Frustum-Traced Raster Shadows: Revisiting Irregular Z-Buffers

Chris Wyman* Rama Hoetzlein Aaron Lefohn
NVIDIA

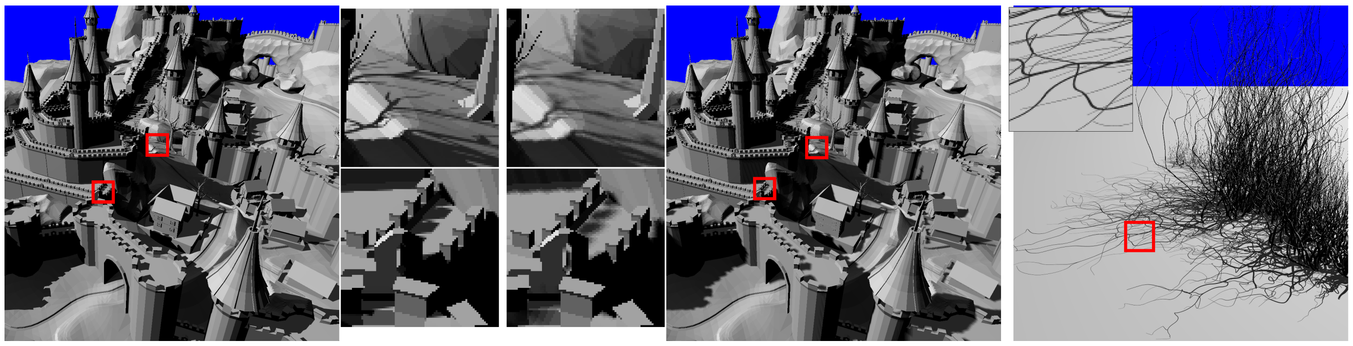


Figure 1: *(Left) Our 32 sample per pixel hard shadows in the 613k triangle Chalmers Citadel (16 ms at 1080p). (Center) Compared with a filtered* 8092^2 *shadow map, we avoid temporal and spatial aliasing and eliminate light leaking. (Right) 32 sample per pixel shadows from the intricate, 3.8M triangle tentacles model (38 ms at 1080p).*

Abstract

We present a real-time system that renders antialiased hard shadows using irregular z-buffers (IZBs). For subpixel accuracy, we use 32 samples per pixel at roughly twice the cost of a single sample. Our system remains interactive on a variety of game assets and CAD models while running at 1080p and 2160p and imposes no constraints on light, camera or geometry, allowing fully dynamic scenes without precomputation. Unlike shadow maps we introduce no spatial or temporal aliasing, smoothly animating even subpixel shadows from grass or wires.

Prior irregular z-buffer work relies heavily on GPU compute. Instead we leverage the graphics pipeline, including hardware conservative raster and early-z culling. We observe a duality between irregular z-buffer performance and shadow map quality; this allows common shadow map algorithms to reduce our cost. Compared to state-of-the-art ray tracers, we spawn similar numbers of triangle intersections per pixel yet completely rebuild our data structure in under 2 ms per frame.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: shadows, irregular z-buffer, frustum tracing

1 Introduction

While conceptually simple, real-time rendering of hard shadows remains an enormous challenge. Most modern applications use variants of shadow mapping [Williams 1978]. But due to discretization and sampling mismatches between eye- and light-space, robust and artifact-free results remain elusive even with high quality filtering [Annen et al. 2008] and cascades [Lauritzen et al. 2011].

We describe the evolution of our system, designed with a simple goal: interactive, sub-pixel accurate hard shadows at usable resolutions on content representative of modern workloads. We began without any preconceptions except that shadow maps induce aliasing, potentially a more challenging problem. While desirable, extendability to soft shadows was not a design goal; our primary goal was robust artifact-free shadows, with a secondary aim of speed.

To avoid aliasing, we only considered analytical shadow techniques that sample in eye space at or below the pixel level. This leaves three broad algorithmic classes: ray tracing [Whitted 1980], shadow volumes [Crow 1977], and irregular z-buffers [Johnson et al. 2005]. Fundamentally, all analytic approaches perform ray-triangle intersections; the key difference is how tests are spawned. Ray tracers query visibility along individual rays, irregular z-buffers rasterize in light space, and shadow volumes indirectly determine ray-triangle occlusion by testing shadow boundaries.

We pursued variants of irregular z-buffers (IZBs), which appeared the most natural fit for today's raster pipelines. Ray tracing requires additional acceleration structures, and shadow volumes either require object-space silhouette detection or introduce complex hierarchies [Sintorn et al. 2014] to accelerate brute force, per-triangle volumes.

Little research has explored efficient algorithms for IZBs. Initial work proposed major hardware changes [Johnson et al. 2005] and alternative data structures [Aila and Laine 2004]. Later work [Arvo 2007; Sintorn et al. 2008] showed hardware implementations using GPU computing capabilities rather than the graphics pipelines. Pan et al. [2009] improved quality, demonstrating antialiased shadow boundaries. To our knowledge, nobody has eliminated the key problems with irregular z-buffering: poor scalability and large performance variations between frames.

We demonstrate that IZB shadows scale to multi-million triangle models and HD resolutions yet remain interactive (see Figure 1). Initial prototypes required nearly 2 seconds for complex models like the tentacles. By simplifying and streamlining the data structure and leveraging existing strengths of the graphics pipeline we achieved two orders of magnitude better performance. A key in-

*e-mail:chris.wyman@acm.org

sight: we observe a duality between irregular z-buffer performance and shadow map quality. Regions that alias with shadow mapping have lower performance with IZBs. This allows decades of research improving quality and consistency of shadow maps to help improve irregular z-buffer performance.

Additional contributions of our system include:

- Demonstration of alias-free shadows on realistic content with consistently interactive performance. Cost is under 8 and 24 ms for game scenes with 1 and 32 samples per pixel (spp).
- An efficient IZB implementation relying on existing hardware in the graphics pipeline to improve culling beyond that available to implementations using only GPU compute.
- An efficient extension of single sample irregular z-buffers to render 32 spp shadows.
- An extensive performance evaluation with quantified gains for individual optimizations.

2 Why Irregular Z-Buffers?

Decades of shadow research provide developers many algorithmic choices [Eisemann et al. 2011; Woo and Poulin 2012]. Until recently, the choices in interactive contexts were limited to shadow volumes [Crow 1977] and shadow maps [Williams 1978].

Shadow volumes generate pixel-accurate shadows by constructing and testing the boundary of shadowed regions. This proves difficult to do robustly and renders invisible "shadow quads" that consume significant fill rate. With reductions to these problems (e.g., McGuire et al. [2003] and Lloyd et al. [2004]) some games shipped using shadow volumes, but most developers still avoid them.

Shadow maps are easily implemented and efficiently run on GPUs, but regular sampling of visibility causes spatial and temporal aliasing. Filtering [Reeves et al. 1987] and using statistical models [Donnelly and Lauritzen 2006; Annen et al. 2008] partially hide aliasing, but often introduce other artifacts. Distorting the light frustum [Stamminger and Drettakis 2002], adaptively refining [Fernando et al. 2001], or fitting multiple shadow maps [Lauritzen et al. 2011] improve sampling quality but can introduce overhead, reduce robustness, and still exhibit aliasing.

