High Performance Graphics (2016)
Ulf Assarsson and Warren Hunt (Editors)

Masked Software Occlusion Culling

J. Hasselgren, M. Andersson, T. Akenine-Möller

Intel Corporation

Figure 1: Left: a Minecraft map with 7M triangles accelerated by our software occlusion culling algorithm as seen by the viewer. Middle: a top-view rendering where frustum culled geometry has been removed and the viewer is located in the lower right corner. Non-culled geometry is darker and occlusion culled geometry is brighter. A substantial amount of geometry is occlusion culled before it is sent to the GPU. Right: a visualization of our hierarchical depth representation, where dark is farther away. Our algorithm uses only 4 ms of CPU time on a single thread, even when sending all geometry through our software occlusion culling engine. The Neu Rungholt map is courtesy of kescha.

Abstract

Efficient occlusion culling in dynamic scenes is a very important topic to the game and real-time graphics community in order to accelerate rendering. We present a novel algorithm inspired by recent advances in depth culling for graphics hardware, but adapted and optimized for SIMD-capable CPUs. Our algorithm has very low memory overhead and is $3\times$ *faster than previous work, while culling 98% of all triangles culled by a full resolution depth buffer approach. It supports interleaving occluder rasterization and occlusion queries without penalty, making it easy to use in scene graph traversal or rendering code.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Display Algorithms I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Hidden line/surface removal I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Visible line/surface algorithms

1. Introduction

Environments in modern games and real-time graphics applications are steadily becoming more dynamic and the user is allowed more freedom to interact with the virtual world. While this increases immersion and makes the virtual environment feel more like the real world, it also sets higher demands on data structures and culling algorithms than ever before.

As a consequence, more and more game engines focus less on traditional precomputed visibility determination algorithms, such as potentially visible sets [ARB90, KCCO01, NBG02], and favor algorithms that rasterize significant occluders to a hierarchical depth buffer [GKM93]. Some recent examples of this are the Frostbite engine [And09, Col11], the Umbra occlusion culling engine (`http://umbra3d.com/`) and Intel's software occlusion culling framework [CMK*16], which all rely on software rasterization to create a hierarchical depth buffer. This buffer is then used to perform occlusion queries on the CPU before sending drawcalls to the GPU. Haar and Aaltonen [HA15] propose a similar system with the same components, but operating entirely on the GPU.

While accurate occlusion queries are desirable, most systems are forced to balance performance and accuracy. For example, the work of Andersson [And09] and Collin [Col11] use a very low resolution depth buffer, and compensate by making occlusion meshes inner-conservative. However, creating an inner-conservative mesh is a difficult task. The mesh should be shrunk by the area of one pixel, which in turn depends on projection and can potentially be unbounded. This can lead to *false negatives* or erroneous culling, and puts high requirements on the artists modeling the occlusion meshes. In contrast, Intel's occlusion culling demo uses a full resolution depth buffer, and will therefore not suffer from the same shortcomings. However, performance is likely lower.

Compared to previous approaches, our main contribution is a hierarchical depth representation that efficiently decouples depth and coverage data. This is key for performance as we can very rapidly

compute accurate coverage for a tile, essentially making tiles into our smallest processing unit, rather than pixels as in previous work. This allows us to keep the benefits of using high-resolution depth buffers, while reaching performance similar to that of low resolution [And09, Col11], or even GPU-accelerated approaches [HA15].

Our work is heavily inspired by the recent work by Andersson et al. [AHAM15] on masked depth culling for graphics hardware. We propose the following four extensions to their algorithm to make it more suitable for software implementation.

- An algorithm for generating the coverage mask of a 32×8 pixel tile in parallel using only a few SIMD-instructions.
- A novel depth update heuristic, similar to that of Andersson et al. [AHAM15], but trading accuracy for performance.
- A SIMD-friendly hierarchical depth buffer representation with low memory overhead, tailored for occlusion culling.
- Low cost occluder rendering and occlusion query interleaving, enabling simple and efficient scene graph traversal algorithms.

In addition, we provide an optimized implementation of the algorithm. In Figure 1, we show an example of our algorithm applied to a scene with complex occlusion.

2. Previous Work

For static scenes, one may precompute a potentially visible set (PVS) [ARB90] or use portals and mirrors [LG95], for example, and there is a wealth of research done on this topic. However, we focus here on methods for dynamic occlusion culling since a majority of applications are non-static.

Greene et al. [GKM93, Gre96] presented the first algorithms that used a hierarchical depth-based data structure, and this work has influenced graphics hardware substantially. Zhang et al. [ZMHH97] introduced hierarchical occlusion maps, which also provided approximate occlusion testing due to the way their (full) hierarchy was created. Morein [Mor00] described a hierarchical Z (HiZ) algorithm with just one level in the hierarchy, which gives a number of advantages in terms of hardware implementation. Most graphics processors are likely to have some variant of HiZ occlusion culling in order to improve performance.

Aila and Miettinen [AM04] unified a number of culling methods into a system that could handle massive scenes with dynamic objects. The system, called dynamic potentially visible set (dPVS), appears to have been the basis of Umbra more than 10 years ago. For a while, Umbra supported both GPU and CPU occlusion culling [SSMT11], but it is currently a software-only engine. Bittner et al. [BWPP04] built a system based on hardware occlusion queries, where a front-to-back traversal of the scene provided for better culling together with interleaving of rendering of previously visible objects and queries. The field of hardware occlusion queries has matured in itself by providing predicated rendering (executing a drawcall only if an occlusion query is successful, which avoids costly communication between GPU and CPU), approximate queries (possibly using the GPU's HiZ buffer), and an "any fragments"-optimization which can terminate the query as soon as a first visible fragment is found. In addition to the software occlusion culling work presented in the introduction, Valient [Val11] rendered a full resolution depth buffer and then conservatively scaled it down to accelerate occlusion testing and Persson [Per12] computes convex occlusion volumes from inner conservative occluder boxes, and culls objects that are fully enclosed in such volumes. There are also alternative, approximate algorithms that rely on downsampling and reprojecting the GPU's depth buffer from previous frames [KSS11, HA15] and use hole filling strategies for filling in missing data. These algorithms may erroneously cull visible objects, and rely on relatively small changes in frame-to-frame visibility. Scenes with fast moving dynamic objects are particularly troublesome as they are not handled well by the reprojection.

