Frustum-Traced Irregular Z-Buffers: Fast, Sub-pixel Accurate Hard Shadows

Chris Wyman, *Member, IEEE,* Rama Hoetzlein, and Aaron Lefohn

(Invited Extension of I3D 2015 Paper)

Abstract—We further describe and analyze a real-time system for rendering antialiased hard shadows using irregular z-buffers (IZBs) that we first presented in Wyman et al. [1]. We focus on identifying bottlenecks, exploring these from an algorithmic complexity standpoint, and presenting techniques to improve performance. Our system remains interactive on a variety of game assets and CAD models while running at resolutions 1920×1080 and above and imposes no constraints on light, camera or geometry, allowing fully dynamic scenes without precomputation. We render sub-pixel accurate, 32 sample per pixel hard shadows at roughly twice the cost of a single sample per pixel. This allows us to smoothly animate even subpixel shadows from grass or wires without introducing spatial or temporal aliasing.

Prior algorithms for irregular z-buffer shadows rely heavily on the GPU's compute pipeline. Instead we leverage the standard rasterization-based graphics pipeline, including hardware conservative raster and early-z culling. Our key observation is noting a duality between irregular z-buffer performance and shadow map quality; irregular z-buffering is most costly exactly where shadow maps exhibit the worst aliasing. This allows us to use common shadow map algorithms, which typically improve aliasing, to instead 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 1 ms per frame.

Index Terms—shadows, irregular z-buffer, cascades, frustum tracing, alias-free shadows, rasterization.

1 Introduction

While conceptually simple, rendering artifact-free, antialiased hard shadows in real-time remains an enormous challenge. Most modern applications use variants of shadow mapping [2]. But due to discretization and sampling mismatches between eye- and light-space, robust and artifact free results remain elusive even with high quality filtering [3] and cascades [4].

We describe the evolution of our system, designed with a simple goal: interactive, sub-pixel accurate hard shadows on content representative of modern workloads at 1920×1080 and above. We began without any preconceptions except that shadow maps induce aliasing. Since aliasing is a difficult fundamental problem in graphics, addressing shadow map aliasing after the fact may be more challenging than simply computing shadows correctly. While desirable, extendability to soft shadows was not a design goal; our primary goal was robust, antialiased hard 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 [5], shadow volumes [6], and irregular z-buffers [7]. Fundamentally, all three 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 primitives representing 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 [8] to accelerate brute force, per-triangle volumes.

Little research has explored efficient algorithms for IZBs. Initial work proposed major hardware changes [7] and alternative data structures [9]. Later work [10], [11] showed hardware implementations using GPU computing capabilities rather than the graphics pipelines. Pan et al. [12] improved quality, demonstrating antialiased shadow boundaries. To our knowledge, nobody has eliminated the remaining 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 hairball. 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 insight: 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 our performance.

Additional contributions of our system include:

- Antialiased shadows with consistently interactive performance.

• *C. Wyman, R. Hoetzlein, and A. Lefohn are with NVIDIA Corporation in Redmond, WA and Santa Clara, CA. E-mail: chris.wyman@acm.org*

Manuscript received ???; revised ???.

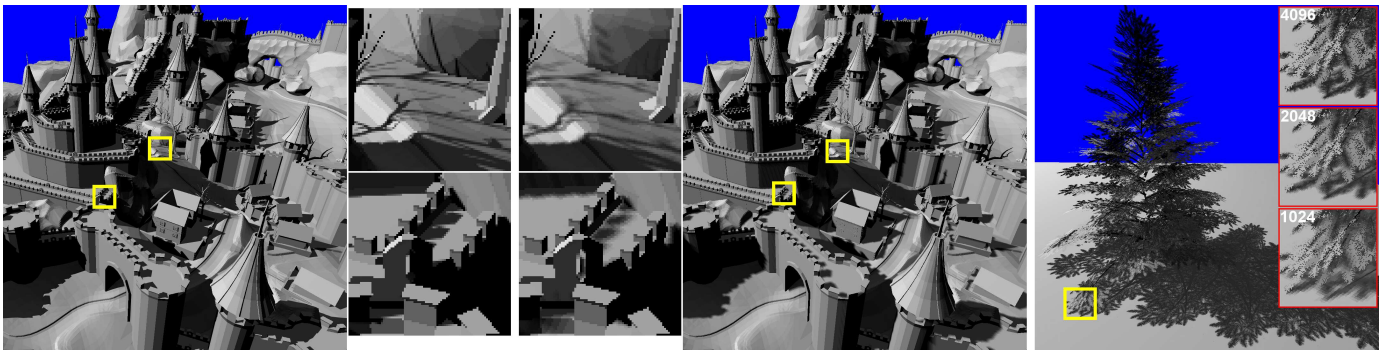


Fig. 1. (Left) Our 32 sample per pixel hard shadows in the 613k triangle Chalmers Citadel (8.1 ms at 1920×1080). (Center) Compared with a filtered 8092^2 shadow map, we avoid temporal and spatial aliasing from sampling mismatches and eliminate light leaking for contact shadows. (Right) Shadows from a billboarded pine tree with alpha mapped textures (5.9, 3.2, and 2.8 ms for three IZB resolutions at 1920×1080). To clarify the impact of our work we used no image-space or postprocess antialiasing, so primary visibility remains aliased.

- An efficient implementation using existing graphics pipelines to improve culling beyond that available to code using only GPU compute.
- An efficient extension to render 32 sample per pixel (spp) shadows.
- Extensive performance evaluation with quantified gains for individual optimizations.
- An analysis of algorithmic complexity that theoretically motivates each of our optimizations.
- An extension of our algorithm to handle shadows from alpha mapped textures.

To clarify these claims, we still explicitly sample shadows in eye-space and *any* discrete sampling can introduce aliasing. However, sampling rate is developer specified and reduces performance sublinearly with number of samples. Unlike shadow mapping, we introduce no new aliasing due to sampling mismatches between eye- and light-space.

2 Previous Work

Decades of shadow research provide developers with many algorithmic choices [13] [14]. But until recently, the choice in interactive contexts was limited to variants of shadow volumes [6] and shadow maps [2].

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 advances addressing these issues [15] [16] 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 [17] and the use of statistical models [3] [18] partially hides aliasing, but often introduces other artifacts. Distorting the light frustum [19], adaptively refining [20], or fitting multiple shadow maps [4] improves sampling quality but can introduce overhead, reduce robustness, and still exhibit aliasing.

Advances in the performance of ray tracing [5] may provide another alternative. Hardware acceleration improves ray tracing performance [21], but adoption rates are unclear. Ray query costs strongly depend on high quality acceleration structures [22], which remain relatively expensive to build. Concurrent work [23] suggests existing GPUs may allow game quality ray traced shadows, though their grid of lists structure resembles IZBs more than standard bounding volume hierarchies.

Irregular z-buffers [7] provide another option within existing graphics pipelines. Shadow maps use a light-space z-buffer; IZBs instead use a light-space A-buffer [24], with each light-space 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 [25]. Key problems with IZBs include poor scalability and high performance variability. Prior GPU implementations [11] [12] typically rendered images only at 512^2 using models with under 100k triangles; performance scaled linearly with triangle and pixel counts. Our initial IZB prototype exhibited 100:1 performance variations between some frames during even basic movements like slight camera translations and rotations.

However, these problems lie entirely in the performance domain. While Johnson et al. [7] explored performance characteristics on their proposed hardware, performance 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 11) required 12.6 seconds in early work by Aila and Laine [9], 1.5 seconds in our first GPU prototype, and 7.0 milliseconds today.

3 Irregular Z-Buffer Review

Before describing our system, we briefly review irregular z-buffer shadows. Johnson et al. [7] provide a much more complete description worth revisiting for further details.

Irregular z-buffers avoid shadow map aliasing due to mismatches between eye- and light-space sampling locations. Shadow maps use a regular sampling grid in both eye- and light-space, and finding a robust bijection between geometric samples on these grids remains unsolved. By allowing light-space samples to occur irregularly, IZB shadows easily pair samples. Visibility tests occur only where needed—at pixels visible to the eye.

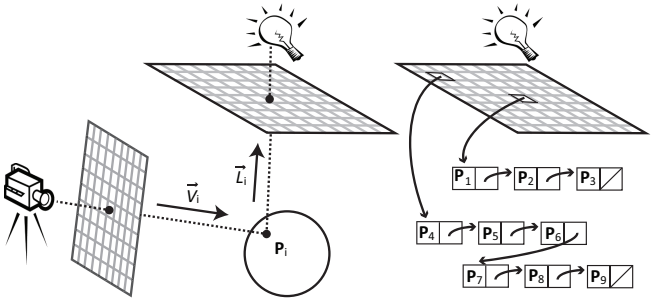


Fig. 2. (Left) A familiar mapping between view vector $\vec{V}_i$, intersection point $\mathbf{P}_i$, and light direction $\vec{L}_i$. Shadow maps discretize light-space, so shadow queries return the nearest neighbor rather than the visibility along $\vec{L}_i$. (Right) Irregular z-buffer shadows store all sample points $\mathbf{P}_i$ that project into each light-space texel, 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.