On the cusp of interactivity, ray tracing [Whitted 1980] may provide another alternative. Hardware acceleration improves performance [Lee et al. 2013], but adoption rates are unclear. Ray query costs strongly depend on existence of quality acceleration structures [Aila et al. 2013], which remain expensive to build. Concurrent work [Mittring 2014] suggests existing GPUs may suffice for game-quality ray tracing, though their grid-of-lists structure resembles IZBs more than standard bounding volume hierarchies.

Irregular z-buffers [Johnson et al. 2005] provide another option within existing graphics pipelines. Shadow maps use a light-space z-buffer; IZBs instead use a light-space A-buffer [Carpenter 1984], with each texel storing all pixels potentially occluded by geometry in that texel. While a grid-of-lists structure increases complexity over shadow maps, today's GPUs construct and traverse linked-lists efficiently [Yang et al. 2010]. Key problems with IZBs include poor scalability and high performance variability. Prior GPU implementations [Sintorn et al. 2008; Pan et al. 2009] typically rendered at 512^2 with models under 100k triangles; performance scaled linearly with triangle and pixel counts. Our initial IZB prototype exhibited 100:1 performance variations between certain frames during even basic interactions.

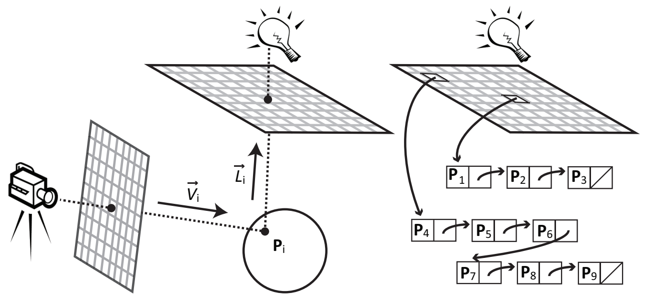


Figure 2: *(Left) A familiar mapping between view vector $\vec{V}_i$, intersection point $\mathbf{P}_i$, and light direction $\vec{L}_i$. Shadow maps discretize light-space, so shadow queries return the nearest neighbor rather than the true visibility along $\vec{L}_i$. (Right) Irregular z-buffer shadows store all sample points $\mathbf{P}_i$, allowing exact shadows. Rather than storing a single depth, as in shadow maps, IZBs store all points that fall in a texel as a linked list.*

However, their problems lie entirely in the performance domain. While Johnson et al. [2005] explored performance characteristics on their proposed hardware, IZB's characteristics on modern GPUs are largely unexplored. By carefully identifying and removing key bottlenecks, we designed an efficient implementation achievable today. The 2.9 million triangle hairball (in Figure 7) required 12.6 seconds in Aila and Laine's [2004] work, 1.5 seconds in our first prototype, and 13.8 milliseconds today.

3 Irregular Z-Buffer Review

Before describing our system, we briefly review irregular z-buffer shadows. IZBs aim to avoid shadow map aliasing from mismatches between eye- and light-space sampling locations. Shadow maps use a regular grid of samples in both eye- and light-space, but finding a robust bijection between samples in these spaces remains unsolved. By allowing light-space samples to occur irregularly, IZB shadows easily pair samples and eliminate aliasing.

By construction, an IZB bijectively maps each pixel to one sample in light-space; the pixel represents ray $\vec{V}_i$ hitting geometry at $\mathbf{P}_i$ and a corresponding light sample represents ray $\vec{L}_i$ from $\mathbf{P}_i$ to the light (see Figure 2). Shadow map queries along $\vec{L}_i$ return the nearest neighbor sample on the texel grid; irregular z-buffers store all samples, generating a unique visibility for each pixel.

In theory, constructing an irregular z-buffer shadow map works identically to regular shadow maps: one "rasterizes" occluders over the irregular set of light rays $\vec{L}_i$, finding the closest triangle along each $\vec{L}_i$. If the depth of the closest triangle lies between the light and $\mathbf{P}_i$ we know that sample is shadowed.

Since modern GPUs rasterize only over regular samples, we need to store our irregular samples in a grid. A grid-of-lists structure achieves this. The irregular z-buffer is a grid of light-space head pointers, each pointing to a linked list containing irregular samples falling within the grid cell. Intuitively, this is identical to a shadow map except texels stores a pointer rather than a depth. Since samples can lie anywhere within a texel, conservative rasterization is required; triangles must test samples for occlusion if they cover any portion of a texel (not just the center, as in traditional rasterization).

Rasterizing over irregular samples requires knowing where they occur; IZBs require a prepass to identify sample locations. Game engines commonly use an eye-space z-prepass to reduce overshading.

This prepass provides exactly the needed data: locations of visible pixels requiring shadow queries. To identify IZB samples we run a compute pass over this z-buffer, transforming pixels into light-space (via the shadow map transformation), and inserting them into their corresponding light-space lists (see Figure 2).

Pseudocode describing this process follows:

```
High Level Pseudocode: Irregular Z-Buffer Shadows

// Step 1: Identify locations we need to shadow
$\mathbf{G}(x, y)$ ← RenderGBufferFromEye()

// Step 2: Add these pixels into the light-space data structure (our IZB)
for pixel $p \in \mathbf{G}(x, y)$ do
    lsTexel$_p$ ← ShadowMapTransform[ GetEyeSpacePos( $p$ ) ]
    izbNode$_p$ ← CreateIZBNode[ $p$ ]
    AddNodeToLightSpaceList[ lsTexel$_p$, izbNode$_p$ ]
end for

// Step 3: Determine shadows; test each triangle with pixels in lists it covers
for triangle $t \in$ SceneTriangles do
    for fragment $f \in$ ConservativelyRasterizeInLightSpace( $t$ ) do
        lsTexel$_f$ ← FragmentLocationInRasterGrid[ $f$ ]
        TraversePixelListFromStep2TestingIfTriangleShadows[ lsTexel$_f$ ]
    end for
end for
```

Initially this seems complex, but is a relatively simple modification to shadow mapping, essentially swapping the order of light-space rasterization and sample projection:

```
High Level Pseudocode: Shadow Maps

// Step 1: Render shadow map in light-space
for triangle $t \in$ SceneTriangles do
    for fragment $f \in$ RasterizeInLightSpace( $t$ ) do
        lsTexel$_f$ ← FragmentLocationInRasterGrid[ $f$ ]
        if depth$_f$ < $\mathbb{Z}$( lsTexel$_f$ ) then $\mathbb{Z}$( lsTexel$_f$ ) ← depth$_f$
    end for
end for

// Step 2: Query shadow map for each pixel
for pixel $p \in$ FinalRender do
    lsTexel$_p$ ← ShadowMapTransform[ GetEyeSpacePos( $p$ ) ]
    isShadowed ← ( DistanceToLight( $p$ ) > $\mathbb{Z}$( lsTexel$_p$ ) ? true : false )
end for
```

3.1 Performance Considerations

As with ray tracing, the key unit of work is ray-triangle intersections. These are spawned as a triangle fragment traverses its list of potentially occluded samples (in pseudocode, step 3); each sample represents a ray (from $\mathbf{P}_i$ along $\vec{L}_i$) that is tested for intersection with the rasterized triangle. Other steps have only small performance costs: a z-prepass typically already occurs, and simple irregular z-buffer creation requires under a millisecond.