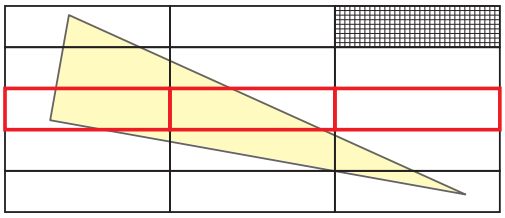

Figure 2: An example triangle being rasterized on an AVX2-capable processor. We traverse all 32×8 pixel tiles overlapped by the triangle's bounding box and compute a 256-bit coverage mask using simple bit operations and shifts. Note that we must consider all three triangle edges in the marked (red) tiles, which overlap the middle vertex (in y), and two edges elsewhere.

3. Algorithm and Implementation

We first implemented our algorithm in Intel's software occlusion culling framework [CMK*16], and will therefore start with a brief description of their system. Their occlusion culling is divided into two main passes. The first pass identifies a set of significant, large occluder meshes, performs basic view frustum and backface culling, and transforms & rasterizes all non-culled triangles to a full-resolution depth buffer. The depth buffer is then reduced by computing the maximum depth for each 8×8 pixel tile, which creates a one-level hierarchical depth buffer [GKM93, Mor00]. The second pass performs occlusion queries in software to determine which objects are visible. The bounding box of each potential occludee is first view frustum culled and then transformed to screen space to form a bounding rectangle with a minimum depth, Z_{min}^{box}. The occlusion query is performed by quickly traversing the bounding rectangle and testing the minimum depth, Z_{min}^{box}, against the relevant depths stored in the hierarchical depth buffer, Z_{max}^{tile}. The object is only classified as occluded if $Z_{min}^{box} > Z_{max}^{tile}$ for all tiles overlapped by the bounding rectangle.

Our rasterization algorithm is similar to any standard two-level hierarchical rasterizer [MM00] with two main exceptions, which will be described in greater detail below. First, we efficiently compute coverage masks from triangle edges for an entire tile in parallel. Since AVX2 supports 8-wide SIMD with 32-bit precision, we have chosen 32×8 as our preferred tile size, which is also shown in Figure 2. As we will see, this allows us to very efficiently compute coverage for 256 pixels in parallel, and it should be easily extendible to 512 pixels given the upcoming AVX-512 instruction set. The second difference is our hierarchical depth buffer representa-

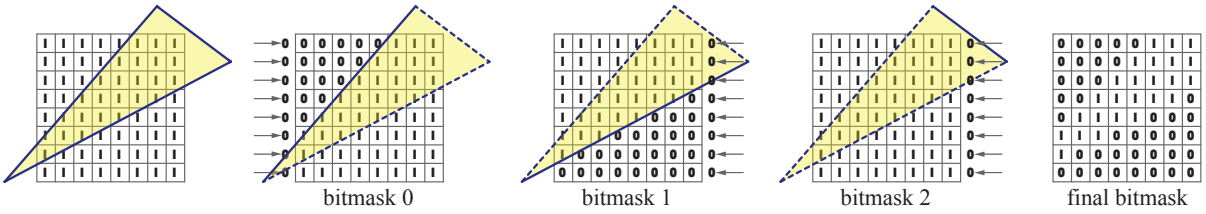

Figure 3: Illustration of how bitshifting is used to create a coverage mask for an 8×8 pixel tile (we use 32×8 in practice). From left to right: 1) For each edge, we start with a bitmask with all bits set to one. 2) The leftmost edge is processed first and zeroes are shifted right until the edge is met. 3) The second edge is processed and zeroes are shifted in from the right side until the edge is met. In practice we accomplish this by shifting in zeroes from the left and inverting the final mask. 4) The tile is fully inside the last edge and no shifting is required. 5) All bitmasks are `and`:ed together to form the final coverage mask.

tion, which bypasses the need for a full-resolution depth buffer, and decouples depth and coverage data. This reduces memory usage by an order of magnitude.

3.1. Efficient Triangle Coverage

Our coverage determination has many similarities with the original edge fill rasterization algorithm [AW80], but is optimized to generate coverage for an entire SIMD-register in a few instructions.

First, looking at the scalar case, we rasterize a triangle by tracking the *left* and *right* events, where the triangle edges intersect each scanline. They can be found by computing $\Delta x/\Delta y$ slopes for each triangle edge, and incrementing the left and right events by the corresponding slope each time a new scanline is traversed. The slope for either the left or right event changes when the scanline coincides with the middle vertex (in y). This has been well-handled by previous work, for example, by splitting the triangle into a top and bottom part [AJ88].

Given the left and right events for a scanline, we create a 32-bit coverage mask for 32 pixels in parallel by pre-loading a register with all bits set, and using shift operations to clear out bits outside the range defined by the left and right events, as shown in the pseudo-code below.

```
// Compute coverage for the 32-pixels at pos. x,
// given the left and right triangle events
function coverage(x, left, right)
  return (~0 >> max(0, left - x))
       & ~(~0 >> max(0, right - x))
```

We let each SIMD-lane operate on a different scanline, effectively computing coverage for 32×8 pixels in parallel. The main challenge in doing so is that while there are only two events per individual scanline (entering and exiting the triangle), all three edges may contribute to the coverage of a tile. Therefore, we must modify our coverage test to the following:

```
// Compute coverage for 32x8 pixel tile. Params
// are SIMD8 registers with 32 bits per lane
function coverageSIMD(x, e0, e1, e2, o0, o1, o2)
    m0 = (~0 >> max(0, e0 - x)) ^ o0;
    m1 = (~0 >> max(0, e1 - x)) ^ o1;
    m2 = (~0 >> max(0, e2 - x)) ^ o2;
    return m0 & m1 & m2;
```

The edge events, `e0-e2`, are similar to the left and right events for the scanline version, and the masks `o0-o2` are used to perform bitwise `not` if an edge is considered a right event. That is, `o1` is `0` for left facing edges and `~0` for right facing edges. This process is visualized in Figure 3.