By construction, an IZB bijectively maps each pixel visible from the eye 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, only approximating true visibility. IZBs store all samples and generate an accurate visibility for every pixel.

3.1 Irregular Z-Buffer Construction

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 only rasterize over regular samples, we need to store our irregular samples in a grid. A grid of lists structure achieves this. Our 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 store a head pointer rather than a depth.

To create this grid of lists, we need to know where in light-space pixels occur. This means IZBs require a prepass to identify sample locations. Game engines commonly use an eye-space z-prepass or deferred shading with a G-buffer [26] to reduce overshading. Either provides exactly the needed data: locations of visible pixels requiring shadow queries. To identify IZB samples we run a compute pass over this buffer, transforming pixels into light-space (via a standard shadow map transformation) and inserting them into their corresponding light-space lists (see Figure 2).

3.2 Computing Visibility with an Irregular Z-Buffer

Once we have our light-space grid of visible pixels, we can compute per pixel shadowing by rasterizing occluder geometry in light-space. Each rasterized fragment represents potential occlusion between its parent triangle and any pixels in its corresponding light-space linked list. Traversing the list and performing ray-triangle intersection or point-in-shadow volume tests provides pixel accurate visibility.

Since pixels can lie anywhere within a light-space texel, conservative rasterization is required; triangles must test pixels for occlusion if the triangle covers any portion of a texel (not just the center as in traditional rasterization).

3.3 Irregular Z-Buffer Pseudocode

Pseudocode describing this process follows:

High Level Pseudocode: Irregular Z-Buffer Shadows

```
// Step 1: Identify pixel locations we need to shadow
G(x, y) ← RenderGBufferFromEye()
// Step 2: Add pixels to our light-space IZB data structure
for pixel p ∈ G(x, y) do
  lsTexel_p ← ShadowMapXform[ GetEyeSpacePos( p ) ]
  izbNode_p ← CreateIZBNode[ p ]
  AddNodeToLightSpaceList[ lsTexel_p, izbNode_p ]
end for
// Step 3: Test each triangle with pixels in lists it covers
for tri t ∈ SceneTriangles do
  for frag f ∈ ConservateLightSpaceRaster( t ) do
    lsTexel_f ← FragmentLocationInRasterGrid[ f ]
    for node n ∈ IZBNodeList( lsTexel_f ) do
      p ← GetEyeSpacePixel( n )
      visMask[p] = visMask[p] | TestVisibility[ p, t ]
    end for
  end for
end for
```

Initially this seems complex, but it is a relatively simple modification to shadow mapping, essentially swapping the order of light-space rasterization and sample projection and inserting a list traversal instead of a simple z-comparison:

High Level Pseudocode: Shadow Maps

```
// Step 1: Render shadow map in light-space
for tri t ∈ SceneTriangles do
  for frag f ∈ RasterizeInLightSpace( t ) do
    lsTexel_f ← FragmentLocationInRasterGrid[ f ]
    if z_f < Z( lsTexel_f ) then Z( lsTexel_f ) ← z_f
  end for
end for
// Step 2: Query shadow map for each pixel
for pixel p ∈ FinalRender do
  lsTexel_p ← ShadowMapXform[ GetEyeSpacePos( p ) ]
  visMask[p] ← DistanceToLight( p ) > Z( lsTexel_p )
end for
```

3.4 A Comment on Notation

To ensure clarity throughout the paper, the term *pixel* represents a sample in eye-space, i.e., a fragment that will be shaded on screen. A *texel* refers to a sample in light-space. We compute primary visibility by rasterizing over pixels and compute shadows by rasterising over texels.

The IZB itself is a grid of lists (shown in Figure 2). We refer to the grid component as the *head texture*, since each texel stores the head pointer for a linked list. The lists store *nodes*, each of which corresponds to a pixel on screen.

When using multisample shadows, each pixel may spawn multiple IZB nodes. However, each node still represents an *entire* pixel (including all of its subsamples). This is why in Section 4 we discuss frustum-triangle tests—each node is an entire pixel, representing a frustum to the light.

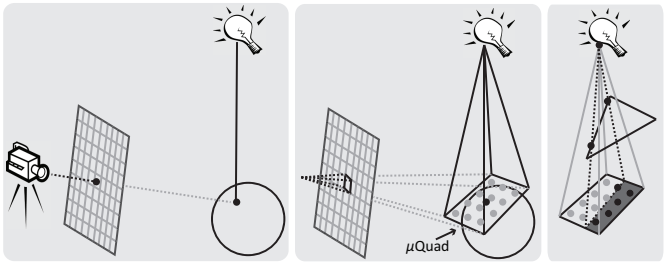


Fig. 3. (Left) A standard shadow ray query. (Center) Creating fragment shadow frustum (at black point): project pixel footprint to the fragment tangent plane; this "micro-quad" (μQuad) becomes the shadow frustum base. (Right) Intersection tests project each triangle edge to the tangent plane (i.e., identifying 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 lookup table construction.

4 Antialiasing via Frustum-Triangle Tests

Like ray tracing, IZB shadows provide one binary visibility for each query—as described above, once per pixel. To achieve antialiased shadows we could simply add more samples per pixel, independently adding each to our irregular z-buffer. This correspondingly increases the lengths of our linked lists, so this naive approach increases cost linearly with sample count.

Instead, we took inspiration from beam tracing [27], which can provide analytic subpixel visibility. Interactive work often replaces beams with packet tracing [28], tiled rasterization, or raster stamps to preserve spatial coherence while using discrete samples. Numerous soft shadow techniques trace beams from a point in space to the area light, including algorithms for ray traced shadows [29], bitmask soft shadows [30], and even IZB based soft shadows [11].

Our insight was by flipping these beams around we would achieve antialiased shadows, tracing a frustum from the point light back to a pixel footprint on the geometry (see Figure 3). This is similar to Pan et al.'s [12] approach, though we directly compute intersections in world space rather than projecting back to the image plane.

More specifically, irregular z-buffers effectively query ray-triangle intersections between a rasterized triangle and each pixel stored in overlapping IZB linked lists. We achieve antialiased shadows by replacing these ray-triangle intersections with frustum-triangle intersections. Efficient frustum intersection occurs per pixel as follows:

1) Construct μQuads representing each pixel's footprint projected onto the tangent plane (see Figure 3).
2) Independently transform each occluder edge into a shadow plane bounding its shadow volume.
3) Project each shadow plane onto the μQuad. Compute occlusion via lookup table, combine with contributions from other edges.

As this intersection represents the inner loop, tight optimization is vital for good performance. We describe these steps in more detail below.

4.1 μQuad Construction

A μQuad is a simple construct: the projection of a pixel footprint onto the visible geometry's tangent plane. Footprint computation can be done on the fly during visibility testing (if geometric normals are available), though we found precomputation as part of G-buffer creation more efficient.

4.2 Shadow Plane Construction

For visibility, we test if points lie in each triangle's shadow volume rather than using explicit ray-triangle intersection. A triangle's shadow volume can be described as the intersection region of four negative half planes: the triangle plane, and the shadow planes for each triangle edge. For point samples (as in Section 3), this means our visibility tests require just four dot products.

For multisample shadows, we also use these shadow planes. We compute them per primitive (i.e., in a geometry shader) based on light position and two triangle vertices.

4.3 Visibility Computation

With the μQuad and shadow planes, we could compute per-pixel shadow occlusion analytically. To allow easy accumulation between subsequent triangles, we instead discretize visibility as binary samples distributed over the μQuad. The number and distribution of samples are developer specified. We use 32 samples from a 2D Halton sampling.

To compute discrete visibility, we treat each of the four planes independently. We project each shadow plane onto the μQuad, giving a line. We parameterize this line using polar coordinates (r, θ) relative to the center of the μQuad. These coordinates are normalized as if on a canonical unit square and then discretized to 5-bits in each r and θ.

We use this 10-bit lookup to index into a precomputed table, returning 32 binary visibility samples (with ones representing occlusion). We can bitwise AND the results from the lookups together to obtain a μQuad's visibility from the triangle. This is similar to lookup tables from, for example, Schwarz and Stamminger [30], Fiume and Fournier [31], and Kautz et al. [32].

