorted in near depth order and passed onto tile cull.
(2) The sum of size of the traversal stack and the size of the fully contained box list equals a prespecified value, presently $\alpha = 64$. Then the lists are merged, sorted in near depth order and passed onto tile cull. Thus, no more than a fixed number of entry points are ever passed from block cull onto tile cull. This parameter can be increased to increase the amount of block traversal for blocks with high scene complexity.

During traversal, the separating axis theorem is used to determine separation between bounding boxes and the beam. When sorting the entry points before hand off to tile cull, the near plane along the dominant axis of the beam is used as the key value.

5.4 Tile Culling

The tile culling phase picks up where the block culling phase left off. Each entry point list produced by a block cull is used in 64 tile cull phases. Tile culling is implemented in an explicit stack AVX traversal, as in block cull. However, rather than beginning by initializing the traversal stack with the root box, the traversal stack is initialized by copying the output of the associated block cull. In this way, tile cull avoids duplicating a significant amount of traversal, performed during block cull. The

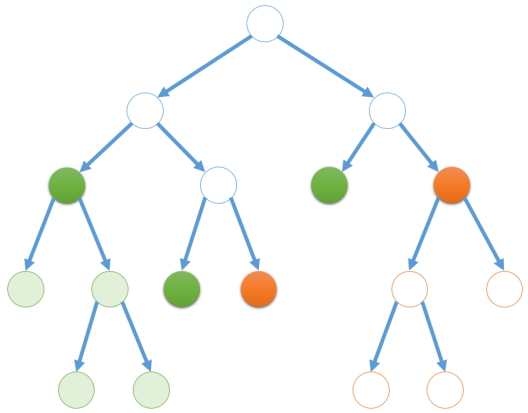

Fig. 5. An example of a tree after block-level entry point search. Hollow blue nodes have been traversed. Solid orange nodes have been tested and determined to not intersect the beam. Hollow orange nodes were never traversed or tested but were culled by a test on an ancestor. Dark green nodes were determined to be good entry points and will be used to initialize the subsequent phase. Light green nodes have not been tested nor culled and will be visited in a subsequent phase.

beam/box tests have the same potential outcomes as in block cull, but traversal always continues until the traversal stack is empty. Once all triangles have been gathered, they are transferred to the GPU for sample testing.

The greedy nature of tile cull is currently a limitation of the system. In high depth complexity scenes, excessive numbers of triangles are eagerly gathered and potentially tested, despite the fact that they may be occluded by nearer geometry. Short circuiting tile traversal will require interleaving tile cull and sample testing, which implies migrating tile cull to a CUDA implementation, a direction in which the system is evolving.

5.5 Sample Testing

Sample testing is performed after the tile culling phase, and broken into per-tile and per-subpixel phases. Both phases are completed using a single CUDA kernel with a workgroup size of 128. In the per tile portion, threads are mapped 1:1 with triangles and in the per-subpixel phase threads are mapped 1:1 with pixels or subpixels. (Note: A subpixel is an individual LED, red, green or blue, and is distinct from a subsample in the multi-sample anti-aliasing sense. A subpixel may have many subsamples.) The system supports both multi-sample anti-aliasing (MSAA) and super-sample anti-aliasing (SSAA), the distinction being that in MSAA shading is performed only once per pixel per triangle and the results are shared across all subsamples of that pixel that strike the same triangle, and that in SSAA shading is computed separately per subsample. The advantage of MSAA is a potentially large reduction in shading rate. Triangle data for the tile is gathered into a shared local cache on the GPU for ease of access from all samples. This triangle cache has 128 entries. The per-tile and per-subpixel phases alternate until all triangles for a tile have been processed. Initialization is performed by clearing all of the per-subsample depth and index values, which are stored in per-thread registers and shared memory, respectively. These depth values serve a similar purpose to the depth buffer in a traditional rasterization system, and could conceivably be compressed in a similar manner [Hasselgren and Akenine-Möller 2006], though keeping the values uncompressed in registers for fast depth-testing is performant.

The per-tile phase contains the following steps:

(1) Gather triangle data into a shared memory cache.