A triangle will always have at least one left facing and one right facing edge, with the final edge being either left or right facing based on triangle configuration. Thus, two of the `xor`-operations above can be removed by sorting the triangle edges. In practice, we go further than this in optimizing our implementation. The bit inversion can be determined during triangle setup based on the orientation of the remaining edge, and we use different code-paths optimized for the two cases. Furthermore, referring again to Figure 2, we only need to consider all three edges in tiles overlapping the middle vertex (in y). We divide the triangle in top, bottom, and mid-segments, and only perform the full three edge tests in the mid-segment using an optimized two-edge version for the top and bottom part.

Precision Our coverage algorithm does not rely on edge functions [Pin88], and it is therefore difficult to guarantee full compliance with DirectX rasterization rules. For example, we use $\Delta x/\Delta y$ slopes for tracking left and right events, and the resulting slope must be rounded to some finite precision, which introduces a rounding error. The division operation may also lead to precision issues for near-horizontal edges, and this is accentuated in our algorithm due to the large tile size. While inexact rasterization may introduce false positives, we note that most occlusion culling algorithms for real-time applications already tolerate a small margin of error. For example, the occlusion culling demo by Intel [CMK*16] already clamps vertex positions to integer coordinates to ensure that edge functions can be represented using 32-bits. This may introduce an error of up to one pixel, which is similar in size to the error produced by our algorithm.

It is possible to use Bresenham interpolation [Bre65] to create a version of our algorithm that respects DirectX rasterization rules. Bresenham interpolation does not introduce any precision loss and can be used to accurately interpolate edges [LKV90]. We have made an experimental implementation and empirically validated that it matches the GPU's rasterizer for a large set of random triangles. However, we leave a SIMD-optimized implementation for future work, and can therefore make no claims on performance.

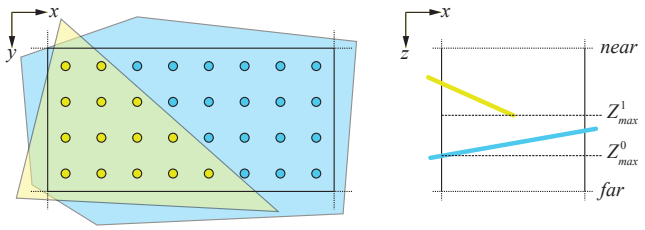

Figure 4: In this example, an 8×4 pixel tile is first fully covered by a blue polygon, which is later partially covered by a yellow triangle. Left: our HiZ-representation seen in screen space, where each sample belongs either to Z^0_{max} or Z^1_{max}. Right: along the depth axis (z), we see that the yellow triangle is closer than the blue polygon. All the yellow samples (left) are associated with Z^1_{max} (working layer), while all blue samples are associated with Z^0_{max} (reference layer).

3.2. Hierarchical Depth Buffer

Our triangle coverage algorithm from the previous section generates a full AVX bitmask, where each 32-bit SIMD-lane corresponds to 32×1 pixels in a 32×8 tile. We use an inexpensive shuffle to rearrange the mask so that each SIMD-lane maps to a more well formed 8×4 tile. For each 8×4 tile, our hierarchical depth buffer stores two floating-point depth values Z^0_{max} and Z^1_{max} and a 32-bit mask indicating which depth value each pixel is associated with. By storing these values as a struct of arrays (SoA) for 4×2 tiles, we can efficiently depth test and update eight tiles in parallel using AVX2 instructions. An example of a populated tile, and its representation can be found in Figure 4.

Depth Buffer Update We need a method to conservatively update our representation each time we rasterize a triangle that (partially) covers a tile. It is possible to directly use the algorithm proposed by Andersson et al. [AHAM15], and we have implemented it as a reference. However, we propose a simpler approach inspired by quad-fragment merging [FBH*10], which is less accurate but more performant. Our heuristic is more sensitive to render order than the original work. However, it is often in the best interest to use a well sorted rendering order in an occlusion culling engine, and we have found it works well in practice.

```
function updateHiZBuffer(tile, tri)
  // Discard working layer heuristic
  dist1t = tile.zMax1 - tri.zMax
  dist01 = tile.zMax0 - tile.zMax1
  if (dist1t > dist01)
    tile.zMax1 = 0
    tile.mask = 0

  // Merge current triangle into working layer
  tile.zMax1 = max(tile.zMax1, tri.zMax)
  tile.mask |= tri.coverageMask

  // Overwrite ref. layer if working layer full
  if (tile.mask == ~0)
    tile.zMax0 = tile.zMax1
    tile.zMax1 = 0
    tile.mask = 0
```

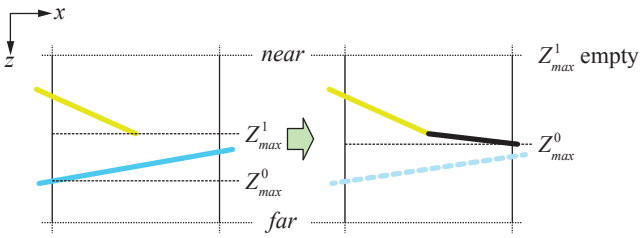

Figure 5: An example, from Figure 4, of when the Z^0_{max} value (reference layer) is updated since the working layer (Z^1_{max}) is covered fully by the yellow and the black triangle.