5 Algorithmic Complexity

As with ray tracing, the key unit of work in irregular z-buffer shadows is a ray-triangle (or frustum-triangle) visibility test. Simplistically, the algorithmic complexity is $O(N)$ for N visibility tests. These N tests are spawned when a shader traverses the lists of potentially occluded pixels for each light-space fragment (in pseudocode, step 3); each pixel represents a ray (from $\mathbf{P}_i$ along $\vec{L}_i$) that is tested for intersection with the rasterized triangle. Prior performance analyses (e.g., Pan et al. [12]) reported N directly, and aimed solely to reduce this value.

Decomposing N gives a more insightful bound, $O(L_a F)$, where F is the total number of light-space fragments and L_a is the average list length in light-space.

Consider the average list length L_a. We test visibility at only eye-space pixels, so the number of IZB nodes depends on eye-space resolution R_{es}. Each pixel is entered into the IZB once, so R_{es} pixels go into a light-space resolution (R_{ls}) grid of lists; another useful decomposition is $L_a \approx R_{es}/R_{ls}$.

However, for GPUs or similar single instruction, multiple thread (SIMT) compute devices, not all parallel threads

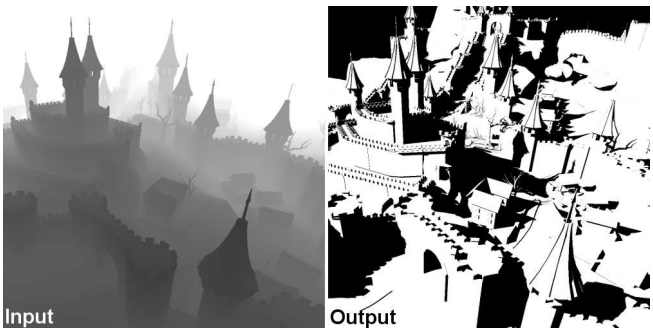


Fig. 5. The input to our irregular z-buffer shadows is the eye-space fragment positions, which can come from a z-prepass (left) or a G-buffer. We compute a multisample visibility mask for each pixel (right), which is consumed during final shading.

will have identical workloads. This divergence causes underutilization as some threads wait for others to finish. In these cases, $N = L_m F$ rather than $L_a F$, where L_m is the maximal light-space list length.

This means the performance can be improved by reducing the number of light-space fragments F, reducing light-space list lengths L_a and L_m, and reducing the variance of list lengths (i.e., causing L_m to approach L_a).

6 System Overview

As discussed in Section 3, irregular z-buffer shadows have three distinct phases: creating the irregular z-buffer, 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 efficiently implement these steps using 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.

6.1 Eye-Space Z-Prepass

To efficiently render antialiased shadows, we need to test occluder visibility at (and only at) fragments visible in the final rendering. Hence, creating 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 (depicted in Figure 5). Our prototype simultaneously creates a G-buffer, used during our deferred final render phase.

Single sample shadows only require fragment depths. To perform frustum intersection for antialiased shadows, we add three floats to the G-buffer describing the μQuad projection onto the tangent plane.

These three floats represent the distance from eye to tangent plane at three of the four μQuad corners. This is sufficient (when combined with fragment location and viewing parameters) to reconstruct these three corner locations in the shader and provides enough information to compute our lookup table indices. Storing these three floats accelerates visibility tests, but μQuads could be explicitly recomputed if G-buffer space is tight.

Fig. 6. Light-space visualizations of intermediate IZB steps. (Left) a standard light-space shadow map for comparison. (Center) A visualization of the irregular z-buffer, showing the length of lists in each light-space texel. White light-space texels contain no visible eye-space fragments. Darker pixels indicate longer lists, containing up to 100 fragments for the citadel scene. (Right) To reduce work during light-space rasterization, we can cull triangle fragments covering white texels – these spawn no visibility tests. We create a z-buffer with 0 in these texels, and the furthest visible fragment elsewhere.

6.2 Scene Bounds

As in shadow mapping, the light's projection matrix needs to tightly bound the scene. Poor bounds increase list length variability (i.e., L_m increases). To avoid poor scene bounds, we recompute the light's projection matrix each frame so the IZB contains only geometry visible in the z-prepass.

We use a persistent thread compute shader to loop over the z-buffer from Section 6.1 to compute a bounding box. This has cost dependent on screen resolution, though the z-buffer can be coarsely sampled for improved speed. Other approaches to bound the scene could be used instead, including explicit bounds from the engine or scene graph or min-max mipmaps [33] over the z-buffer. We did not explore alternate approaches, as this step is not a major bottleneck in most scenes.

6.3 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 outlined in Section 3 (and detailed in Section 7.2). Figure 6 shows the light space structure created for the Chalmers Citadel input from Figure 5.

The intuition: in a shadow map, p_i pixels project into a light space texel t_i, which causes aliasing if $p_i > 1$. In our irregular z-buffer, texel t_i contains an explicit list of these p_i pixels, allowing us to analytically compute visibility later.

Using multiple visibility samples per pixel can require multiple IZB nodes per pixel. Unlike a point query, a μQuad may project to multiple light-space texels; triangles touching any of these texels could shadow the pixel. Naively, a n sample per pixel shadow requires n IZB nodes per pixel. However, since we need not insert a pixel redundantly into light-space lists, if a μQuad projects to m light-space texels, we need $\min(m, n)$ IZB nodes.

Based on an empirical observation about m for our μQuads, Section 7.1 introduces an approximation using at most eight (and averaging two) IZB nodes per pixel with little quality impact.

6.4 Light-Space Culling Prepass

In Section 6.5 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.

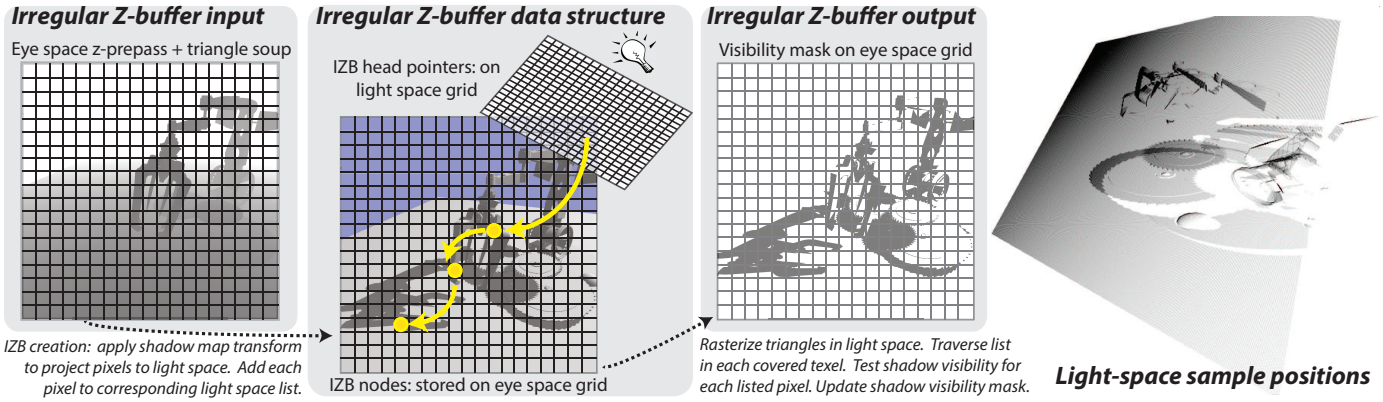


Fig. 4. An overview of our irregular z-buffer implementation. We require an eye-space z-buffer plus triangle geometry as input; each pixel location is transformed to light-space and added to the projected texel's linked list. Triangles are rasterized over this light-space grid; each fragment traverses 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) A visualization of the corresponding irregular light-space samples. Intensity represents number of pixels projecting into a light-space texel (darker means more pixels, white means no pixels project to a texel).

Using early-z hardware requires a light-space z-buffer. Section 6.3 could generate this as a side effect. 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.

This prepass essentially creates a stencil: setting depth to 0 in texels with empty lists and the distance to the furthest IZB node elsewhere (see Figure 6). Except in our most trivial scenes, hardware z-cull provides a substantial speedup (between 30 and 50%) as it reduces the number of fragments F generated by rasterization and reduces variability in list lengths (by skipping computations on lists of length zero).

6.5 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). Conservative rasterization is required, as pixels stored in the IZB may lie anywhere within the volume represented by a texel; triangles partially covering a texel may occlude an arbitrary subset of its list.

Each light-space fragment traverses the entire texel list. During traversal we load each eye-space pixel in the list, perform a frustum-triangle visibility test, and atomically OR the result into the pixel's eye-space 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.