Simplistically, the algorithmic complexity is $O(N)$ for N ray-triangle visibility tests. But N depends on triangle count and screen resolution; pixels create IZB nodes, and triangles generate light-space fragments that traverse lists of pixels. So $N = t_f \langle l_{izb} \rangle$ for t_f the number of light-space triangle fragments and $\langle l_{izb} \rangle$ the average IZB list length traversed by each fragment.

This means performance depends on total ray-triangle tests, number of light-space fragments, and the average length of IZB node lists. Additionally, traversing IZB lists causes GPU underutilization if list lengths vary significantly between threads, so reducing variance of l_{izb} improves performance for GPU implementations. These factors interact in non-intuitive ways. Increasing light-space grid resolution reduces average list length by distributing samples over larger numbers of texels but also increases the number of light-space fragments generated by rasterization. At first glance the utility of such a change is unclear, and prior work rarely explored such tradeoffs.

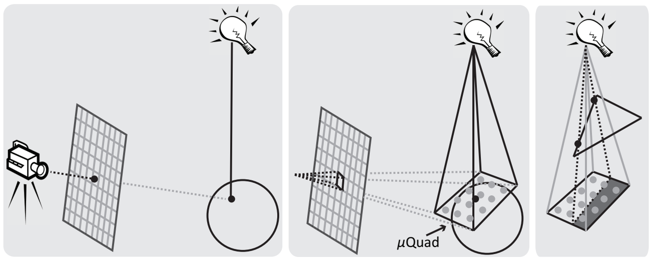


Figure 3: *(Left) A standard shadow ray query. (Center) Creating fragment shadow frusta (at black point): project pixel footprint to the fragment tangent plane; this "micro-quad" (µQuad) becomes the shadow frustum base. (Right) Intersection tests project each triangle edge to the tangent plane (i.e., identifying the shadow quad intersection), and use the projected edge to index a lookup table providing visibility for each sample. Visibility sample locations (gray points) are developer specified during LUT construction.*

4 Antialiasing via Frustum-Triangle Tests

Like ray tracing, IZB shadows provide pixel accurate shadows. Naive extensions achieve subpixel accuracy via multiple sampling (e.g., shooting additional rays or using multiple IZB samples per pixel), but increase cost linearly with sample count.

Alternatively a beam tracing approach [Heckbert and Hanrahan 1984] can provide analytic subpixel visibility. Interactive work often replaces beams with packet tracing [Boulos et al. 2007], tiled rasterization, or raster stamps to preserve spatial coherence while using discrete samples. Numerous soft shadow algorithms use these methods, including for ray traced shadows [Overbeck et al. 2007], bitmask soft shadows [Schwarz and Stamminger 2007], and even IZB-based soft shadows [Sintorn et al. 2008].

To achieve antialised shadows we flip these beams around, essentially tracing a frustum from the point light back to the geometry (see Figure 3). This is similar to Pan et al.'s [2009] approach, though we directly compute intersections in world space rather than projecting back to the image plane.

We achieve antialised shadows by replacing the ray-triangle intersections in IZB shadows with frustum-triangle intersections. Efficient frustum intersection occurs per pixel as follows:

- Construct a μQuad representing the pixel projection onto the tangent plane (see Figure 3).
- Treat each edge of the occluder triangle independently. An edge plus the light define a shadow quad. The three shadow quads plus the triangle plane bound its shadow volume.
- Project each shadow quad to the tangent plane, determining which half-plane is shadowed. Use projection to lookup a 32-sample visibility bitmask from a lookup table. Combine the edge results with a binary AND to create a visibility bitmask representing which pixel subsamples this triangle occludes.

As this intersection represents the inner loop, tight optimization is vital for good performance. We provide an example GLSL implementation as supplementary material.

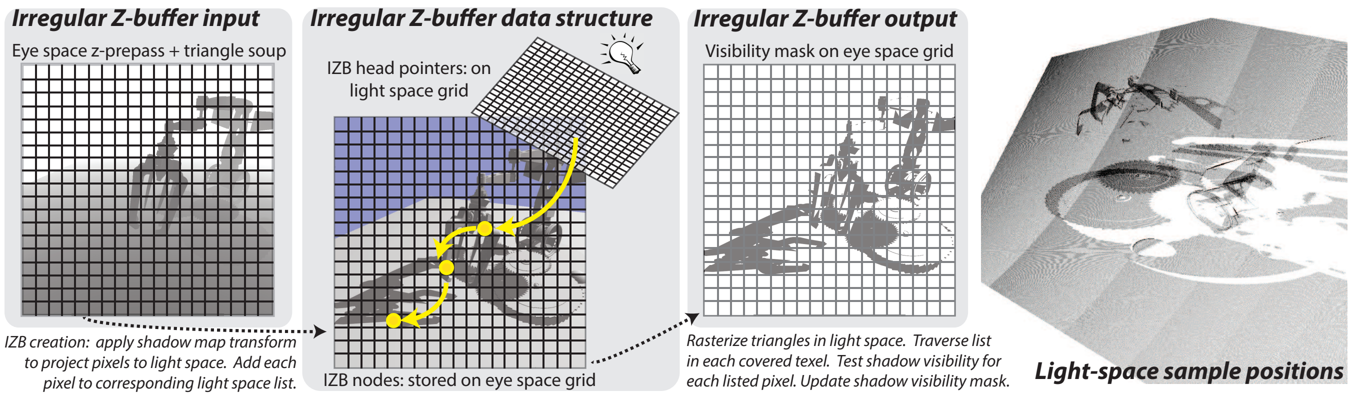

Irregular Z-buffer input
Eye space z-prepass + triangle soup
Irregular Z-buffer data structure
IZB head pointers: on light space grid
Irregular Z-buffer output
Visibility mask on eye space grid
IZB creation: apply shadow map transform to project pixels to light space. Add each pixel to corresponding light space list.
IZB nodes: stored on eye space grid
Rasterize triangles in light space. Traverse list in each covered texel. Test shadow visibility for each listed pixel. Update shadow visibility mask.
Light-space sample positions

Figure 4: *An overview of our irregular z-buffer implementation. We require an eye-space z-buffer plus triangle geometry as input; each pixel location is transformed to light-space and added to the projected texel's linked list. Triangles are rasterized over this light-space grid; each fragment traverses its list and tests listed pixels for occlusion. When tests identify occlusion, the pixels' visibility mask is updated. The final render pass consumes these visibility masks to correctly shadow pixels. (Right) An example visualization of irregular light-space samples.*

5 System Overview

As discussed in Section 3, irregular z-buffer shadows have three distinct phases: creating the IZB data structure, spawning triangle occlusion tests via light-space rasterization, and rendering a final shadowed image. Figure 4 visually outlines this process, describing inputs, outputs, and the light-space data structure. We implement these steps with six passes: an eye-space z-prepass, bounding the scene's visible regions, creating the irregular z-buffer, a light-space culling prepass, spawning triangle visibility tests, and the final render. These are outlined in greater detail below.

5.1 Eye-Space Z-Prepass

Create an irregular z-buffer requires knowing which samples to add. Modern rendering engines often use a z-only prepass to facilitate culling; this pass provides the required information. Our implementation simultaneously creates a G-buffer, used during our deferred final render phase.