(2) Perform back-face and near plane culling on the triangles.
(3) Test tile corner rays against triangles and classify as “all in”, “partial” and “all out”.
(4) Perform common origin intersection precomputations (when applicable).

Once the per-tile phase has completed, each thread associates itself with a pixel or subpixel, and performs the following steps:

(1) Look up sample location and differentials.
(2) Transform ray and differentials to world space.
(3) Iterate over cached triangles
  (a) Fetch the triangle from the cache.
  (b) Iterate over subsamples.
    (i) Compute subsample offset within the pixel footprint via lookup table.
    (ii) Compute ray-triangle intersection [Möller and Trumbore 1997] for the ray defined by this subsample, including depth-test.
    (iii) Update subsample depth and triangle index in the case of successful intersection.

Subsample rays are regenerated just-in-time from the pixel footprint for the ray-triangle intersection. This adds some extra computation to the inner loop, but saves on memory bandwidth and register usage.

Once all of the triangles have been processed, visibility for the tile is fully resolved and the per-subsample depth registers and triangle index shared memory buffers contain the closest hit for each subsample. This data is then compressed and emitted to a “gbuffer” in preparation for shading. The “gbuffer” in this case consists of only visibility information: pairs of triangle indices and subsample masks, which is sufficient to recompute barycentrics and fetch vertex attributes in the shading phase (depth is redundant information and is discarded). The “gbuffer” is allocated in global memory on the GPU, and sized to handle the worst-case, where each subsample of each subpixel strikes a different triangle. Memory is arranged such that the first triangle for each pixel are adjacent in memory, followed by the second triangles etc., so in practice only a small prefix of this buffer is actually used. Compression performs the following steps:

(1) Sort the subsamples by triangle index.
(2) Iterate over the subsamples and emit (triangleIndex, multiSampleMask) for each unique triangle index.

Once the “gbuffer” has been constructed the sample testing phase is complete and the resolve/shading phase begins.

5.6 Resolve / Shading

After visibility has been computed during the sample testing phase, the Resolve / Shading pass performs shading, aggregates MSAA or SSAA samples and computes final pixel/subpixel color to a buffer that can be presented for display. We use GGX [Walter et al. 2007] for our BRDF in our results. Each sample location is read from the output of the previous stage and ray intersections are computed for each “gbuffer” entry at the pixel center. Then shading is performed using attributes interpolated using barycentrics obtained during intersection, and the shading result is accumulated per pixel. Once all “gbuffer” entries are processed per pixel, we perform filmic tonemapping and output the results to the final buffer for display. In the case of SSAA, rays are generated, intersected, and shaded independently rather than having a single weighted shading result per entry.

6 PERFORMANCE RESULTS

The performance of a rendering system is difficult to quantify, as rendering is highly sensitive to the input scene, camera configuration, and underlying hardware architecture. This difficulty is

increased by the fact that HVVR , as a ray*caster* is a fundamentally different system than any of the industry standard state-of-the-art rendering systems. It aims to be significantly more flexible than hardware rasterizers while still restricting the workload to primary visibility in order to enable significant optimizations not available in general raytracing engines like Embree and OptiX.

We choose to evaluate HVVR 's performance on a desktop computer with a Quadro P6000 running on the Windows 10 with an Intel Core i7 6850K: a high-end configuration available today. We evaluate on a test suite of four scenes, pictured in Figure 6, chosen to approximate the complexity of VR scenes in use today. These scenes are intentionally lower complexity than those chosen for most production raytracing benchmarks, partially because of the low angular resolutions of present day VR headsets and partially because we are targeting frame rates in excess of 90Hz.

We first demonstrate that HVVR acheives real-time performance for primary visibility at high resolutions and subsample counts, and provide detailed timings, broken down by stage. Additionally, we also evaluate the cost of BVH refit, the performance advantages of MSAA vs. SSAA and performance of a variety of non-uniform primary visibility configurations unsupported by hardware z-buffering. We finally demonstrate that HVVR outperforms renderers built in Embree and OptiX Prime for the VR workloads in our test suite.