6.6 Final Render

Section 6.5 performs shadow tests and stores results in a per-pixel visibility mask. Our final render pass loads from the G-buffer (Section 6.1) and this mask and performs Phong shading modulated by the shadow mask.

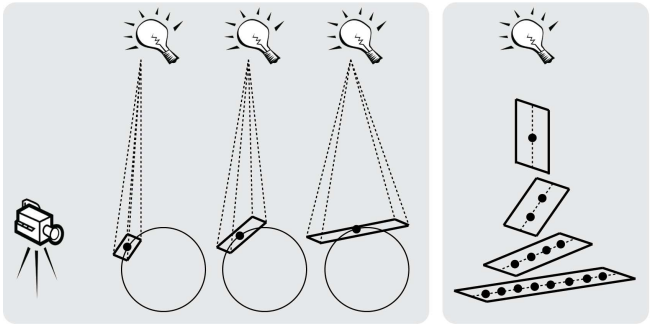


Fig. 7. (Left) As a fragment's tangent plane changes orientation, μQuads elongate along only one axis (the other dimension depends on screen resolution). (Right) This suggests sampling them one dimensionally; we add from one to eight samples to the IZB depending on orientation.

7 Implementation Details

Section 6 provides a high level description of our system, but key aspects deserve greater explanation. First, we discuss how multisample 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.

7.1 Simplifying IZBs for Multi-Sample Shadows

Moving to 32 spp shadows complicates our algorithm. The key is that light-space rasterization must spawn frustum-triangle tests (described in Section 4) for any triangle occluding a pixel. Since a μQuad has finite area, it projects to a variable number of light-space texels; its pixel must be added to each of those texels' lists to avoid light leaking.

One expensive approach would rasterize μQuads in light-space during IZB construction. Another method adds all 32 subpixel visibility samples to the IZB. We initially prototyped this sample-based insertion, though even with optimization to avoid duplicate IZB entries it often adds 8 or more nodes to the IZB for each pixel in the final render.

Considering μQuads' geometric behavior suggests a simple approximation to improve performance. μQuads enlarge along eye-space silhouettes as $\vec{N} \cdot \vec{V} \to 0$ (see Figure 7).

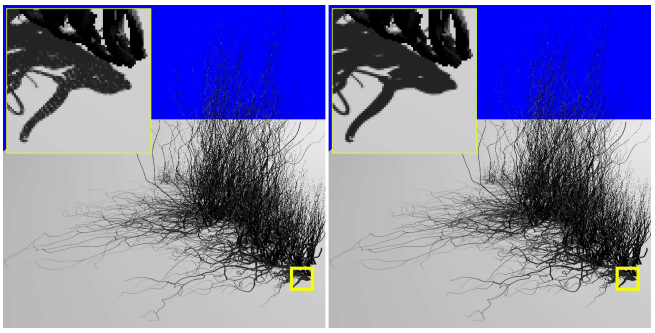


Fig. 8. (Left) The tentacles scene using our approximate insertion from Section 7.1. (Right) Using an over-conservative, 1 pixel dilation eliminates this light leaking.

Fig. 9. Our approximate insertion does not entirely eliminate light leaks, especially with poorly chosen IZB resolution. (Left) The 750k triangle YeahRight model using four 4096^2 IZB cascades rendered at 1920×1080. (Center) Zooming far away, with four 4096^2 cascades casting shadows onto a roughly 100^2 region introduces an extreme sampling mismatch that still leaks light with our approximation. (Right) Using only two 512^2 cascades better matches image resolution and removes noticeable light leaking.

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 only *approximates* the irregular z-buffer. As μQuads grow we miss inserting nodes in some light space texels, introducing light leaks for small or distant occluders falling between our approximate 1D samples (see Figure 8). A few ways exist to reduce light leaks. The easiest is to closely match eye- and light-space sampling rates, when our approximation works well.

When sampling rates cannot be exactly matched, another way is to use an overly conservative rasterization in light-space. Standard conservative rasterization effectively dilates each triangle by $\frac{1}{2}$ pixel (e.g., see [34]), but we use a 1 pixel dilation for our conservative raster. The intuition: when doing sphere-sphere intersections, you can either analytically intersect two spheres or do a point-in-sphere test, testing the center of sphere S_1 against an artificially enlarged sphere S_2 (with radius $r_1 + r_2$). Since we reduced the dimension of the μQuads, we compensate by slightly enlarging each triangle.

Given the complexity analysis in Section 5, this trade-off may seem questionable. Reducing the number of IZB nodes directly reduces list length L_a. Extra triangle dilation increases our fragments F, but by a much smaller amount. Approximate insertion averages two IZB nodes per pixel compared to eight with the exact approach, for a $4\times$ reduction in L_a; increasing dilation from $\frac{1}{2}$ to 1 pixel only increases F between 6–40%, giving a large net speedup.

Please note this insertion is *approximate*, but remaining light leaking occurs in extreme cases and when parameters have been poorly set. Figure 9 also shows remaining light leaking for a very distant occluder when the IZB resolution has been set way too high—though viewed in isolation an inexperienced viewer may not easily identify which result is correct.

7.2 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 [11] 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 explicit G-buffer index by storing it implicitly. 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. In other words, the node at index $y \cdot screenWidth + x$ corresponds to the pixel (x, y) on screen. Besides cutting list size in half and eliminating explicit G-buffer indices, this provides other advantages. Inserting list nodes no longer requires global synchronization, as we directly map pixels 1:1 to node indices (though we still need per-list synchronization to update each head pointer). 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, each pixel corresponds to multiple nodes. However 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 naive linked list structure). But we felt the improved performance was worth this moderate memory increase.

7.3 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 $O(L_a F)$ complexity. Halving resolution grows L_a by $4\times$ while lowering F by $4\times$, suggesting a minimal performance impacts. But conservative raster generates extra fragments, which has larger impact at lower resolutions. Larger resolutions increase memory consumption for the light-space grid, though the number of IZB nodes is largely invariant with light-space resolution.

Considering our goals, we want neither high L_a nor lots of fragments testing empty lists. This suggests closely matching light- and eye-space resolutions. Early tests showed a 2048^2 head texture worked well for all scenes at 1920×1080, though later experiments (Figure 14) suggest the sweet spot varies from 1400^2 to 2500^2.

7.4 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 [19], logarithmic [35], cascaded [36], and sample distribution shadow maps (SDSMs) [4] 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 [4] logarithmic partitioning and individually bounds each cascade.

We generate a separate IZB for each cascade. Creation of cascaded IZBs occurs in parallel, as each cascade can be defined containing a unique set of IZB samples. This is trivial for single sample shadows, but requires care for multisampled shadows to ensure μQuads that span cascade boundaries are split between them correctly. We create all IZBs into a single 2D texture array and also perform the light-space culling prepass (Section 6.4) in parallel into an array of depth textures.

Light-space rasterization needs to occur over each IZB to accumulate full visibility. We naively use one render pass per cascade. Using a single render pass to route primitives to the appropriate cascade could further improve performance.

Cascades' divergence reduction pays for the overhead of rerasterizing geometry (see Figure 10). Usually, two cascades sufficiently smooth over performance cliffs due to sudden changes in divergence. For modern game scenes, moving to three or four cascades slightly reduces average performance but provides tighter bounds on performance variability. For larger CAD scenes, moving beyond two cascades does not provide sufficient benefit to overcome the extra overhead of rerasterizing geometry additional times.

7.5 $\vec{N} \cdot \vec{L}$ Culling

Standard lighting models trivially shadow when $\vec{N} \cdot \vec{L} \leq 0$. As this term already shadows backfacing pixels, their visibility mask can be left fully lit. Not adding pixels with $\vec{N} \cdot \vec{L} \leq 0$ (for either shading or geometry normals) to the IZB reduces L_a. This also avoids a common problem along light silhouettes where geometric and shading normals provide different shadow terms. Adding this culling consistently improves performance 10-15%.

7.6 Early Out: IZB Node Removal

While testing visibility, we often determine a pixel has become fully shadowed. Occluders rasterized later in the frame can have no additional impact, so spawning additional frustum-triangle tests for this pixel is wasteful. This suggests removing fully shadowed 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-shadowed 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.

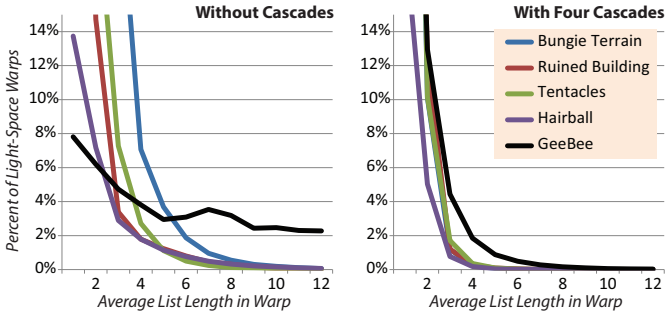


Fig. 10. A comparison of warp divergence in multiple scenes with and without cascades, with graphs showing the average lengths of IZB nodes traversed by threads in a warp. Since the number of light-space warps varies greatly between scenes, we normalize as a percentage of all warps. A key take away: in complex scenes without cascades, over 10% of warps average over 10 traversal steps per thread. With four cascades, *virtually none* average over 10 traversal steps in *any* scene.