Single sample shadows only require fragment depths. To perform frustum intersection for antialised shadows, we add three floats to the G-buffer describing the μQuad projection onto the tangent plane; this accelerates visibility tests, but could be recomputed from the fragment normal if G-buffer space is tight.

5.2 Scene Bounds

As with shadow maps, a priori knowing the correct settings for the light frustum is challenging. To avoid poorly bounding the scene, we recompute the light's projection matrix each frame to tightly bound geometry visible in the z-prepass. We use a persistent thread compute shader over the z-buffer from Section 5.1.

5.3 Creating the Irregular Z-Buffer

We walk through the eye-space z-buffer, transforming each pixel into light-space and inserting it into the appropriate texel's linked list, as described in Section 3.

Using multiple visibility samples per pixel may also require multiple IZB nodes per pixel, in theory up to one node per visibility sample. Unlike a point query, our μQuad may project to multiple light-space texels; triangles touching any of these texels could shadow the pixel. Section 6.1 introduces an approximation using at most eight (and averaging two) IZB nodes per pixel with little quality impact.

5.4 Light-Space Culling Prepass

In Section 5.5 we spawn visibility tests via conservative rasterization. The GPU's early-z hardware can accelerate this process by culling triangle fragments covering empty pixel lists or falling behind the furthest list node. Intuitively, this provides frustum culling based on actual pixel geometry.

To use early-z hardware, we need a light-space z-buffer. This could be generated as a side effect in Section 5.3. But as that pass runs in eye-space, standard raster operations cannot output light-space depth. Given the GPU's opaque depth format, we had difficulty using global memory writes to create a z-buffer usable by the early-z hardware. Our prepass essentially creates a stencil: setting depth to 0 in texels with empty lists and the distance to the furthest IZB node elsewhere. Except in our most trivial scenes, using hardware z-cull provides a substantial speedup (between 30 and 50%).

5.5 Spawning Triangle Visibility Tests

Spawning and performing visibility tests represents our system's major cost. We conservatively rasterize triangles over a light-space grid (shown in Figure 4). Each texel stores a list of pixels potentially occluded by triangles overlapping it. Conservative rasterization is required, as pixels may lie anywhere within the volume represented by a texel; triangles partially covering a texel may occlude an arbitrary subset of its list.

Each fragment traverses the entire texel list. During traversal we load each listed pixel, perform a visibility test, and atomically OR the result into the pixel's visibility mask.

A key bottleneck stems from thread divergence at this step; since lists have variable length, some threads wait on adjacent threads. For naive implementations, this wait can be extreme; in certain views we observed 1000:1 variations between adjacent list lengths.

5.6 Final Render

Section 5.5 performs shadow tests and stores results in a per-pixel visibility mask. Our final render pass loads from the G-buffer (Section 5.1) and this mask and performs Phong shading modulated by the shadow mask.

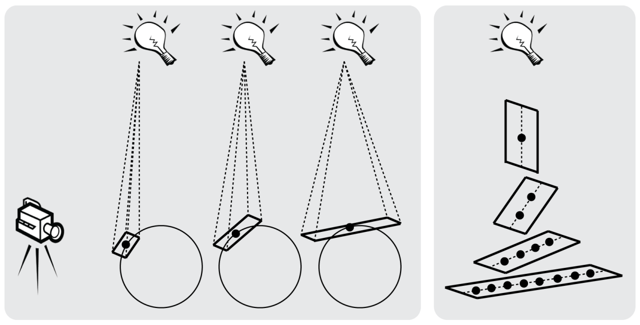


Figure 5: *(Left) As a fragment's tangent plane changes orientation, μQuads elongate along only one axis (the other dimension depends on screen resolution). (Right) This suggests sampling them one dimensionally; we add from one to eight samples to the IZB depending on orientation.*

6 Implementation Details

Section 5 provides a high-level description of our system, but key aspects deserve greater explanation. First, we discuss how multi-sample shadows change our data structure. Then we explore efficient layouts for the irregular z-buffer data structure and outline some smaller optimizations that significantly impact performance.

6.1 Simplifying IZBs for Multi-Sample Shadows

Moving from 1 to 32 spp shadows complicates our algorithm. The key point is light-space rasterization needs to spawn frustum-triangle tests (described in Section 4) for any triangle occluding a pixel. Turning that around, a μQuad projects to a variable number of light-space texels; its pixel must be added to those texels' lists. One expensive approach rasterizes μQuads in light-space during IZB construction. Another method adds all 32 per pixel visibility samples to the IZB. An optimization can skip duplicate insertions within a μQuad. We initially prototyped this sample-based insertion, though it often adds 8 or more IZB nodes per pixel.

Observing μQuad geometry suggests a simple approximation to improve performance. μQuads enlarge along eye-space silhouettes as $\vec{N} \cdot \vec{V} \to 0$ (see Figure 5). By construction they only elongate in depth, remaining constant width regardless of surface orientation. We treat them as 1-dimensional samples. As μQuads elongate we insert IZB samples along the pixel's view ray (the μQuad center), using between 1 and 8 samples.

This creates an approximate irregular z-buffer. As μQuads enlarge we miss inserting some nodes, introducing light leaks for small distant occluders falling between samples (i.e., that fail to spawn needed frustum-triangle tests). Examples of light leaking are provided in our supplementary materials. To reduce these missed tests, triangles can be over-conservatively rasterized. Our implementation rasterizes light-space triangles with a 1 texel dilation (rather than a half texel in typical conservative rasterization), ensuring triangles touch more sample locations.

Given the algorithmic complexity in Section 3.1, this tradeoff initially seems questionable. Reducing the number of IZB nodes directly decreases average list length $\langle l_{izb} \rangle$, but enlarging conservative raster dilation increases triangle fragment count t_f by a smaller amount. Our approximate insertion averages two IZB nodes per pixel compared to eight with the exact approach, for a $4\times$ reduction in $\langle l_{izb} \rangle$; increasing triangle dilation from 0.5 to 1.0 pixels only increases t_f 6–40%, giving a large net speedup.

6.2 Data Structure and Memory Layout

In any complex data structure, it is worth considering the best layout for efficient construction and traversal. We initially prototyped a simple 2D grid of linked lists, allocating nodes from a global node pool (see Figure 2). Each list node contained two entries: a next pointer and a G-buffer index (to fetch pixel data). But this structure needs two synchronizations per insertion: a global atomic to find a free node and a per texel atomic to update the head pointer.

We tried compacting our lists, similar to Sintorn et al.'s [2008] variable length arrays. This reduces memory consumption by eliminating next pointers. But our experiments consistently showed a performance drop of exactly the compaction cost. We also tried sorting lists by distance to the light; this reduced performance by roughly the sort cost. This suggests either traversal order has little impact or we sufficiently load the GPU to hide traversal latency.

But cutting node size is appealing, so we instead eliminated the G-buffer index. By preallocating a screen-space grid of nodes, the node address implicitly provides its G-buffer index (see Figure 4); each node just contains a next pointer. Besides cutting list size in half, this provides other advantages. Inserting list nodes no longer requires global synchronization, as we directly map pixels to node IDs. This roughly halves build cost and removes a memory indirection during list traversal; as addresses directly map to pixels, next pointers provide information to load the next node and G-buffer data simultaneously.