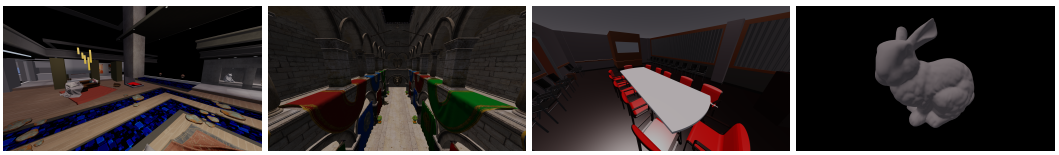

Fig. 6. The four scenes we use to evaluate performance. From left to right: Home , Sponza , Conference , Bunny

6.1 Base Performance

Table 1. Performance breakdown of our BVH refit implementation (Section 5.2) on our test scenes. Refit takes a total of less than half a millisecond on all test scenes. Note that in practice, only a subset of a given scene may need to be refit every frame.

Scene	# Verts	# Tris	# Nodes	Total (µs)	Vert Xform (µs)	Initialize Bounds (µs)	Compute Tris & Leaf Bounds (µs)	Fixup Bounds (µs)	Propagate Bounds (µs)
Home	115,986	159,588	17,422	139	27.7	4.9	61.6	13.2	31.6
Bunny	34,834	69,451	8,196	69	10.8	2.9	26.3	10.4	18.6
Conference	949,014	331,179	34,391	481.7	203.4	9.5	182	25.8	61
Sponza	184,406	262,137	27,872	208.2	41.2	8.1	97.1	17.2	44.6

In order to demonstrate the small costs of refitting BVHs, we provide full scene refit times in Table 1. We report times for each stage of the refit on all of our test scenes. The entire refitting process takes less than a half millisecond on all scenes. These results indicate that acceleration structure refit times are not a blocking issue in creating a real-time ray caster (or ray tracer) for dynamic scenes. For implementation simplicity, we refit the entire scene whenever any part of the scene changes. A more optimized implementation would avoid refitting static geometry.

Rendering times for a 2160x1200 screen with 32x MSAA for all scenes are provided in Table 2. HVVR easily makes framerate at even 120Hz with several milliseconds to spare, demonstrating that ray casting is viable for real-time applications on current hardware. The distribution of work between CPU Traversal, GPU Intersection, and GPU shading is highly scene-dependent, as is evident by comparing the Home and Bunny scenes; the Bunny has very low scene complexity and thus requires much less time to traverse on the CPU, but spends a comparable amount of time on intersection and shading.

Table 2. Performance breakdown for a regular grid sampling pattern at 2160x1200 on all scenes at 32x MSAA. CPU and GPU work is pipelined between frames, so total frame time is lower than the sum of each stage. In these configurations, the GPU work (Intersect + Shade) is dominant and hides the cost of the CPU Traversal. The Intersect timing results include data-transfer overhead for triangle indices, which are streamed over on demand by the GPU kernel.

Scene	Frame Time (ms)	CPU Traversal (ms)	Intersect (ms)	Shade (ms)	Avg # Shade Samples Per Non-empty Pixel
Home	4.50	2.34	2.83	1.25	1.2
Bunny	3.36	1.37	1.17	0.35	1.62
Conference	5.24	2.43	3.96	0.93	1.25
Sponza	6.70	4.49	4.91	1.09	1.26

6.2 Subpixel Rendering

In order to support large numbers of per-pixel primary visibility samples and as described in Section 5.5, HVVR uses procedural subray generation, as described in Section 5.1 and defaults to MSAA instead of SSAA. In Table 3, we show that super sampling, which drastically lowers bandwidth through procedural generation of rays, lead to five times improvement in intersection performance over naively increasing resolution for 16 rays per pixel. In addition, Table 4 shows that shading once per triangle per pixel (MSAA) leads to an order of magnitude faster frame to frame time. With this optimization, shading is not a bottleneck even when a large number of primary samples are used for anti-aliasing and/or depth of field.

Table 3. Using procedural subrays for supersampling drastically decreases bandwidth compared to naively using a higher resolution ray buffer, which leads to a substantial decrease in intersection time. Benchmarked on the Home scene at 2160x1200 with 16 samples/pixel.

Ray Mode	Intersect Time (ms)	Ray Memory (MB)
High-resolution	6.92	497.7
Supersampling	1.35	31.1

Table 4. Using MSAA drastically descreases the shading rate per-pixel, which leads to an order of magnitude performance improvement in shading. Benchmarked on the Home scene at 2160x1200 with 32 samples/pixel.