7.7 Memory Synchronization: Unchanged Masks

Visibility tests are inexpensive relative to a GPU's maximum theoretical performance. Memory latency, throughput, and synchronization are thus key bottlenecks.

One synchronization point is a pixel's visibility mask; triangles testing visibility at the same pixel need to atomically combine results to avoid races. Hence, mask updates should only occur if a triangle *changes* the existing visibility. This requires loading the prior mask for comparison, but avoiding contention provides up to a 14% speed boost.

7.8 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.

8 Alpha Mapped Textures in IZBs

Alpha mapped textures that modulate geometric transparency on a per-texel basis are a vital part of most game content today. Prior papers on irregular z-buffer shadows generally avoid dealing with alpha mapped triangles, perhaps since a discretely sampled alpha inherently reintroduces aliasing, which goes against the goals of "alias free" techniques.

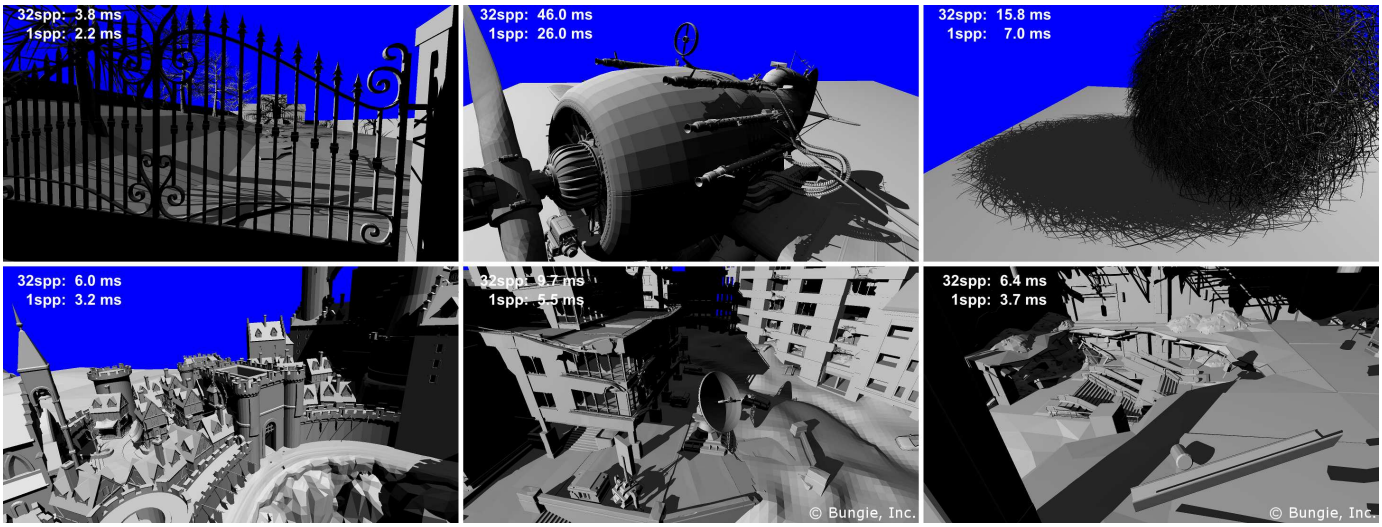


Fig. 11. Scenes used to test our system include: Chalmers Villa, GeeBee Plane, Hairball, Epic Citadel, Bungie Terrain, Bungie Building. Timings give total frame time using two 2048^2 cascades, rendered at 1920×1080. Some scenes courtesy Epic Games and Bungie, Inc.

The results of our per pixel shadow tests are stored in a visibility mask buffer (see Figure 5), but to enable efficient memory writes in this buffer we always store multiples of 32 bits for each pixel. We realized we could use these bits to represent opacity in addition to just geometric visibility. A common way of handling transparency for primary visibility is alpha-to-coverage, and we observe a similar approach can be used here. In other words, we first perform the visibility tests described in prior sections to compute geometric coverage, then determine the occluder's opacity by indexing into its alpha texture, perform alpha-to-coverage on this opacity value to get a 32-bit *opacity mask*, and finally combine this (via a bitwise AND) with our geometric coverage before outputting the result to our visibility buffer.

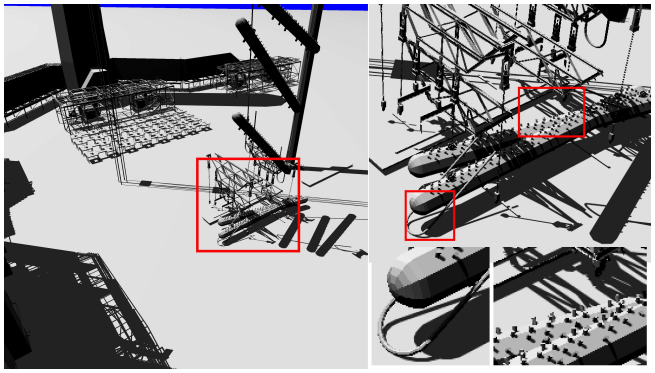


Fig. 12. The 12.3 million triangle UNC Powerplant model running at 3840×2160 in under 53 ms for 32 sample per pixel shadows. We zoom twice to highlight the shadow quality.

8.1 Determining Per-Query Alpha

To enable alpha mapped textures, we must ask how each IZB node determines the alpha of an occluding triangle. We see two ways to determine alpha for these visibility queries: a fast or a high quality way. For quick computation, the alpha texture can be queried once per light-space texel prior to traversing the list of IZB nodes. This assumes alpha remains constant over a light-space texel and reintroduces visible aliasing based on the light-space grid resolution. We implemented this approach.

A higher quality test queries the alpha texture once per IZB node. This allows each frustum-triangle visibility test to intersect the occluder with a different footprint, using varying texture coordinate and LoD bias to index into the alpha map. However, this adds substantial cost: a per-query footprint computation and alpha texture access.

9 Results and Discussion

Our system uses OpenGL 4.4 with various extensions. We tested on various NVIDIA GPUs and an AMD Radeon 290X to ensure cross-platform execution. Unless otherwise specified, we report times on a GeForce Titan X at 1920×1080 and rely on three NVIDIA-specific extensions for performance: *NV_conservative_raster*, *NV_conservative_raster_dilate*, and *NV_geometry_shader_passthrough*, enabling GPU-accelerated conservative rasterization not yet available elsewhere.

Figures 1, 11, 12, and 15 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 provides data for various high-level observations. First, most steps (i.e., from Section 6) have effectively constant cost, except for G-buffer creation and light-space rasterization (when we perform visibility tests). Computing scene extents requires a pass over all G-buffer samples running Lauritzen et al.'s [4] logarithmic partitioning to delineate cascades; this cost depends on screen resolution. Section 6.4 notes initializing the light-space z-buffer for culling may be unnecessary; however, it only requires a copy into our light-space z-buffer, which is fairly inexpensive. Our final render loads the G-buffer and visibility mask and applies a simple shading model.

IZB creation costs vary depending on the number of nodes inserted to our linked lists and how much atomic

TABLE 1
(1^{st} row) Performance breakdown for individual algorithmic steps for 32 spp (and 1 spp) IZB hard shadows. To allow comparison all scenes use identical, rather than optimal, settings: 1920×1080 resolution, 2 cascades each containing 2048^2 lists of IZB nodes, and using all optimizations. (2^{nd} row) Explicit measures of the work done, including light-space fragment counts (F), average tests per list (L_a), and total frustum-triangle (or ray-triangle) tests performed (N). Based on those counts, we provide average visibility tests performed per shaded pixel in the final rendering. (3^{rd} row) Statistics for speed of light tests, which enumerate all visibility tests then execute them fully coherently (no divergence). (4^{th} row) We performed our frustum tracing through a fast bounding volume hierarchy [22] to compare the number of visibility tests with those we generate.