For 32 spp shadows we continue using this method, storing up to 8 nodes per pixel. Since we allocate 8 nodes per pixel in advance, this wastes some memory (roughly doubling the memory of a linked-list structure). But we felt the improved performance was worth this moderate memory increase.

6.3 Light-Space Texture Resolution

As in shadow maps, selecting the correct light-space resolution is important. Unlike shadow maps, resolution does not impact quality but it does affect performance.

Consider IZB's $O(t_f \langle l_{izb} \rangle)$ complexity. Halving resolution grows average list lengths $4\times$ while lowering triangle fragments $4\times$, suggesting resolution minimally impacts performance. But conservative raster also generates more fragments, and this effect grows for small triangles and low resolutions. Larger resolutions increase memory consumption of the head pointer texture, though the number of IZB nodes is largely invariant with light-space resolution.

Considering our goals, we want neither high $\langle l_{izb} \rangle$ nor lots of fragments testing empty lists. This suggests closely matching light-space and image resolutions. Early tests showed a 2048^2 head texture worked well for all scenes at 1080p, though later experiments (Figure 9) suggest the sweet spot varies from 1400^2 to 2500^2.

6.4 Matching Sampling Rates: Cascades

Ideally, we would match eye- and light-space sampling 1:1 so each triangle fragment spawns exactly one visibility test. Shadow map research has explored this sampling problem for decades, suggesting various methods to approach our ideal sampling: perspective [Stamminger and Drettakis 2002], logarithmic [Lloyd et al. 2008], cascaded [Lloyd et al. 2006], and sample distribution shadow maps (SDSMs) [Lauritzen et al. 2011] all improve sampling. For our purposes, perspective shadow maps have hard-to-control singularities and logarithmic approaches require non-linear rasterization. However, cascades give better sampling with manageable overhead and SDSMs provide automatic partitioning.

Adding cascades is straightforward, though it affects multiple steps of our algorithm. Our extents pass not only bounds the entire scene, but also splits it into multiple cascades with Lauritzen et al.'s [2011] logarithmic partitioning and individually bounds each cascade.

We generate a separate IZB for each cascade. Creation of cascaded IZBs for single sample shadows easily occurs in parallel (cascades contain unique pixels). Cascades for multisample shadows must overlap slightly to avoid light leaks along boundaries; we serially generate these IZBs prior to use but expect a careful implementation could insert samples in multiple cascades in parallel.

Light-space rasterization needs to occur over each IZB to accumulate full visibility. Currently we naively use one render pass per cascade. Culling geometry separately for each light frustum or using a single render pass to route primitives to the appropriate cascade would both improve performance.

Except for complex models that naturally fit in one frustum (e.g., the hairball), cascades' significant reduction in GPU divergence pays for the overhead of rasterizing geometry multiple times. Divergence data with and without cascades is provided as supplementary material.

6.5 $\vec{N} \cdot \vec{L}$ Culling

Standard lighting models trivially shadow pixels with $\vec{N} \cdot \vec{L} \leq 0$. Not adding these pixels to the IZB reduces $\langle l_{izb} \rangle$. Since the $\vec{N} \cdot \vec{L}$ term already shadows these pixels, their visibility mask can be left fully lit. This avoids a common problem along light silhouettes where geometric and shading normals provide different shadow terms. This culling consistently improves performance 10-15%.

6.6 Early Out: IZB Node Removal

While testing visibility, a pixel often becomes fully occluded. Triangles rasterized later in the frame can have no additional impact, so spawning additional frustum-triangle tests is wasteful. This suggests removing occluded pixels from IZB lists (analogous to ray tracing with "any hit" rays).

Importantly, node removal requires no atomic operations. Race conditions can occur, but at worst cause extra visibility tests on already-occluded pixels (after which we retry removal). Node removal provides a 10-15% performance win despite additional logic and memory operations in the inner traversal loop.

6.7 Memory Synchronization: Unchanged Masks

Visibility tests are inexpensive relative to a GPU's maximum theoretical performance. Memory latency, throughput, and synchronization thus prove key bottlenecks. One synchronization point is a pixel's visibility mask; multiple triangles testing visibility at the same pixel need to atomically combine results to avoid races. To reduce contention, mask updates should only occur if a triangle *changes* the existing visibility. This requires loading the prior mask, but the avoided contention provides up to a 14% speed boost.

6.8 Latency Hiding: Software Pipelining

When traversing a list of IZB nodes, the inner loop: loads the next node, loads its pixel data, performs a visibility test, and updates the visibility mask. This forms a dependency chain, with long memory latencies between tasks. The GPU may be unable to hide all these latencies. Fortunately we can apply software pipelining, loading the next node and computing G-buffer coordinates in the prior loop iteration. This hides latency and improves speed 5-15%. Pushing visibility mask updates to the next iteration may be a further win.

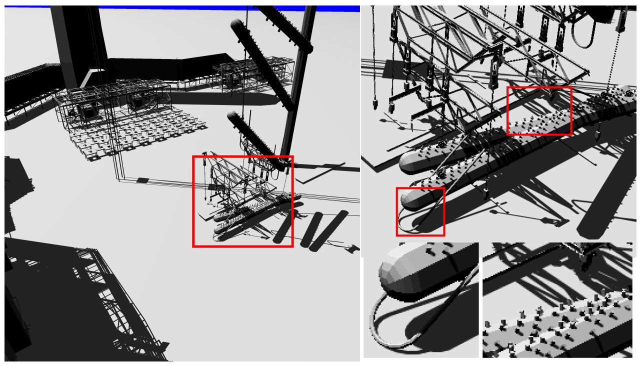


Figure 6: *The 12.3 million triangle UNC Powerplant model running at* 3840×2160 *in under 140 ms for 32 sample per pixel shadows. We zoom twice to highlight the shadow quality.*

7 Results and Discussion

Our system uses OpenGL 4.4 and has been tested on various NVIDIA GPUs and an AMD Radeon 290X. Timings come from a GeForce GTX 980. Some results use two extra NVIDIA extensions: *NV_conservative_raster* and *NV_geometry_shader_pass through*. These results are identified as "GM204 optimized."

Figures 1, 6 and 7 show our test scenes. Table 1 gives performance breakdowns using consistent rendering parameters; these do not provide optimal performance in all scenes, but provide reasonably high performance across our test suite.

Table 1 shows that bounding scene extents, priming the z-buffer for culling, and final render all have consistent costs. Computing scene extents requires a pass over all G-buffer samples running Lauritzen et al.'s [2011] logarithmic partitioning to delineate cascades; this cost depends on screen resolution. Section 5.4 notes priming light-space culling should be unnecessary; however it just requires a blit to our light-space z-buffer. Our final render loads the G-buffer and visibility mask and applies a simple shading model.

IZB creation costs vary depending on how many nodes are added to our linked lists and how much atomic contention occurs. Incoherent geometry (hairball and tentacles) and complex tessellated models (GeeBee and powerplant) exhibit slightly lower performance. For 1 sample shadows, we add at most one IZB node per pixel. 32 sample shadows insert roughly twice as many nodes (see Section 6.1), increasing atomic contention. Still, IZB creation costs remain remarkably consistent across all our test scenes.