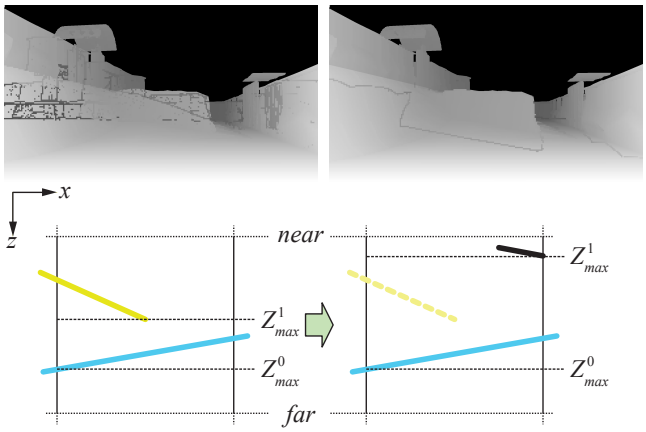

Figure 6: Top: two visualizations of the hierarchical depth buffer. The left image is generated without using a heuristic for discarding layers. Note that the silhouette of background objects leak through occluders. The right image uses our simple heuristic and retains nearly all the occlusion power of an exact depth buffer. Bottom: our discard heuristic applied to the sample tile from Figure 4. The black triangle discards the current working layer, and overwrites the Z^1_{max} value, according to our heuristic. The rationale is that a large discontinuity in depth indicates that a new object is being rendered, and that consecutive triangles will eventually cover the entire tile.

Referring to the pseudo-code, we assign the Z^1_{max} value as the *working layer* and Z^0_{max} as the *reference layer*. After determining triangle coverage, we update the working layer as $Z^1_{max} = \max\left(Z^1_{max}, Z^{tri}_{max}\right)$, where Z^{tri}_{max} is the maximum depth of the triangle within the bounds of the tile, and combine the masks. The tile is covered when the combined mask is full, and we can overwrite the reference layer and clear the working layer. This is illustrated in Figure 5. For details on how to compute accurate triangle depth bounds, we refer to pages 856–857 in the book by Akenine-Möller et al. [AMHH08].

In addition to the rules above, we need a heuristic for when to discard the working layer. This helps preventing silhouettes leaking through foreground occluders, as illustrated in Figure 6. As shown above in the `updateHiZBuffer()` function, we discard the working layer if the distance to the triangle is greater than the distance between the working and reference layers. Our update pro-

	Clear	Geom	Proj	Rast	Gen	Test	Total
HiZ	377	33	163	2145	509	278	3505
Mask	23	33	161	584	0	255	1056

Table 1: Performance breakdown for the first frame of the scene in Figure 7. Time in μs for the algorithmic passes required for generating the hierarchical depth buffer with the different algorithms. Note that the total time for our algorithm is only about 1 ms.

Clear clear the exact depth buffer (HiZ) or masked hierarchical depth buffer (Mask).
Geom select significant occluders and transform vertices to camera space.
Proj per-triangle frustum/backface culling and projection.
Rast triangle setup, rasterization, and depth buffer update (HiZ) or masked hierarchical depth buffer update (Mask).
Gen generate hierarchical depth buffer from per-pixel depth buffer (only required for HiZ).
Test transform, project, and occlusion test all occludee objects.

cedure is designed to guarantee that $Z^0_{max} \geq Z^1_{max}$, so we may use the signed distances for a faster test, since we never want to discard a working layer if the current triangle is farther away. Note that when discarding the working layer, we clear both the mask and the Z^1_{max} value to 0, which may seem counter-intuitive. However, Z^1_{max} is updated using the maximum of Z^1_{max} and Z^{tri}_{max}, which makes initialization to 0 correct.

Hierarchical Depth Test While rasterizing occluders, we may also perform hierarchical depth testing by discarding all tiles where $Z^{tri}_{max} \geq Z^0_{max}$. Our goal in doing so is not to perform occlusion queries, but rather to optimize the rasterizer. For all discarded tiles, we may skip coverage testing and updating the hierarchical depth buffer. Even though these functions are heavily optimized, the hierarchical depth test is simple enough to improve overall performance significantly. Our update guarantees that $Z^0_{max} \geq Z^1_{max}$, and it is therefore sufficient to compare only to Z^0_{max}. This makes our culling test slightly less expensive than the one described by Andersson et al. [AHAM15].

While it seems natural to use hierarchical depth testing, we note that this option is not available to most CPU-based culling frameworks [And09,CMK*16]. Recall that these algorithms compute the hierarchical depth buffer from the full resolution depth buffer by finding the maximum depth in each tile. Doing this while rasterizing occluders would be prohibitively expensive. In contrast, this option is available to us due to the light-weight update operations of the masked depth representation.

Discussion Our hierarchical depth representation is similar to the one proposed by Andersson et al. [AHAM15], but we omit the Z_{min}-value as it is not needed for occlusion queries. It could be useful in a hierarchical culling system for determining if an entire group of objects is completely visible, thereby skipping further occlusion queries. However, occlusion queries are typically inexpensive for visible objects as they may terminate whenever the first visible pixel is found. Therefore, we leave it for future work to determine if it is worthwhile to maintain the Z_{min}-value.

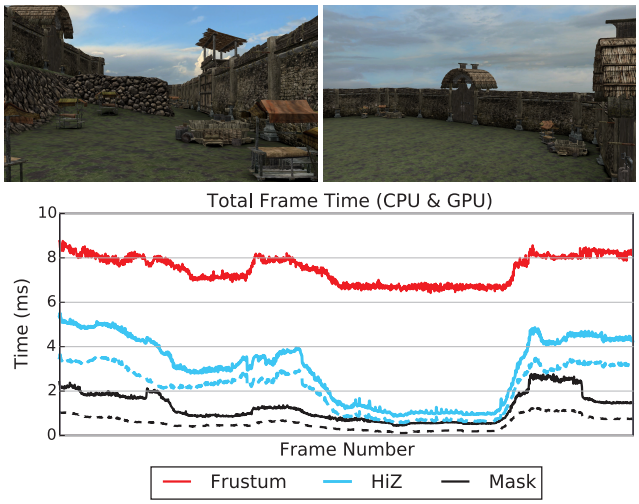

Figure 7: Total frame time and time spent on occlusion culling (dashed lines) for the HiZ and Mask algorithms. For reference and to motivate that culling is indeed beneficial in this application, we also include total frame time with occlusion culling disabled (denoted Frustum). Note that our algorithm is somewhat more conservative than HiZ and culls 2% fewer triangles, but the total performance is still much higher.

4. Results

In this section, we evaluate our algorithm (Mask) and compare performance to the Hierarchical Z buffer algorithm [GKM93] (HiZ). For reference, we also include simple view frustum culling (Frustum) with drawcalls submitted in rough front-to-back order. It represents a GPU limited workload without over-emphasizing pixel shader cost, as the GPU's hierarchical depth test will avoid unnecessary shading in occluded regions.