Algorithmic Step Times	Chalmers Villa, 89k	Bungie Building, 255 k	Epic Citadel, 374 k	Chalmers Citadel, 613 k	Bungie Terrain, 1.5 M	Hairball, 2.9 M	Tentacles, 3.8 M	GeeBee Plane, 11.7 M	Powerplant, 12.3 M
G-buffer creation	0.5 (0.3)	0.8 (0.3)	0.6 (0.3)	0.7 (0.4)	1.2 (0.8)	1.5 (1.1)	1.1 (1.0)	5.4 (5.3)	4.1 (3.9)
Z-extent bounds	0.3 (0.3)	0.3 (0.3)	0.2 (0.2)	0.3 (0.3)	0.3 (0.3)	0.2 (0.2)	0.2 (0.2)	0.2 (0.2)	0.3 (0.3)
Create IZB	0.6 (0.3)	0.7 (0.4)	0.6 (0.4)	0.6 (0.4)	0.7 (0.4)	0.7 (0.4)	0.6 (0.4)	0.6 (0.3)	0.8 (0.4)
Z-cull setup	0.5 (0.5)	0.3 (0.3)	0.3 (0.3)	0.3 (0.3)	0.3 (0.3)	0.3 (0.3)	0.3 (0.3)	0.3 (0.3)	0.3 (0.3)
Raster tris over IZB	1.5 (0.5)	3.7 (1.7)	3.7 (1.4)	5.7 (2.4)	6.6 (3.1)	12.4 (4.4)	7.9 (4.7)	38.6 (19.3)	26.1 (14.4)
Final render	0.2 (0.2)	0.2 (0.2)	0.2 (0.2)	0.2 (0.2)	0.2 (0.2)	0.2 (0.2)	0.1 (0.1)	0.3 (0.2)	0.2 (0.2)
Total Time (msec)	3.8 (2.2)	6.4 (3.7)	6.0 (3.2)	8.1 (4.3)	9.7 (5.5)	15.8 (7.0)	10.9 (7.2)	46.0 (26.0)	32.6 (20.1)
Measures of Work Performed In Our System for 32 spp (and 1 spp)									
Light space frags, F ($\times 10^6$)	2.2 (0.9)	17.5 (9.5)	6.4 (4.1)	8.6 (5.4)	31.0 (15.2)	55.0 (15.5)	6.8 (1.7)	30.8 (10.8)	45.6 (17.8)
Visibility tests, N ($\times 10^6$)	4.9 (1.6)	17.6 (8.9)	12.5 (8.0)	17.8 (9.9)	36.7 (15.3)	21.3 (4.6)	8.1 (1.7)	64.1 (16.9)	25.6 (6.4)
Avg tests per tri frag, L_a	2.2 (1.7)	1.0 (0.9)	1.9 (2.0)	2.1 (1.8)	1.2 (1.0)	0.4 (0.3)	1.2 (1.0)	2.1 (1.6)	0.6 (0.4)
Avg tests per final pixel	3.0 (1.0)	8.5 (4.3)	7.8 (5.0)	9.9 (5.5)	17.7 (7.3)	13.2 (2.8)	7.0 (1.5)	41.9 (10.9)	12.9 (3.2)
Speed of Light Tests: Frustum Tracing with No Divergence or Irregular Z-Buffer Traversal									
SoL visibilty tests ($\times 10^6$)	6.8	20.6	19.9	31.8	40.7	83.0	9.6	–	37.6
SoL visiblity cost (msec)	1.3	6.1	2.5	3.8	9.2	31.5	2.6	–	9.5
SoL tests per ms ($\times 10^6$)	**5.2**	3.4	**8.0**	**8.4**	4.4	**2.6**	**3.7**	–	**4.0**
IZB tests per ms ($\times 10^6$)	3.3	**4.7**	3.4	3.1	**5.6**	1.7	1.0	1.7	1.0
Compare Data Structure Traversal: Bounding Volume Hierarchy (in a Ray Tracer) Versus Irregular Z-Buffer (with a Rasterizer)									
BVH nodes tested ($\times 10^6$)	11.0	15.1	14.3	11.6	12.9	–	–	–	–
BVH visibility tests ($\times 10^6$)	10.7	14.6	14.1	11.3	12.4	–	–	–	–
Avg tests per pixel (BVH)	14.6	8.8	9.3	**7.3**	**7.1**	–	–	–	–
Avg tests per pixel (IZB)	**3.0**	**8.5**	**7.8**	9.9	17.7	13.2	7.0	41.9	12.9

contention occurs. Incoherent geometry (hairball and tentacles) and finely tessellated models (GeeBee and powerplant) exhibit slightly lower performance. For single sample shadows, we add just one IZB node per pixel. 32 sample shadows insert roughly twice as many nodes (see Section 7.1), increasing atomic contention. Still, IZB creation costs remain remarkably consistent across all our test scenes.

9.1 Cost of Visibility Tests

Our major cost is light-space rasterization (Section 6.5), which tests μQuad visibility. Theoretically, cost varies linearly with additional frustum-triangle tests, though Table 1 shows this correspondence breaks in practice as GPU utilization varies between scenes. For game-like scenes with a mix of large and small polygons, we average between 3 and 4 million frustum-triangle tests per millisecond. For more complex scenes, we average between 0.6 and 1.5 million frustum-triangle tests per millisecond.

Early in development, we ran "speed of light" tests that explicitly enumerated all required frustum-triangle visibility tests then performed these tests in parallel using a compute shader. This requires 2+ GB to store all required visibility tests, but allows full GPU utilization. The $100\times$ cost difference between speed of light and early prototypes motivated our work. Table 1 provides speed of light numbers and shows we now average 67% the speed of fully coherent queries. In fact, in two scenes we *exceed* our speed of light; we attribute that to the overhead of reading 2+ GB lists of visibility queries, which the irregular z-buffer need not do.

Note: our speed of light test performed all visibility queries in parallel, disallowing culling and early removal of fully occluded pixels. This explains the strictly larger number of tests quoted (see Table 1) and provides a measure of improvement when using these optimizations: up to $4\times$.

TABLE 2
Total render times for some scenes from Table 1, shown with and without hardware-accelerated conservative rasterization. We use optimal settings for these scenes, so times vary from Table 1.

Scenes	**32 spp shadows**		**1 spp shadows**	
	With	Without	With	Without
Villa	3.4 ms	3.9 ms	2.0 ms	2.2 ms
Citadel	8.1 ms	10.1 ms	4.3 ms	5.1 ms
Hairball	15.4 ms	30.8 ms	6.8 ms	12.7 ms
Tentacles	10.9 ms	30.6 ms	7.2 ms	15.7 ms
Powerplant	32.2 ms	91.3 ms	20.0 ms	49.7 ms

9.2 Hardware-Accelerated Conservative Rasterization

IZB shadows require conservative rasterization. New DirectX12-class GPUs include hardware accelerated conservative raster, and newer NVIDIA GPUs further accelerate the added triangle dilation discussed in Section 7.1.

Table 2 shows the speed gains from hardware acceleration. The gains can exceed 3x in scenes with many small triangles, though more typical game scenes gain 10–20%.

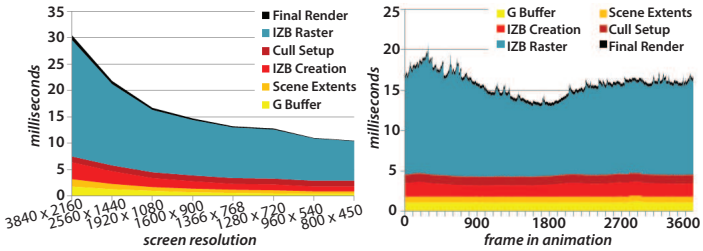


Fig. 13. Performance variation in the Chalmers Citadel (left) with varying render resolution and (right) over a 3600 frame animation at 1920×1080. Timings made on a GeForce 980 GTX.

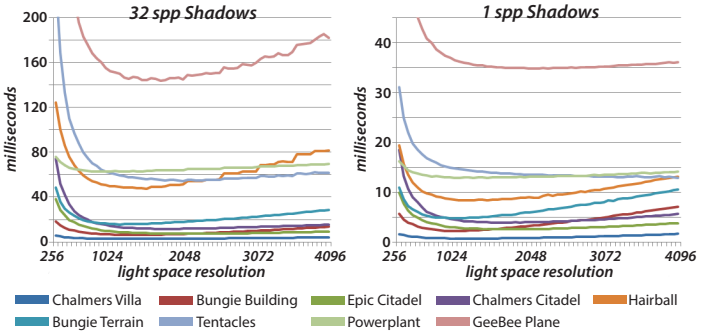


Fig. 14. Performance variation for (left) 32 spp and (right) 1 spp shadows with varying cascade resolutions. Timings represent just the light-space rasterization step (on a GeForce 980 GTX), using 4 cascades with resolution between 256^2 and 4096^2.