Shading Mode	Shade Time (ms)	Avg # Shade Samples Per Non-empty Pixel
SSAA	18.12	31.09
MSAA	1.25	1.2

6.3 Rendering Techniques

To demonstrate viability as a rendering system for VR we evaluated performance for the applications described in Section 2. Most of these applications are impossible to directly implement in a single pass using current-generation hardware accelerated z-buffering but are gracefully handled by HVVR . The timing breakdown is given in Table 5. HVVR suffers performance degradation as tile and block beams become larger, which occurs when rendering defocus and in the sparse regions during foveated rendering, but maintains real-time framerates.

Table 5. Comparison of our raycaster on a variety of sampling schemes. DoF denotes depth-of-field rendering with a 3mm thin lens approximation, focal distance is given in parenthesis. HVVR easily handles a large variety of sampling schemes which would be impossible to efficiently leverage a standard rasterizer for, with gracefully degrading performance. HVVR 's performance degrades when bounding frusta of tiles or blocks become large (as more work is deferred until the per-subsample level), but still reaches realtime rates even with 32x MSAA.

Technique	Shading Configuration	Sample Count	Full Frame (ms)	CPU Traversal (ms)	Intersect (ms)	Shade (ms)	Shade Count (per pixel)	Throughput MRays/s
Normal	8x	2,592,000	4.50	2.34	0.87	0.92	1.15	4,608
Normal	16x	2,592,000	4.50	2.34	1.35	1.05	1.17	9,216
Normal	32x	2,592,000	4.50	2.34	2.83	1.25	1.20	18,432
VR Pre-Distort	32x	4,225,104	6.51	3.31	4.00	1.85	1.20	20,769
VR Post-Distort	16xR/B + 16xG	4,724,260	7.50	3.90	2.42	1.93	1.32	10,078
VR Post-Distort	32xR/B + 32xG	4,724,260	8.08	3.90	5.13	2.31	1.36	18,710
Foveated	32x	319,488	5.41	0.94	2.96	0.30	1.31	1,890
Wide FoV 210x130	32x	2,592,000	5.42	3.01	3.60	1.45	1.18	15,303
DoF (.5m)	16x	2,592,000	5.57	2.56	3.76	1.44	1.45	7,446
DoF (.25m)	16x	2,592,000	6.66	2.73	4.62	1.69	1.89	6,227
DoF (.1m)	16x	2,592,000	10.33	3.46	7.77	2.24	2.94	4,015
DoF (.5m)	32x	2,592,000	11.96	2.56	9.70	1.95	1.54	6,935
DoF (.25m)	32x	2,592,000	14.41	2.73	11.87	2.25	2.07	5,756
DoF (.1m)	32x	2,592,000	23.07	3.46	20.10	3.06	3.65	3,595

6.4 Comparison with OptiX Prime and Embree

Table 6. HVVR comparison against two state-of-the-art ray*tracers*, Embree and OptiX Prime. Throughput is reported in millions of rays per second (MRay/s). Throughputs for Embree and OptiX Prime are computed based on traversal and intersect time *only*, while throughput for HVVR is based on *full frame to frame* time, including GGX shading and driver overhead. By explicitly leveraging ray coherence, our ray caster outperforms these ray tracers by over 20x. Image resolution is 2160x1200 with 32x MSAA.

	Home	Bunny	Conference	Sponza
HVVR	18,432	24,674	15,840	12,375
Embree	322	543	263	223
OptiX Prime	578	529	568	552

In order to validate that the optimizations available to ray casters provide compelling performance advantages over a general purpose ray tracing engines, we compare HVVR to OptiX and Embree, both state-of-the-art general raytracing systems. In order to perform the comparison, we implemented simple ray casters using OptiX and Embree aimed at minimizing non-intersection work (e.g. shading) and bandwidth. For simplicity, and to favor OptiX Prime and Embree, we measure "speed-of-light" times for them, measuring only traversal and intersection (and using acceleration structures that favor traversal speed over build speed when possible), and compare those with standard usage of HVVR , which includes a full GGX shading model, tonemapping, and driver and display overhead.