All measurements were run at 1920×1080 resolution both for rendering and the occlusion buffer, and was made on a machine with an Intel Core i7-4770 processor and a GeForce 760 GTX. We wanted to focus on pure algorithmic performance rather than multithreading, and our algorithm typically runs at real-time rates on a single core. We see this as a great strength as a game engine could dedicate the remaining threads to other jobs, such as AI, collision detection, and physics. With regards to threading, we see no reason why our algorithm would scale worse than any other rasterization based approach, but we leave this evaluation for future work.

We have integrated our algorithm into Intel's software occlusion culling framework [CMK*16], and it is also from their framework we have taken the optimized implementation of HiZ we use as a performance baseline. This is further discussed in Section 4.1. We also use a standalone framework built around the strengths of our occlusion culling algorithm, and this is described in Section 4.2.

4.1. Intel Software Occlusion Culling Framework

The January 2016 version of this framework features a marketplace scene with 2M triangles, and a 49K triangle occlusion mesh. It uses an AVX2-optimized version of the Hierarchical Z buffer algorithm,

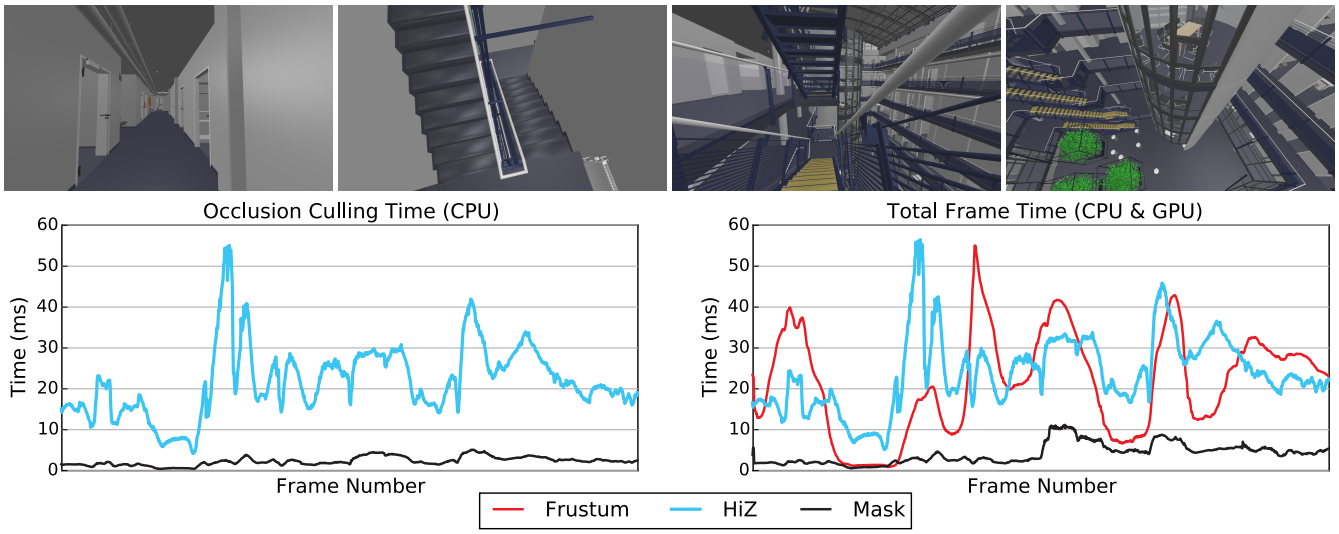

Figure 8: The MPI Informatics building scene (courtesy of Max Planck Institute for Informatics). The graphs show time spent on occlusion culling and total frame time.

and performs occlusion culling in a two step process. The occluder mesh is first rasterized to a full resolution depth buffer, the depth buffer is reduced by computing the maximum depth in each 8×8 pixel tile, and finally occludee objects are occlusion queried against the lower resolution hierarchical depth buffer.

We made two general optimizations to the framework, which we have carried over to all evaluated algorithms. First, we removed the exact bounding box occlusion query, which triangulates and rasterizes each object's transformed bounding box with pixel accuracy. We kept only their coarse test, which traverses an object's screen space bounding rectangle and tests against the hierarchical depth buffer. Our second optimization is that we introduced an axis-aligned bounding box (AABB) tree for the occludees, and perform hierarchical occlusion queries. Many occludees in the scene are very small with each occlusion query representing only a handful of triangles. Rather than modifying the scene, we considered it reasonable to use a tree structure for a workload with this many small individual objects. While these optimizations make the culling test more conservative, and classify an additional 1% of the scene triangles as visible, total frame time is reduced significantly.

In Table 1, we show a timing breakdown of the traditional Hierarchical Z buffer algorithm and our algorithm. As can be seen in the table, our algorithm reduces execution time by almost $4\times$ for the rasterization pass, which is the target of all our algorithmic optimizations. Furthermore, we directly operate on the hierarchical depth representation and can therefore completely skip the pass for generating the hierarchical depth buffer from the regular depth buffer. Typically, this is done by computing the max depth of all pixels in the tile, but we can directly use Z^0_{max} as the conservative max value. Similarly, the cost of clearing the depth buffer is greatly reduced as our hierarchical representation uses $\sim 10\%$ of the storage of the full resolution depth buffer.

Figure 7 shows a breakdown of frame time during a short camera animation. The diagram shows that total frame time is reduced considerably with occlusion culling, compared to rendering all objects inside the view frustum. This is in part due to reduced GPU load, but also the CPU time spent in the rendering code is significantly reduced. Total frame time with occlusion culling enabled is mostly limited by CPU performance, and therefore scales very well for our algorithm. Applications with more complex rendering may be limited by GPU performance, which diminishes the benefits of faster culling. However, we remark that CPU time is often a scarce resource that the occlusion culling algorithm has to share with many other tasks.