Our major cost is light-space rasterization (Section 5.5), which tests visibility via frustum- or ray-triangle intersections. Theoretically, cost varies linearly with additional frustum-triangle tests, though Table 1 shows this correspondence breaks in practice as GPU utilization varies between scenes. Tests of peak throughput show our GeForce GTX 980 achieves up to 7 million frustum tests per millisecond (Table 2). While we fall short of this ideal, geometry representative of game assets achieves up to 30% this limit. Given the extra memory latency and logic overhead in our list traversal, we feel this is quite good. More complex models achieve a much lower throughput; we expect this is due to reduced culling efficiency for small triangles (full pixel occlusions occur less frequently and triangles grow by a larger percent with conservative raster).

For single sample shadows, our GM204-optimized code (using

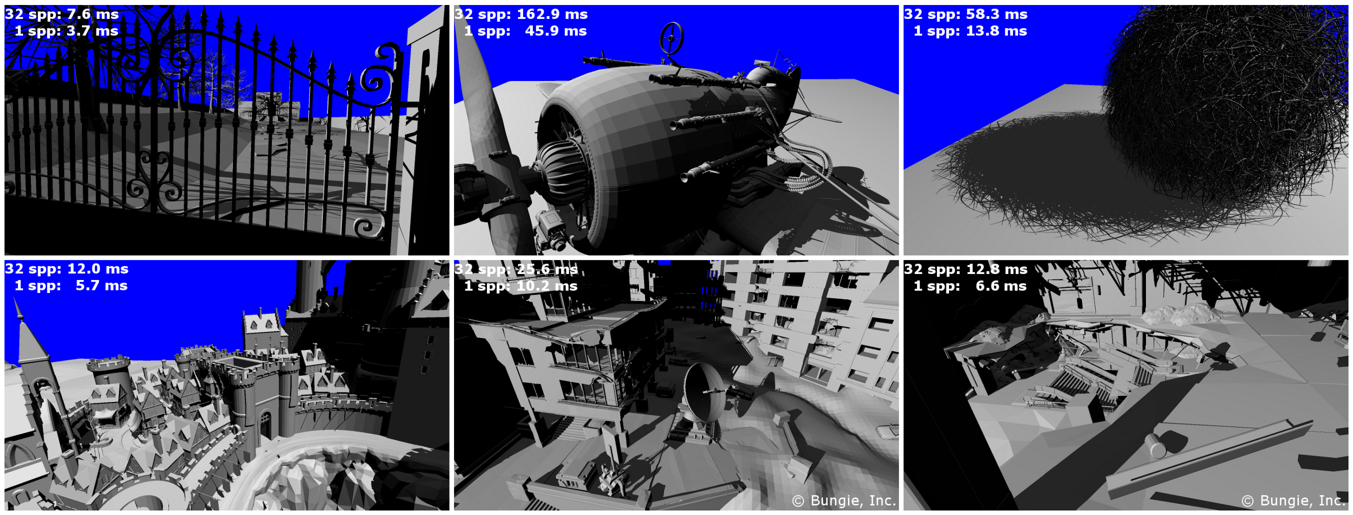


Figure 7: *Scenes used to test our system include: Chalmers Villa, GeeBee Plane, Hairball, Epic Citadel, Bungie Terrain, Bungie Building. Timings give total frame time using four* 2048^2 *cascades, rendered at* 1920×1080*. Some scenes courtesy Epic Games and Bungie, Inc.*

Scene and Triangle Count (sorted by complxeity)	**Individual Step Times for 32 spp (1 spp), in milliseconds**						Total Render Time (msec)	Light-Space Tri Frags, millions	Tri-Pixel Vis Tests, millions	**Avg. Tests Per**	
	G-Buf	Z-Extent Bounds	Create IZB	Z-Cull Setup	Raster Tris Over IZB	Final Render				Tri Frag	Pixel
Chalmers Villa, 89 k	0.7 (0.4)	0.8 (0.7)	1.7 (0.6)	1.1 (1.1)	2.9 (0.8)	0.3 (0.2)	7.6 (3.7)	4.6 (1.3)	6.8 (1.5)	1.5 (1.2)	4.1 (0.9)
Bungie Building, 255 k	1.2 (0.6)	0.8 (0.8)	1.7 (0.6)	1.1 (1.1)	7.5 (3.3)	0.4 (0.2)	12.8 (6.6)	19.3 (9.7)	19.2 (8.8)	1.0 (0.9)	9.3 (4.3)
Epic Citadel, 374 k	1.0 (0.5)	0.7 (0.7)	1.6 (0.6)	1.1 (1.1)	7.3 (2.6)	0.3 (0.2)	12.0 (5.7)	10.1 (5.8)	13.8 (7.1)	1.4 (1.2)	8.6 (4.4)
Chalmers Citadel, 613 k	1.0 (0.6)	0.7 (0.7)	1.6 (0.6)	1.1 (1.1)	11.8 (4.0)	0.4 (0.2)	16.6 (7.2)	14.1 (7.4)	20.2 (8.8)	1.4 (1.2)	11.2 (4.9)
Bungie Terrain, 1.5 M	1.9 (1.0)	0.8 (0.8)	1.9 (0.7)	1.1 (1.1)	19.5 (6.4)	0.4 (0.2)	25.6 (10.2)	39.7 (16.0)	38.8 (14.8)	1.0 (0.9)	18.7 (7.1)
Hairball, 2.9 M	2.6 (1.5)	0.7 (0.7)	1.9 (0.7)	1.1 (1.1)	51.7 (9.6)	0.3 (0.2)	58.3 (13.8)	77.2 (16.5)	24.2 (3.9)	0.3 (0.2)	15.0 (2.4)
Tentacles, 3.8 M	1.8 (1.2)	0.7 (0.7)	1.7 (0.6)	1.1 (1.1)	55.0 (14.6)	0.3 (0.2)	70.5 (18.4)	7.1 (1.6)	7.9 (1.5)	1.1 (0.9)	6.8 (1.4)
GeeBee Plane, 11.7 M	6.3 (5.1)	0.7 (0.7)	1.9 (0.6)	1.1 (1.1)	152.6 (38.1)	0.3 (0.2)	162.9 (45.9)	51.5 (11.9)	58.7 (9.4)	1.1 (0.8)	38.4 (6.1)
UNC Powerplant, 12.3 M	3.8 (3.3)	0.8 (0.8)	2.1 (0.7)	1.1 (1.1)	67.4 (14.0)	0.4 (0.2)	75.6 (20.1)	11.4 (4.7)	20.4 (4.6)	1.8 (1.0)	10.3 (2.3)

Table 1: *Performance breakdown for individual algorithmic steps for 32 spp (and 1 spp) IZB hard shadows. To allow comparison all scenes use identical settings:* 1920×1080 *resolution, 4 cascades each containing* 2048^2 *lists of IZB nodes, and using all our optimizations. Single sample shadows use our GM204-optimized code path. We provide explicit measures of the work done, including light-space fragment counts (after culling) and total frustum-triangle (or ray-triangle) tests performed. Based on those counts, we provide averages for visibility tests performed per light-space fragment and per shaded pixel in the final rendering.*