Roughly half this performance win comes from the new hardware raster (which generates fewer fragments than our software implementation, from Hasselgren et al. [34]), with the rest coming from simplified geometry shaders more amenable to hardware streaming.

9.3 Performance Scaling: Screen Resolution

Interestingly, Figure 13 shows non-linear performance scaling with screen resolution. Scaling from 1920×1080 to 3840×2160 roughly increases cost by $2\times$ rather than the expected $4\times$. This likely stems from our culling, specifically the node removal in Section 7.6, which reduces the penalty for longer linked lists. As fully occluded pixels are quickly eliminated, increasing resolution affects performance sub-linearly.

This also varies based on geometric complexity: while the citadel from Figure 13 takes twice as long at 3840×2160 as 1920×1080, the powerplant only increases in cost by 60%, and the tentacles by 30%. Though this effect may come from reduced divergence, as surfaces poorly sampled at lower resolution are more coherent when densely sampled.

9.4 Performance Scaling: Cascade Resolution

Figure 14 tracks performance under varying light-space resolution. Larger resolutions distribute IZB nodes among more lists (reducing L_a), but also cause rasterization to emit more fragments F. 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.

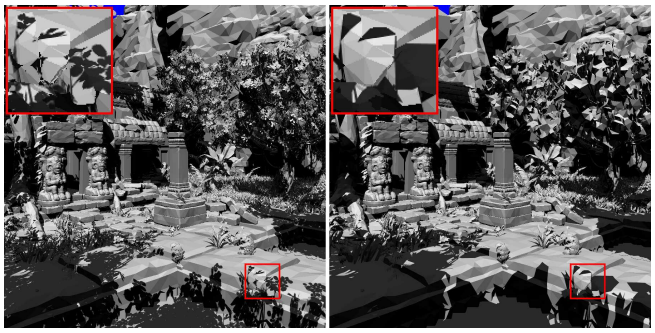


Fig. 15. The Epic Foliage Map with (left) and without (right) support for alpha mapped opacity rendered in 17.7 and 14.8 ms, respectively.

9.5 GPU Divergence

We instrumented our code to count the number of visibility tests done per frame, per warp, and per thread and ran these tests over various scenes. Figure 10 compares *average* visibility tests per light-space fragment (i.e., per thread) without cascades and using four cascades.

Since absolute fragment counts vary between scenes and frames, we show what percent have various workloads. Ideally, we always perform one visibility test per thread. Across our scenes, adding cascades moved us much closer to that ideal. Without cascades, our workload has a very long tail; many warps average over 100 tests per thread. With cascades virtually zero threads average over 10 tests per thread, even in our most complex scenes. We see similar reductions in work variance within a warp, with cascades giving a much more uniform work distribution.

On the Chalmers Villa scene we averaged 2.9 frustum-triangle visibility tests per thread without cascades. With 4 cascades, we averaged 1.5 visibility tests per thread. Due to rerendering into each cascade, we test 25% more fragments but it only takes 61% of the total time. This improves the throughput of visibility tests per millisecond by $2\times$. For difficult views, this can reach 10 or $100\times$.

9.6 Alpha Mapped Textures

Figures 1 and 15 show alpha mapped textures in our system. Our prototype uses the simple alpha query from Section 8 that reintroduces aliasing based on IZB resolution, as shown in Figure 1. Since good IZB performance already requires a good match between eye-space and light-space sampling, we believe this alpha aliasing will remain acceptable, especially since simple interpolation should reduce the blocking artifacts seen at lower resolutions.

Alpha mapping shadows reduce performance, due to additional memory traffic. Our naive prototype queries alpha for all surfaces, even those without transparency. Less performance loss should occur when using separate shaders, especially if all opaque triangles are rasterized prior to those with transparency.

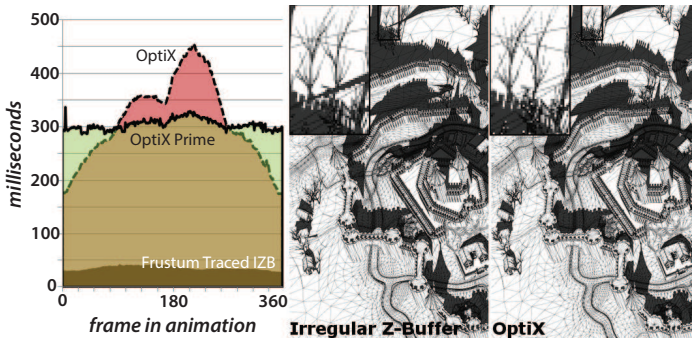


Fig. 16. (Left) Performance over an animation compared with a 32 spp OptiX ray tracer (on a Quadro K6000). (Right) A quality comparison from one animation frame.

9.7 Data Structure Quality

Fundamentally, the irregular z-buffer is an acceleration structure for visibility queries, akin to a ray tracer's bounding volume hierarchy. An interesting question is then how these acceleration structures compare. We implemented our frustum tracing in Optix and OptiX Prime (a wavefront based OptiX API [37]), with performance and quality comparisons depicted in Figure 16.

For ray traced quality comparisons, note the OptiX implementation uses a different sub-pixel sampling pattern and world-space epsilon value. This leads to slight differences in antialiasing patterns on branches and some missing shadows in the OptiX version, where occluders are mistakenly skipped due to a large epsilon. OptiX also supersamples pixels, allowing partial illumination on branches that we render as fully shadowed.

More interesting is an explicit comparison of visibility tests and traversal steps required, as ray tracers and rasterizers currently have vastly different levels of hardware acceleration. We implemented our frustum-tracing algorithm in a beam tracer using a state of the art bounding volume hierarchy [22]. The bottom row in Table 1 shows the number of visibility tests, N. The optimized IZB performs a similar number of tests as a high quality BVH, and even in outlying cases the number of tests are within a factor of $3\times$. This is surprisingly good performance considering we rebuild our IZB dynamically each frame in under 1 ms.

9.8 Specific Comparisons to Prior Work

Beyond ray tracers, our work shares similarities with other techniques. Lecocq et al. [38] and Sen et al. [39] 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 [8], [40]; 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 [9] 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.

9.9 Summary of Key Insights

Our key insight is the duality between shadow map quality and irregular z-buffer performance. Shadow maps alias where a texel projects onto multiple pixels. An irregular z-buffer stores per-texel lists of potentially occluded pixels. This enables IZBs to avoid all aliasing from eye- to light-space resampling in exchange for speed variability from variable length lists, i.e., where shadow maps have artifacts IZBs need more computation.

With this in mind, we explored various techniques to reduce visibility tests and mitigate GPU thread divergence. The most surprising observation leverages existing shadow mapping techniques, like cascades, to improve the performance of irregular z-buffering. However, we also showed that both raster techniques (z-culling) and ray tracing techniques (any-hit queries) work to improve our performance.

10 Conclusions

We designed a system that renders antialiased shadows in real time for both modern game content and CAD models. Adding 32 samples for subpixel visibility costs just two times more than a single sample. Quality is very robust and avoids the need to find a shadow map bias by trial and error.

We designed primarily for accuracy but introduced one optimization for multisample shadows that can introduce light leaks for small, distant occluder triangles. A modified light-space parameterization may eliminate this leaking, but developers needing quality can also skip this optimization.

11 Future Work

We hope to spur further exploration into real-time analytic shadows. A number of improvements still remain: a more programmable depth test could enable better culling during light space rasterization and applying conservative rasterization to all triangles seems wasteful (but robustly identifying silhouettes is challenging). Additionally, we did not fully explore ways to avoid inserting nodes into the irregular z-buffer. For instance, when inserting pixels into light-space lists we may find geometry that fully occludes a texel; we could avoid inserting more distant geometry.

This paper focused exclusively on hard shadows, though many applications today want soft shadows so handling area lights is important. Either an explicit analytic solution or an approximate image-space filtering approach similar to percentage closer soft shadows [41] would be valuable.

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

[1] C. Wyman, R. Hoetzlein, and A. Lefohn, "Frustum-traced raster shadows: Revisiting irregular z-buffers," in *Symposium on Interactive 3D Graphics and Games*, 2015, pp. 15–23.

[2] L. Williams, "Casting curved shadows on curved surfaces," in *Proceedings of SIGGRAPH*, 1978, pp. 270–274.

[3] T. Annen, T. Mertens, H.-P. Seidel, E. Flerackers, and J. Kautz, "Exponential shadow maps," in *Graphics Interface*, 2008, pp. 155–161.

[4] A. Lauritzen, M. Salvi, and A. Lefohn, "Sample distribution shadow maps," in *Symposium on Interactive 3D Graphics and Games*, 2011, pp. 97–102.

[5] T. Whitted, "An improved illumination model for shaded display," *Communications of the ACM*, vol. 23, no. 6, pp. 343–349, June 1980.