4.2. Interleaved Rasterization and Queries

We also implemented our algorithm in a stand-alone framework, where we tailored the traversal and culling algorithm to the strengths of our rasterizer. As shown in Listing 1, we store the scene in an AABB-tree, and use a heap to traverse the nodes in approximate front-to-back order. During traversal, our code performs frustum and occlusion queries to terminate traversal early.

While this may seem like a textbook example of an occlusion culling system, we remark that there has traditionally been issues with integrating traversal and occlusion culling this closely. For software algorithms, such as `HiZ`, the overhead of generating the hierarchical depth buffer is typically very large (see Table 1) which makes it impractical to interleave rasterization and occlusion queries. Similarly, for GPU algorithms, latency is much too large to use hardware occlusion queries to guide fine-grained per-node traversal decision, as this typically leads to synchronization issues and stalling.

Our algorithm both rasterizes occluders and performs occlusion queries directly to the hierarchical depth buffer. We therefore do not incur any penalty from interleaving the two operations, which makes fine-grained traversal algorithms simple to implement. As we intend to show with our results, this is beneficial as we need

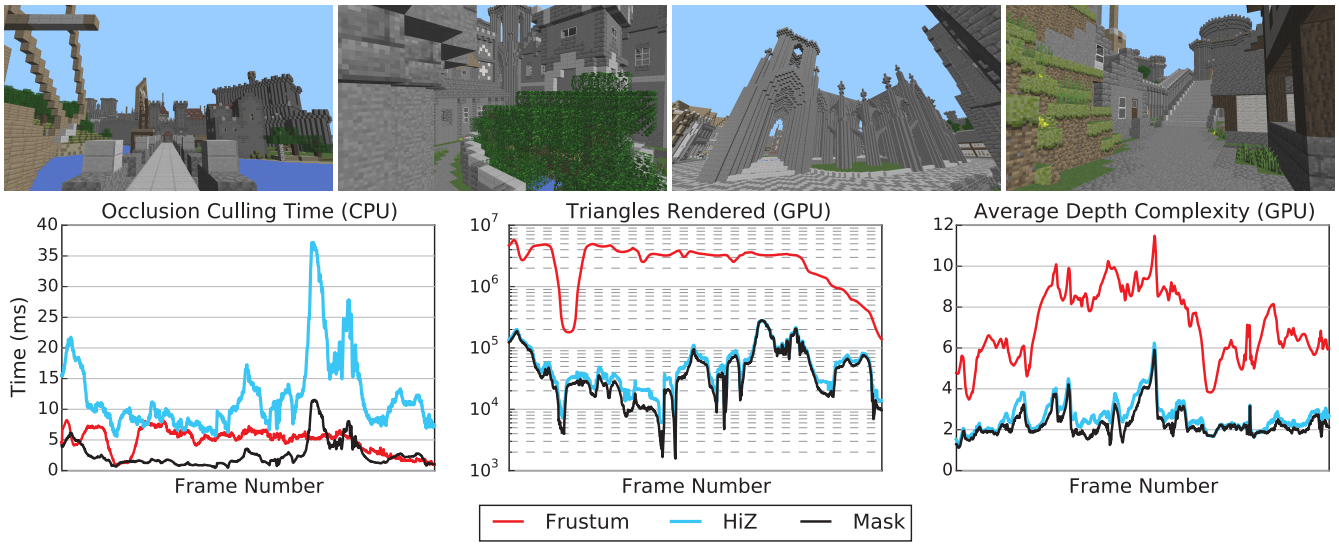

Figure 9: Our second test scene is a Minecraft map named Neu Rungholt. The left diagram shows CPU time spent on occlusion culling, but we have also added the total frame time for frustum culling into the same graph. The curves are directly comparable because the occlusion culling algorithms efficiently remove almost all GPU work, making rendering CPU-bound. The diagram in the middle shows the number of triangles submitted to the GPU and the diagram to the right shows the average depth complexity of those triangles.

Listing 1: Traverse and cull in approximate front-to-back order.

```
function traverseSceneTree(worldToClip)
  heap = rootNode
  while !heap.empty():
    node = heap.pop()
    if node.isLeaf():
      rasterizeOccluders(node.triangles)
      node.visible = true
    else:
      for c in node.children:
        culled = frustumCull(c.AABB)
        clipBB = transform(worldToClip, c.AABB)
        rect = screenspaceRect(clipBB)
        culled |= isOccluded(rect, clipBB.minZ)
        if !culled:
          heap.push(c, clipBB.minZ)
```

only perform work proportional to the number of non-occluded triangles.

Our first test scene is shown Figure 8, along with occlusion culling performance and total rendering time for a short camera animation. The scene contains a total of 73M triangles, but for occlusion we only use the architectural mesh with 143K triangles. This scene is considerably more complex than the one used in Intel's demo and the occluder mesh contains many large but sliver triangles, which are difficult to rasterize efficiently as they typically have large screen space bounding rectangles. Still, we see that our algorithm performs very well, having by far the best *worst case* frame time, and only being outperformed by frustum culling in some very rare cases which are already running at very high framerates.

Note that we include the `HiZ` algorithm for reference. This comparison was particularly difficult as we needed to balance the cost of generating the hierarchical depth buffer with the gains of performing fine grained occlusion culling (see Listing 1). We wanted to keep `HiZ` as similar to the original implementation as possible, so we decided to re-generate the hierarchical depth buffer every $N = 100$ rasterized leaves as this gave the best overall performance for both test scenes. It may not be fair to claim a $10\times$ speed up for our algorithm, but we conclude that it is very robust to complex occluder meshes, and that the interleaved traversal algorithm is very beneficial in this scene.

Our second scene, Rungholt, shown in Figure 9, uses the entire 7M triangle mesh both for rendering and occlusion culling, which makes it our most geometrically complex scene from an occlusion culling standpoint. For this scene, it is very difficult to compete with the raw GPU performance of simple frustum culling. We use a simple surface shader, and the scene only contains one material and texture, which makes it possible to submit everything to the GPU without any shader/texture/state changes. We note that our algorithm still performs very well, typically using well below 5 ms running on a single core, but is outperformed by frustum culling for some difficult camera positions. We find these results encouraging as we use the same assets and are essentially keeping even steps with a discrete GPU using only a single CPU core, granted that we only need to rasterize an approximate (but conservative) hierarchical depth buffer and may exploit the scene data structure to do early outs. It is possible to change the balance of this scene by making pixel or vertex shaders more expensive (for example by adding more light sources) or adding more state changes, and we can easily tweak it to produce more convincing results but we opted to keep it unmodified for fairness.