For OptiX, we used the low-level OptiX Prime API, which, after an acceleration structure is built over triangles, simply takes a buffer of rays and produces a buffer of hitpoints. We leverage the GPU intersection kernels, and keep all data on the GPU to avoid including transfer time in our profiling. In order to minimize bandwidth and provide the most favorable comparison possible, we configure the rays to contain only an origin and direction, and the hitpoints to be a single bit that indicated if the ray intersected the scene.

For Embree, we used the C++ API and configured it to run in stream processing mode, with 8-ray packet tracing. This packet size produced the highest ray throughputs in our experiments. We also used Intel's Thread Building Blocks to fully multithread the packet tracing. No extra processing is done once rays are intersected, the only write-bandwidth is from the writes Embree uses internally to fill out the ray datastructure. We note that the relatively low ray rates compared to OptiX are likely due to the unbalanced CPU-GPU configuration we profile on.

The full performance breakdown is given in Table 6. HVVR outperforms both ray tracers on this workload from 4x to over an order of magnitude despite HVVR performing full shading and account for driver and display overhead; confirming our hypothesis that specialized ray casting algorithms can easily outperform even highly-tuned general-purpose ray tracers.

7 FUTURE WORK

The system presented is a snapshot of an ongoing research infrastructure project and will be subject to a significant amount of additional development. We believe the fundamentals are in place and have been documented in this publication. Because of significant differences between our approach and the current z-buffer based graphics pipeline, many common features will need to be reimplemented or reimagined. We provide the implementation as a BSD-style open sourced project[4] and actively encourage community engagement and development. Here we list a variety of next steps.

Multi-Tier BVH & Instancing. Introducing a multi-level BVH would allow HVVR to support instancing and partial data-structure rebuild for dynamic geometry, in turn supporting highly dynamic, game-like content. An additional benefit of a multi-level BVH is the ability to support *massive* amounts of geometry, including instances, without the need for additional content-specific visibility culling systems, such as forests or cityscapes.

Particle Systems. Presently, no ray tracing or ray casting system that the authors are aware of has explicit support for particles. It's possible that current BVH build technology could provide real-time performance for sufficient numbers of dynamic particles for VR use-cases. Alternatively, a specialized acceleration structure and culling and traversal mechanism may be required.

Improved Visibility Culling. Interleaving tile culling with sample testing would allow the results of sample testing to short-circuit traversal, saving a potentially significant amount during the traversal phase. In order to interleave these phases, tile culling needs to be moved to the GPU to avoid a costly CPU-GPU synchronization.

Multiple Materials & Improved Shading Efficiency. Presently the system does not contain machinery to support a variety of materials simultaneously. Supporting multiple materials efficiently would involve inserting a batch sorting pass between sample resolve and shading to aggregate shading results by material prior to shading. Batch sizes would need to be tuned to balance memory usage, reorder potential and latency.

Partial Transparency. The depth order nature of the ray caster makes it ideally suited to compute efficient ordered transparency. Supporting ordered transparency requires interleaving sample testing and shading, or recomputing sample testing multiple times with different minimum depths (depth peeling).

Beam Racing. Due to its pixel major nature, the ray casing system is capable of rendering an image *just in time*, reducing system latency to a few lines. Even significant batching, such as

[4] https://github.com/facebookresearch/HVVR

hundreds of lines (hundreds of thousands of pixels) could provide large multiplicative reductions in latency over waiting for the full frame before scan-out. Beam racing requires more coordinated interaction with scan-out hardware than is presently exposed by graphics APIs.

Motion Blur & Rolling Shutter. Support for camera movement during the frame would require per-sample time offsets and changes to the way conservative frusta are computed. This change would enable native distortion correction for a rolling illumination display, such as in the Oculus DK2, and could be used in conjunction with beam racing. In addition, HVVR could be modified to support motion blur by linearly interpolating bounding volumes [Gruenschloß et al. 2011] at the cost of increased BVH memory and traversal time.

Dynamic Tessellation. The hardware graphics pipeline supports dynamic tessellation, in order to better match the geometric resolution to the spatial sampling resolution. Tessellation within ray tracing systems is desirable but non-trivial [Benthin et al. 2015].