[6] F. Crow, "Shadow algorithms for computer graphics," in *Proceedings of SIGGRAPH*, 1977, pp. 242–248.

[7] G. S. Johnson, J. Lee, C. A. Burns, and W. R. Mark, "The irregular z-buffer: Hardware acceleration for irregular data structures," *ACM Transactions on Graphics*, vol. 24, no. 4, pp. 1462–1482, Oct. 2005.

[8] E. Sintorn, V. Kämpe, O. Olsson, and U. Assarsson, "Per-triangle shadow volumes using a view-sample cluster hierarchy," in *Symposium on Interactive 3D Graphics and Games*, 2014, pp. 111–118.

[9] T. Aila and S. Laine, "Alias-free shadow maps," in *Eurographics Symposium on Rendering*, 2004, pp. 161–166.

[10] J. Arvo, "Alias-free shadow maps using graphics hardware," *Journal of Graphics Tools*, vol. 12, no. 1, pp. 47–59, 2007.

[11] E. Sintorn, E. Eisemann, and U. Assarsson, "Sample based visibility for soft shadows using alias-free shadow maps," *Computer Graphics Forum*, vol. 27, no. 4, pp. 1285–1292, 2008.

[12] M. Pan, R. Wang, W. Chen, K. Zhou, and H. Bao, "Fast, sub-pixel antialiased shadow maps," *Computer Graphics Forum*, vol. 28, no. 7, pp. 1927–1934, 2009.

[13] E. Eisemann, M. Schwarz, U. Assarsson, and M. Wimmer, *Real-Time Shadows*. A. K. Peters, Ltd., 2011.

[14] A. Woo and P. Poulin, *Shadow Algorithms Data Miner*. A. K. Peters/CRC Press, 2012.

[15] M. McGuire, J. F. Hughes, K. Egan, M. Kilgard, and C. Everitt, "Fast, practical and robust shadows," NVIDIA Corporation, Tech. Rep., Nov 2003.

[16] D. B. Lloyd, J. Wendt, N. Govindaraju, and D. Manocha, "CC shadow volumes," in *Eurographics Symposium on Rendering*, 2004, pp. 197–206.

[17] W. Reeves, D. Salesin, and R. Cook, "Rendering antialiased shadows with depth maps," in *Proceedings of SIGGRAPH*, 1987, pp. 283–291.

[18] W. Donnelly and A. Lauritzen, "Variance shadow maps," in *Symposium on Interactive 3D Graphics and Games*, 2006, pp. 161–165.

[19] M. Stamminger and G. Drettakis, "Perspective shadow maps," in *Proceedings of SIGGRAPH*, 2002, pp. 557–562.

[20] R. Fernando, S. Fernandez, K. Bala, and D. Greenberg, "Adaptive shadow maps," in *Proceedings of SIGGRAPH*, 2001, pp. 387–390.

[21] W.-J. Lee, Y. Shin, J. Lee, J.-W. Kim, J.-H. Nah, S. Jung, S. Lee, H.-S. Park, and T.-D. Han, "Sgrt: A mobile gpu architecture for real-time ray tracing," in *High-Performance Graphics*, 2013, pp. 109–119.

[22] T. Aila, T. Karras, and S. Laine, "On quality metrics of bounding volume hierarchies," in *High-Performance Graphics*, 2013, pp. 101–107.

[23] M. Mittring, "Real-time ray traced shadows," http://kosmokleaner.wordpress.com/2014/09/26/, 2014.

[24] L. Carpenter, "The a-buffer, an antialiased hidden surface method," in *Proceedings of SIGGRAPH*, 1984, pp. 103–108.

[25] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz, "Real-time concurrent linked list construction on the gpu," *Computer Graphics Forum*, vol. 29, no. 4, pp. 1297–1304, 2010.

[26] T. Saito and T. Takahashi, "Comprehensible rendering of 3-d shapes," in *Proceedings of SIGGRAPH*, 1990, pp. 197–206.

[27] P. S. Heckbert and P. Hanrahan, "Beam tracing polygonal objects," in *Proceedings of SIGGRAPH*, 1984, pp. 119–127.

[28] S. Boulos, D. Edwards, J. D. Lacewell, J. Kniss, J. Kautz, I. Wald, and P. Shirley, "Packet-based Whitted and Distribution Ray Tracing," in *Graphics Interface*, 2007, pp. 177–184.

[29] R. Overbeck, R. Ramamoorthi, and W. R. Mark, "A real-time beam tracer with application to exact soft shadows," in *Eurographics Symposium on Rendering*, 2007, pp. 85–98.

[30] M. Schwarz and M. Stamminger, "Bitmask soft shadows," *Computer Graphics Forum*, vol. 26, no. 3, pp. 515–524, 2007.

[31] E. Fiume and A. Fournier, "A parallel scan conversion algorithm with anti-aliasing for a general-purpose ultracomputer," in *Proceedings of SIGGRAPH*, 1983, pp. 141–150.

[32] J. Kautz, J. Lehtinen, and T. Aila, "Hemispherical rasterization for self-shadowing of dynamic objects," in *Eurographics Symposium on Rendering*, 2004, pp. 179–184.

[33] A. Tevs, I. Ihrke, and H.-P. Seidel, "Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering," in *Symposium on Interactive 3D graphics and games*, 2008, pp. 183–190.

[34] J. Hasselgren, T. Akenine-Moller, and L. Ohlsson, *GPU Gems 2*. Addison-Wesley, 2005, ch. Conservative Rasterization, pp. 677–690.

[35] D. B. Lloyd, N. K. Govindaraju, C. Quammen, S. E. Molnar, and D. Manocha, "Logarithmic perspective shadow maps," *ACM Transactions on Graphics*, vol. 27, no. 4, pp. 106:1–106:32, 2008.

[36] D. B. Lloyd, D. Tuft, S.-e. Yoon, and D. Manocha, "Warping and partitioning for low error shadow maps," in *Eurographics Symposium on Rendering*, 2006, pp. 215–226.

[37] S. Laine, T. Karras, and T. Aila, "Megakernels considered harmful: Wavefront path tracing on gpus," in *High-Performance Graphics*, 2013, pp. 137–143.

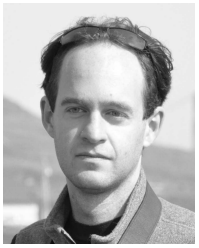
[38] P. Lecocq, J.-E. Marvie, G. Sourimant, and P. Gautron, "Sub-pixel shadow mapping," in *Symposium on Interactive 3D Graphics and Games*, 2014, pp. 103–110.

[39] P. Sen, M. Cammarano, and P. Hanrahan, "Shadow silhouette maps," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 521–526, Jul. 2003.

[40] E. Sintorn, O. Olsson, and U. Assarsson, "An efficient alias-free shadow algorithm for opaque and transparent objects using per-triangle shadow volumes," *ACM Transactions on Graphics*, vol. 30, no. 6, pp. 153:1–153:10, 2011.

[41] R. Fernando, "Percentage-closer soft shadows," in *ACM SIGGRAPH Sketches*, 2005, p. 35.

Chris Wyman is a Senior Research Scientist in NVIDIA's real-time rendering research group located in Redmond, WA. Prior to joining the team in Redmond, he was a Visiting Professor at NVIDIA and an Associate Professor of Computer Science at the University of Iowa. He earned a PhD in computer science from the University of Utah and a BS in mathematics and computer science from the University of Minnesota. Recent research interests include efficient shadows, antialiasing, and participating media, but his interests span the range of real-time rendering problems from lighting to global illumination, materials to color spaces, and optimization to hardware improvements.



Rama Hoetzlein is an artist and computer scientist, having completed a dual BFA and BA at Cornell University in 2001 and a PhD in Media Arts and Technology at the University of California Santa Barbara in 2010 with research in procedural modeling and systems for creative interaction. He was co-founder of the Game Design Initiative at Cornell, and Assistant Professor of Computer Graphics at Medialogy in Copenhagen, Denmark in 2011. Rama currently works at NVIDIA with research interests in fluid simulation and volume rendering.

Aaron Lefohn is Director of Real-Time Rendering Research at NVIDIA, has led real-time rendering and graphics programming model research teams for over six years, and has productized many research ideas into GPU hardware and GPU APIs. Prior to joining NVIDIA, Aaron led a real-time rendering and graphics programming model research team at Intel. He joined Intel in 2007 via Intel's acquisition of the graphics startup, Neoptica. Before Neoptica, Aaron worked in rendering R&D at Pixar Animation Studios, creating interactive rendering tools for film artists. Aaron received his PhD in computer science from UC Davis and his MS in computer science from University of Utah.