We believe that the gains of occlusion culling would likely be

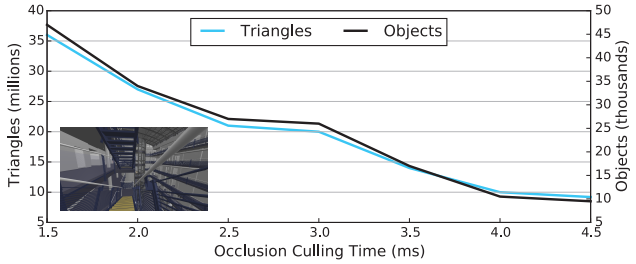

Figure 10: We limit the time spent on occlusion culling for one of the more complex frames of the MPI Informatics building scene, and study how many triangles and objects are classified as visible.

larger in a complex game engine where we could cull shader/texture/state changes and all CPU overhead related to managing scene assets, as in Figure 7. Therefore, we also present the number of triangles submitted to the GPU (see Figure 9) as an indication of the workload. Note that this value is just an effect of the algorithms, and independent of their execution time. Recall that the `HiZ` only regenerates the hierarchical depth buffer periodically, which explains why our algorithm culls more triangles for this scene.

Limited Time Budget Occlusion culling is often accelerated by picking significant occluders using some heuristic approach. Our traversal is sorted in front-to-back order, and it is therefore reasonable to assume that the most significant occluders will be rasterized first. If only a fixed amount of CPU time can be spent on culling, then we periodically poll the system clock, and simply stop rasterizing occluders if a given threshold has been exceeded. Occlusion queries will still be performed, which makes it hard to guarantee total time spent on occlusion culling, but it is still a powerful way to balance the cost of the CPU and GPU workloads. Figure 10 shows the correlation between time spent on occlusion culling and culling efficiency for the MPI Informatics building scene.

4.3. Scaling

We evaluate how our algorithm scales for various triangle sizes using a synthetic benchmark which rasterizes 32k right-angled isosceles triangles with randomized position and orientation. The triangles are rendered back-to-front to make sure no fragments are culled by our early depth test, to simplify comparison against `HiZ`. As shown in Figure 11, our algorithm scales well for large triangles, while approaching the performance of `HiZ` for very small triangles. This is expected and we argue that it is a good trade-off for occlusion culling. It is relatively easy to create low polygon meshes and choose only the most significant occluders, and most game engines already do this as an optimization. In contrast, it is much more difficult to guarantee correctness when using lower resolution render targets. The second diagram of the figure shows CPU cycles per rasterized pixel.

5. Conclusion

We have shown that the masked depth culling algorithm [AHAM15] can be adapted to run efficiently on modern CPUs. The evaluation shows that our algorithm is extremely

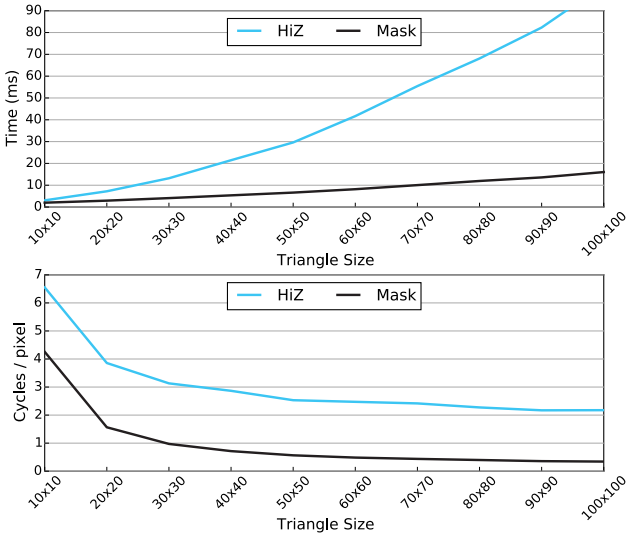

Figure 11: Top: CPU time for rasterizing 32k random occluder triangles with varying size. This includes transform, projection, clipping, backface culling, triangle setup, and rasterization. The two algorithms begin to converge for very small triangles, but our algorithm scales considerably better for large triangles. Bottom: the same data normalized to CPU clock cycles per rasterized pixel.

efficient both in terms of performance and culling accuracy, and we hope that it will inspire future rendering engines. Our algorithm gives most of the benefits of rendering a full resolution depth buffer, such as not requiring the occlusion meshes to account for pixel and tile sizes to give a conservative result, or require hole-filling heuristics to account for conservative rasterization. At the same time, we retain much of the performance characteristics of algorithms using low resolution depth buffers, as we can compute coverage and update the hierarchical depth buffer for 256 pixels in parallel on AVX2-capable machines.

We hope that our work will inspire future research into conservative but lossy culling algorithms that retain most of the efficiency of exact algorithms. Future work may further improve on the hierarchical depth buffer update heuristics to evaluate if depth buffer quality may be improved without significantly impacting performance. Similarly, it is questionable if full 32-bit floating point precision is really needed to represent the Z^i_{max} reference values, and it may be possible to improve performance further by packing them as two 16-bit values instead. It would also be interesting to evaluate the usefulness of a GPU implementation. Laine and Karras [LK11] showed that it is extremely difficult to compete with the fixed-function rasterizer. However, in systems where culling and visible surface determination is performed using compute shaders [HA15], there may be some merit in keeping the occlusion culling in the same kernel.

Acknowledgements

We thank Intel's Advanced Rendering Technology team. We also thank David Blythe and Chuck Lingle for supporting this research.

The Neu Rungholt map is courtesy of kescha, and exported by Morgan McGuire. The MPI Informatics building mesh [HZDS09] is courtesy of Max Planck Institute for Informatics.