Ray Tracing Integration. Although we have explicitly chosen to accelerate coherent primary visibility in HVVR , the integration of an optimized higher order ray tracer would enable secondary effects such as reflection and refraction, as well as more efficiently handling cases where coherence naturally breaks down, such as in regions of an image with incredibly high geometric complexity.

8 CONCLUSION

We introduced a primary visibility algorithm based on ray casting that provides real-time performance at large samples/pixel and features suited to rendering for virtual reality displays. We describe our approach and demonstrate that the algorithm is capable of rendering in real time on current GPU hardware at current virtual reality resolutions and frame-rates. We conclude with a collection of future directions for the system and a link to the open source repository to encourage community engagement. The authors believe that virtual and augmented reality present a novel set of requirements, distinct from those of traditional displays, and believe that new approaches, such as HVVR , are required to implement visibility efficiently. Ultimately, novel specialized hardware will be required, but in the meantime we demonstrate that a software solution, leveraging the last dozen years of research in real-time ray tracing, can be made real-time to enable exploration of algorithms and applications.

REFERENCES

Kurt Akeley. 1993. Reality engine graphics. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. ACM, 109–116.

Carsten Benthin, Sven Woop, Matthias Niessner, Kai Selgrad, and Ingo Wald. 2015. Efficient Ray Tracing of Subdivision Surfaces Using Tessellation Caching. In *Proceedings of the 7th Conference on High-Performance Graphics*. 5–12.

Nikolaus Binder and Alexander Keller. 2016. Efficient stackless hierarchy traversal on GPUs with backtracking in constant time. In *Proceedings of High Performance Graphics*. 41–50.

Nathan A. Carr, Jesse D. Hall, and John C. Hart. 2002. The Ray Engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS '02)*. Aire-la-Ville, Switzerland, Switzerland, 37–46.

Robert L. Cook, Thomas Porter, and Loren Carpenter. 1984. Distributed Ray Tracing. *SIGGRAPH Comput. Graph.* 18, 3 (1984), 137–145.

Holger Dammertz, Johannes Hanika, and Alexander Keller. 2008. Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays. In *Proceedings of Eurographics Conference on Rendering (EGSR)*. 1225–1233.

Tomáš Davidovič, Thomas Engelhardt, Iliyan Georgiev, Philipp Slusallek, and Carsten Dachsbacher. 2012. 3D rasterization: a bridge between rasterization and ray casting. In *Proceedings of Graphics Interface 2012*. 201–208.

Joe Demers. 2004. Depth of Field: A Survey of Techniques. In *GPU Gems*. Addison-Wesley, 375–390.

Kirill Garanzha and Charles Loop. 2010. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. In *Computer Graphics Forum*, Vol. 29. 289–298.

Jeffrey Goldsmith and John Salmon. 1987. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Comput. Graph. Appl.* 7, 5 (1987), 14–20.

Leonhard Gruenschloß, Martin Stich, Sehera Nawaz, and Alexander Keller. 2011. MSBVH: An Efficient Acceleration Data Structure for Ray Traced Motion Blur. In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*.

Brian Guenter, Mark Finch, Steven Drucker, Desney Tan, and John Snyder. 2012. Foveated 3D graphics. *ACM Trans. Graph.* 31, 6 (2012), 164.

Jon Hasselgren and Tomas Akenine-Möller. 2006. Efficient Depth Buffer Compression. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (GH '06)*. ACM, New York, NY, USA, 103–110. https://doi.org/10.1145/1283900.1283917

Warren Hunt and W. R. Mark. 2008. Ray-Specialized acceleration structures for ray tracing. *IEEE Symposium on Interactive Ray Tracing 2008* 00 (2008), 3–10.

Homan Igehy. 1999. Tracing Ray Differentials. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)*. New York, NY, USA, 179–186. https://doi.org/10.1145/311535.311555

Gregory S. Johnson, Juhyun Lee, Christopher A. Burns, and William R. Mark. 2005. The irregular Z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph.* 24, 4 (2005), 1462–1482.

Tero Karras and Timo Aila. 2013. Fast Parallel Construction of High-quality Bounding Volume Hierarchies. In *Proceedings of High-Performance Graphics Conference*. 89–99.