Scene	**Speed of Light Tests at 32 spp**			Our Best Perf (Tests / ms)
	Tri-Frustum Tests	Speed of Light (ms)	Tests / ms	
Chalmers Villa	6.8 M	1.4	4.9 M	2.3 M
Bungie Building	20.6 M	6.1	3.4 M	2.6 M
Epic Citadel	19.9 M	2.9	6.9 M	1.9 M
Chalmers Citadel	31.8 M	4.8	6.6 M	1.7 M
Bungie Terrain	40.7 M	9.6	4.2 M	2.0 M
Hairball	83.0 M	29.8	2.8 M	0.5 M
Tentacles	9.6 M	2.7	3.6 M	0.3 M
UNC Powerplant	37.6 M	9.5	4.0 M	0.5 M
Average			**4.6 M**	**1.5 M**

Table 2: *Our "speed-of-light" tests. A prepass computes and stores a list of all necessary frustum-triangle visibility tests. We then launch a compute pass performing one test per thread, fully utilizing the GPU without divergence. For representative game scenes, our system achieves throughput up to 75% the speed-of-light.*

hardware conservative raster and NVIDIA's fast geometry shader) provides up to a 3.5× win over an equivalent software path. For example, our software light-space rasterization takes 7.2, 52, and 115 ms for the Chalmers Citadel, Tentacles, and GeeBee models. Hardware conservative rasterization using an extra half-pixel dila-

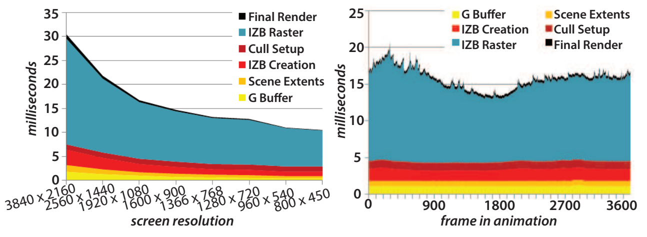


Figure 8: *Performance variation in the Chalmers Citadel (left) with varying render resolution and (right) over a 3600 frame animation.*

tion would enable these optimizations for our 32-sample shadows, and we expect a similar performance benefit. Roughly half this performance win comes from hardware conservative raster, which generates fewer fragments than our software implementation (using [Hasselgren et al. 2005]).

Interestingly, Figure 8 shows non-linear performance scaling with screen resolution. Scaling from 1080p to 2160p roughly reduces performance by 2× rather than the expected 4×. This likely stems from our culling, specifically the node removal in Section 6.6, which reduces the penalty for longer linked lists. As fully occluded

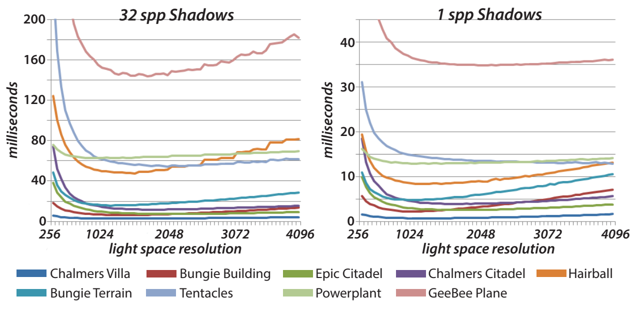


Figure 9: *Performance variation for (left) 32 spp and (right) 1 spp shadows with varying cascade resolutions. Timings represent just the light-space rasterization step, using 4 cascades with resolution between* 256^2 *and* 4096^2.

pixels are quickly eliminated, increasing resolution along epipolar lines affects performance sublinearly.

Figure 9 tracks performance under varying light-space resolution. Larger resolutions distribute IZB nodes among more lists (reducing average list length), but also cause rasterization to emit more fragments. For single sample shadows, these effects largely cancel out and performance remains consistent for sufficiently large cascades. For multisample shadows, higher resolutions cause μQuads to project to more texels (requiring additional IZB nodes); this distinctly lowers performance at higher resolutions. The sweet spot ranges from 1400^2 to 2500^2, and in moderately complex scenes performance varies only slightly through this range.

Consider shadow map aliasing. A shadow map texel aliases when it projects to multiple pixels. By construction, an irregular z-buffer lists these same pixels in a single texel. As sampling mismatches increase, shadow maps alias further and IZB lists lengthen. For standard shadow map problem cases, e.g., the shadowed ground plane at sunset, our lists can become quite large. Increasing resolution and better matching via warping or cascades addresses this problem by splitting these long lists.

One can view an IZB as a ray-triangle acceleration structure, similar to a ray tracer's bounding volume hierarchy. Given these intersections are the key cost, we asked how many tests our system performs compared to a ray tracer. We implemented our frustum-tracing algorithm in a beam tracer using state-of-the-art acceleration structures [Aila et al. 2013] and compared the number of frustum-triangle tests required. Figure 10 compares these numbers with our results from Table 1. Our system tested $0.3\times$ to $2.6\times$ as many triangles per pixel as the ray tracer, surprisingly well given our IZB construction costs only 2 ms. For validation, we compare quality with a multisample OptiX ray tracer and show costs over an animated sequence.

7.1 Specific Comparisons to Prior Work

Beyond ray tracers, our work shares similarities with other techniques. Lecocq et al. [2014] and Sen et al. [2003] augment a shadow map with geometry, which can give subpixel accurate shadows. But they fail in complex texels when multiple triangles or silhouettes intersect. Irregular z-buffers seamlessly handle these cases.

Despite the name, we share similarities with per-triangle shadow volumes [Sintorn et al. 2011; Sintorn et al. 2014]; our light-space rasterization and traversal essentially spawns per-triangle shadow volumes, but Sintorn et al.'s data structure more closely resembles Aila and Laine's [2004] space partitioning. Because they use GPU compute, they implement a complex space partitioning to achieve culling similar to our hardware-accelerated z-cull.

Key differences from prior irregular z-buffers include: our use of shadow map techniques like cascades to improve performance, culling via hardware z-cull, IZB node removals to avoid redundant tests on shadowed pixels, and an efficient way to build and traverse an IZB for multisample shadows.

8 Conclusions and Future Work

We introduced a system designed to render subpixel accurate shadows. This system runs in real-time for modern game content and interactively for more complex CAD models, in both cases at HD or higher resolutions. Adding multiple samples for subpixel visibility only costs slightly more than a single sample.

While we designed primarily for accuracy, we introduced one optimization for multisample shadows that can introduce light leaks when small triangles shadow pixels with $\vec{N} \cdot \vec{V}$ near 0. We believe a modified light space parameterization could eliminate this leaking, but developers needing quality could skip this optimization.

We hope our work spurs further exploration into real-time analytic shadow algorithms. While we feel our performance impressive, a number of improvements still remain: a more programmable depth test could enable better culling during light space rasterization, applying conservative rasterization to all triangles seems wasteful (but identifying silhouettes robustly is challenging), and extensions to soft shadows increase culling challenges.

Acknowledgments

Thanks to Bungie, Epic Games, and Erik Sintorn for generously sharing models and other assets. Many others at NVIDIA provided helpful suggestions, discussion, and brainstorming throughout this project, including: Anjul Patney, Henry Moreton, Cyril Crassin, Dave Luebke, Marco Salvi, Eric Enderton, and Craig Kolb.

References

AILA, T., AND LAINE, S. 2004. Alias-free shadow maps. In *Proc. Eurographics Symposium on Rendering*, 161–166.

AILA, T., KARRAS, T., AND LAINE, S. 2013. On quality metrics of bounding volume hierarchies. In *Proc. High-Performance Graphics*, 101–107.