References

[AHAM15] ANDERSSON M., HASSELGREN J., AKENINE-MÖLLER T.: Masked Depth Culling for Graphics Hardware. *ACM Transactions on Graphics 34*, 6 (2015), 188:1–188:9. 2, 4, 5, 8

[AJ88] AKELEY K., JERMOLUK T.: High-performance Polygon Rendering. In *Proceedings of SIGGRAPH 88* (1988), ACM, pp. 239–246. 3

[AM04] AILA T., MIETTINEN V.: dPVS: An Occlusion Culling System for Massive Dynamic Environments. *IEEE Computer Graphics and Applications, 24*, 2 (2004), 86–97. 2

[AMHH08] AKENINE-MÖLLER T., HAINES E., HOFFMAN N.: *Real-Time Rendering*, 3rd ed. AK Peters Ltd., 2008. 4

[And09] ANDERSSON J.: Parallel Graphics in Frostbite - Current & Future. SIGGRAPH Course: Beyond Programmable Shading, http://s09.idav.ucdavis.edu/talks/04-JAndersson-ParallelFrostbite-Siggraph09.pdf, 2009. 1, 2, 5

[ARB90] AIREY J. M., ROHLF J. H., BROOKS JR. F. P.: Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. In *Symposium on Interactive 3D Graphics* (1990), ACM, pp. 41–50. 1, 2

[AW80] ACKLAND B., WESTE N.: Real Time Animation Playback on a Frame Store Display System. In *Proceedings of SIGGRAPH 80* (1980), ACM, pp. 182–188. 3

[Bre65] BRESENHAM J. E.: Algorithm for Computer Control of a Digital Plotter. *IBM Systems Journal 4*, 1 (1965), 25–30. 3

[BWPP04] BITTNER J., WIMMER M., PIRINGER H., PURGATHOFER W.: Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. *Computer Graphics Forum, 23*, 3 (2004), 615–624. 2

[CMK*16] CHANDRASEKARAN C., MCNABB D., KUAH K., FAUCONNEAU M., GIESEN F.: Software Occlusion Culling. Published online at: https://software.intel.com/en-us/articles/software-occlusion-culling, 2013–2016. 1, 2, 3, 5

[Col11] COLLIN D.: Culling the Battlefield. Game Developer's Conference (presentation), 2011. 1, 2

[FBH*10] FATAHALIAN K., BOULOS S., HEGARTY J., AKELEY K., MARK W. R., MORETON H., HANRAHAN P.: Reducing Shading on GPUs using Quad-Fragment Merging. *ACM Transactions on Graphics, 29*, 4 (2010), 67:1–67:8. 4

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical Z-Buffer Visibility. In *Proceedings of SIGGRAPH 1993* (1993), ACM, pp. 231–238. 1, 2, 5

[Gre96] GREENE N.: Hierarchical Polygon Tiling with Coverage Masks. In *Proceedings of SIGGRAPH 96* (1996), ACM, pp. 65–74. 2

[HA15] HAAR U., AALTONEN S.: GPU-Driven Rendering Pipelines. SIGGRAPH Advances in Real-Time Rendering in Games course, 2015. URL: http://advances.realtimerendering.com/s2015/aaltonenhaar_siggraph2015_combined_final_footer_220dpi.pdf. 1, 2, 8

[HZDS09] HAVRAN V., ZAJAC J., DRAH. J., SEIDEL H.: *MPII Building Model as Data for Your Research*. Res.rep. MPI-I-2009-4-004, MPI Informatik, 2009. 9

[KCCO01] KOLTUN V., CHRYSANTHOU Y., COHEN-OR D.: Hardware-Accelerated From-Region Visibility Using a Dual Ray Space. In *Proceedings of Eurographics Workshop on Rendering Techniques* (2001), pp. 205–216. 1

[KSS11] KASYAN N., SCHULZ N., SOUSA T.: Secrets of CryENGINE 3 Graphics Technology. SIGGRAPH Advances in Real-Time Rendering in 3D Graphics and Games course, 2011. URL: http://www.crytek.com/download/S2011_SecretsCryENGINE3Tech.ppt. 2

[LG95] LUEBKE D., GEORGES C.: Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. In *Symposium on Interactive 3D Graphics* (1995), ACM, pp. 105–106,212. 2

[LK11] LAINE S., KARRAS T.: High-Performance Software Rasterization on GPUs. In *High-Performance Graphics* (2011), pp. 79–88. 8

[LKV90] LATHROP O., KIRK D., VOORHIES D.: Accurate Rendering by Subpixel Addressing. *IEEE Computer Graphics and Applications 10*, 5 (Sept 1990), 45–53. 3

[MM00] MCCORMACK J., MCNAMARA R.: Tiled Polygon Traversal using Half-Plane Edge Functions. In *Graphics Hardware* (2000), pp. 15–21. 2

[Mor00] MOREIN S.: ATI Radeon HyperZ Technology. In *Graphics Hardware, Hot3D Proceedings* (2000). 2

[NBG02] NIRENSTEIN S., BLAKE E., GAIN J.: Exact From-region Visibility Culling. In *Eurographics Workshop on Rendering* (2002), pp. 191–202. 1

[Per12] PERSSON E.: Creating Vast Game Worlds: Experiences from Avalanche Studios. In *ACM SIGGRAPH 2012 Talks* (2012). 2

[Pin88] PINEDA J.: A Parallel Algorithm for Polygon Rasterization. In *Computer Graphics (Proceedings of SIGGRAPH 88)* (1988), vol. 22, ACM, pp. 17–20. 3

[SSMT11] SILVENNOINEN A., SOININEN T., MÄKI M., TERVO O.: Occlusion Culling in Alan Wake. In *ACM SIGGRAPH Talks* (2011), ACM, pp. 47:1–47:1. 2

[Val11] VALIENT M.: Practical Occlusion Culling in KILLZONE 3. In *ACM SIGGRAPH 2011 Talks* (2011). 2

[ZMHH97] ZHANG H., MANOCHA D., HUDSON T., HOFF III K. E.: Visibility Culling Using Hierarchical Occlusion Maps. In *Proceedings of SIGGRAPH 97* (1997), pp. 77–88. 2