A. Keller, L. Fascione, M. Fajardo, I. Georgiev, P. Christensen, J. Hanika, C. Eisenacher, and G. Nichols. 2015. The Path Tracing Revolution in the Movie Industry. In *ACM SIGGRAPH Courses (SIGGRAPH)*. Article 24, 24:1–24:7 pages.

Gregory Kramida and Amitabh Varshney. 2016. Resolving the Vergence-Accommodation Conflict in Head-Mounted Displays. *IEEE Trans. Visualization & Computer Graphics* 22, 7 (2016), 1912–1931.

Oliver Mattausch, Jirí Bittner, Alberto Jaspe, Enrico Gobbetti, Michael Wimmer, and Renato Pajarola. 2015. CHC+ RT: Coherent hierarchical culling for ray tracing. In *Computer Graphics Forum*, Vol. 34. 537–548.

Tomas Möller and Ben Trumbore. 1997. Fast, Minimum Storage Ray-triangle Intersection. *J. Graph. Tools* 2, 1 (1997), 21–28.

Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. 1994. A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl.* 14, 4 (1994), 23–32.

Ryan Overbeck, Ravi Ramamoorthi, and William R Mark. 2008. Large ray packets for real-time whitted ray tracing. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*. IEEE, 41–48.

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. *ACM Trans. Graph.* 29, 4, Article 66 (2010), 66:1–66:13 pages.

Suryakant Patidar and PJ Narayanan. 2008. Ray casting deformable models on the GPU. In *Computer Vision, Graphics & Image Processing, 2008. ICVGIP'08. Sixth Indian Conference on*. IEEE, 481–488.

Anjul Patney, Marco Salvi, Joohwan Kim, Anton Kaplanyan, Chris Wyman, Nir Benty, David Luebke, and Aaron Lefohn. 2016. Towards Foveated Rendering for Gaze-tracked Virtual Reality. *ACM Trans. Graph.* 35, 6, Article 179 (Nov. 2016), 12 pages. https://doi.org/10.1145/2980179.2980246

Voicu Popescu, Paul Rosen, and Nicoletta Adamo-Villani. 2009. The Graph Camera. In *ACM SIGGRAPH Asia 2009 Papers (SIGGRAPH Asia '09)*. ACM, New York, NY, USA, Article 158, 8 pages. https://doi.org/10.1145/1661412.1618504

Alexander Reshetov, Alexei Soupikov, and Jim Hurley. 2005. Multi-level Ray Tracing Algorithm. In *ACM SIGGRAPH Papers (SIGGRAPH)*. 1176–1185.

David Roger, Ulf Assarsson, and Nicolas Holzschuch. 2007. Whitted ray-tracing for dynamic scenes using a ray-space hierarchy on the GPU. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*. 99–110.

Qi Sun, Fu-Chung Huang, Joohwan Kim, Li-Yi Wei, David Luebke, and Arie Kaufman. 2017. Perceptually-guided Foveation for Light Field Displays. *ACM Trans. Graph.* 36, 6, Article 192 (Nov. 2017), 13 pages. https://doi.org/10.1145/3130800.3130807

Robert Toth, Jim Nilsson, and Tomas Akenine-MÃűller. 2016. Comparison of Projection Methods for Rendering Virtual Reality. In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*.

Ingo Wald, Solomon Boulos, and Peter Shirley. 2007a. Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies. *ACM Trans. Graph.* 26, 1, Article 6 (2007).

Ingo Wald, William R. Mark, Johannes GÃijnther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley. 2007b. State of the Art in Ray Tracing Animated Scenes. In *Eurographics - State of the Art Reports*.

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Trans. Graph.* 33, 4, Article 143 (2014), 143:1–143:8 pages.

Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. 2007. Microfacet Models for Refraction Through Rough Surfaces. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques (EGSR)*. 195–206.

Turner Whitted. 1980. An Improved Illumination Model for Shaded Display. *Commun. ACM* 23, 6 (1980), 343–349.

Chris Wyman, Rama Hoetzlein, and Aaron Lefohn. 2015. Frustum-traced Raster Shadows: Revisiting Irregular Z-buffers. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games (i3D)*. 15–23.