ANNEN, T., MERTENS, T., SEIDEL, H.-P., FLERACKERS, E., AND KAUTZ, J. 2008. Exponential shadow maps. In *Proc. Graphics Interface*, 155–161.

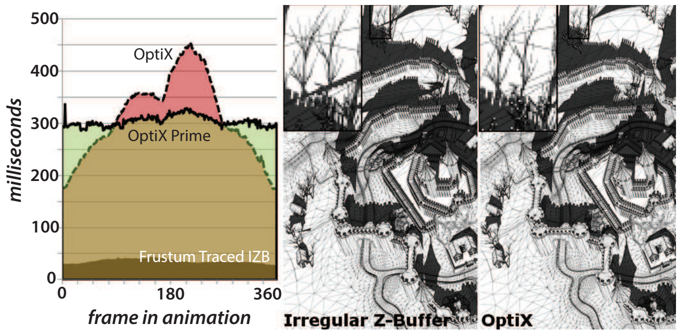
ARVO, J. 2007. Alias-free shadow maps using graphics hardware. *Journal of Graphics Tools 12*, 1, 47–59.

BOULOS, S., EDWARDS, D., LACEWELL, J. D., KNISS, J., KAUTZ, J., WALD, I., AND SHIRLEY, P. 2007. Packet-based Whitted and Distribution Ray Tracing. In *Proc. Graphics Interface*, 177–184.

CARPENTER, L. 1984. The a-buffer, an antialiased hidden surface method. In *Proceedings of SIGGRAPH*, 103–108.

CROW, F. 1977. Shadow algorithms for computer graphics. In *Proceedings of SIGGRAPH*, 242–248.

DONNELLY, W., AND LAURITZEN, A. 2006. Variance shadow maps. In *Symposium on Interactive 3D Graphics and Games*, 161–165.



Scene	Ray Tracing Numbers			32 spp IZB
	BVH Nodes Traversed	Tri-Frag Vis Tests	Avg Tests Per Pixel	Avg Tests Per Pixel
Chalmers Villa	11.0 M	10.7 M	14.6	**4.1**
Bungie Building	15.1 M	14.6 M	**8.8**	9.3
Epic Citadel	14.3 M	14.1 M	9.3	**8.6**
Chalmers Citadel	11.6 M	11.3 M	**7.3**	11.2
Bungie Terrain	12.9 M	12.4 M	**7.1**	18.7

Figure 10: *(Left) Compare our speed over an animation with a 32 spp OptiX ray tracer (on a Quadro K6000) and a quality comparison from one frame. (Right) The number of frustum-triangle intersections in a beam tracer (with a state-of-the-art BVH) compared with our system.*

EISEMANN, E., SCHWARZ, M., ASSARSSON, U., AND WIMMER, M. 2011. *Real-Time Shadows*. A. K. Peters, Ltd.

FERNANDO, R., FERNANDEZ, S., BALA, K., AND GREENBERG, D. 2001. Adaptive shadow maps. In *Proceedings of SIGGRAPH*, 387–390.

HASSELGREN, J., AKENINE-MOLLER, T., AND OHLSSON, L. 2005. *GPU Gems 2*. Addison-Wesley, ch. Conservative Rasterization, 677–690.

HECKBERT, P. S., AND HANRAHAN, P. 1984. Beam tracing polygonal objects. In *Proceedings of SIGGRAPH*, 119–127.

JOHNSON, G. S., LEE, J., BURNS, C. A., AND MARK, W. R. 2005. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph. 24*, 4 (Oct.), 1462–1482.

LAURITZEN, A., SALVI, M., AND LEFOHN, A. 2011. Sample distribution shadow maps. In *Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '11, 97–102.

LECOCQ, P., MARVIE, J.-E., SOURIMANT, G., AND GAUTRON, P. 2014. Sub-pixel shadow mapping. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, 103–110.

LEE, W.-J., SHIN, Y., LEE, J., KIM, J.-W., NAH, J.-H., JUNG, S., LEE, S., PARK, H.-S., AND HAN, T.-D. 2013. Sgrt: A mobile gpu architecture for real-time ray tracing. In *Proc. High-Performance Graphics*, 109–119.

LLOYD, D. B., WENDT, J., GOVINDARAJU, N., AND MANOCHA, D. 2004. CC shadow volumes. In *Proc. Eurographics Symposium on Rendering*, 197–206.

LLOYD, D. B., TUFT, D., YOON, S.-E., AND MANOCHA, D. 2006. Warping and partitioning for low error shadow maps. In *Proc. Eurographics Symposium on Rendering*, 215–226.

LLOYD, D. B., GOVINDARAJU, N. K., QUAMMEN, C., MOLNAR, S. E., AND MANOCHA, D. 2008. Logarithmic perspective shadow maps. *ACM Trans. Graph. 27*, 4, 106:1–106:32.

MCGUIRE, M., HUGHES, J. F., EGAN, K., KILGARD, M., AND EVERITT, C. 2003. Fast, practical and robust shadows. Tech. rep., NVIDIA Corporation, Austin, TX, Nov.

MITTRING, M., 2014. Real-time ray traced shadows. `http://kosmokleaner.wordpress.com/2014/09/26/`.

OVERBECK, R., RAMAMOORTHI, R., AND MARK, W. R. 2007. A real-time beam tracer with application to exact soft shadows. In *Proc. Eurographics Symposium on Rendering*, 85–98.

PAN, M., WANG, R., CHEN, W., ZHOU, K., AND BAO, H. 2009. Fast, sub-pixel antialiased shadow maps. *Computer Graphics Forum 28*, 7, 1927–1934.

REEVES, W., SALESIN, D., AND COOK, R. 1987. Rendering antialiased shadows with depth maps. In *Proceedings of SIGGRAPH*, 283–291.

SCHWARZ, M., AND STAMMINGER, M. 2007. Bitmask soft shadows. *Computer Graphics Forum 26*, 3, 515–524.

SEN, P., CAMMARANO, M., AND HANRAHAN, P. 2003. Shadow silhouette maps. *ACM Trans. Graph. 22*, 3 (July), 521–526.

SINTORN, E., EISEMANN, E., AND ASSARSSON, U. 2008. Sample based visibility for soft shadows using alias-free shadow maps. *Computer Graphics Forum 27*, 4, 1285–1292.

SINTORN, E., OLSSON, O., AND ASSARSSON, U. 2011. An efficient alias-free shadow algorithm for opaque and transparent objects using per-triangle shadow volumes. *ACM Trans. Graph. 30*, 6, 153:1–153:10.

SINTORN, E., KÄMPE, V., OLSSON, O., AND ASSARSSON, U. 2014. Per-triangle shadow volumes using a view-sample cluster hierarchy. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, 111–118.

STAMMINGER, M., AND DRETTAKIS, G. 2002. Perspective shadow maps. In *Proceedings of SIGGRAPH*, 557–562.

WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (June), 343–349.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. In *Proceedings of SIGGRAPH*, 270–274.

WOO, A., AND POULIN, P. 2012. *Shadow Algorithms Data Miner*. A. K. Peters/CRC Press.

YANG, J. C., HENSLEY, J., GRÜN, H., AND THIBIEROZ, N. 2010. Real-time concurrent linked list construction on the gpu. *Computer Graphics Forum 29*, 4, 1297–1304.