Merging Masked Occlusion Culling Hierarchical Buffers for Faster Rendering

By Leigh Davies, Senior Application Engineer, and Filip Strugar, Senior Graphics Software Engineer

---

Abstract

Efficient occlusion culling in dynamic scenes can accelerate rendering, which makes it an essential topic for the game and real-time graphics community. *Masked Software Occlusion Culling*, the paper published by J. Hasselgren, M. Andersson and T. Akenine-Möller, presented a novel algorithm optimized for SIMD-capable CPUs that culled 98 percent of all triangles culled by a traditional occlusion culling algorithm. While highly efficient and accurate for many use cases, there were still some issues that the heuristics didn't adequately solve. Here, we present an addition to the preceding work by Andersson et al. that addresses many of these problem cases by splitting a scene into multiple buffers that better fit local dynamic ranges of geometry and that can be computed concurrently. We then augment the algorithm's discard heuristics and combine the partial result buffers into a new hierarchical depth buffer, on which applications can perform reliably accurate, efficient occlusion queries.

Introduction

Masked Software Occlusion Culling was invented by—J. Hasselgren, M. Andersson, and T. Akenine-Möller of Intel—in 2015. It was designed for efficient occlusion culling in dynamic scenes suitable for the game and real-time graphics community. The benefit of the Masked Software Occlusion Culling algorithm subsequently proposed by Andersson, Hasselgren, and Akenine-Möller in 2016 was that it culled 98 percent of all triangles culled by a traditional occlusion culling algorithm, while being significantly faster than previous work. In addition, it still takes full advantage of single instruction, multiple data (SIMD) instruction sets and, unlike graphics processing unit (GPU)-based solutions, didn't introduce any latency into the system. This is important to game-engine developers, as it can free the GPU from needlessly rendering non-visible geometry, and it could instead render other, richer game visuals.

Figure 1: *Left: A visualization of the original Masked Occlusion hierarchical depth representation for the Intel® castle scene, where dark is farther away; conservative merging errors are highlighted in red. Middle: The in-game view of the castle scene, including bounding boxes for meshes. Right: A visualization of the Masked Occlusion hierarchical depth representation for the Intel castle scene using the merging of two separate hierarchical buffers, with significantly improved accuracy.*

An updated version [HAAM16] of the algorithm inspired by quad-fragment merging [FBH∗10], which is less accurate but performs better, was also added to the Masked Occlusion library. This approach works best if the incoming data is roughly sorted front to back, which also improves efficiency by reducing overdraw in the depth buffer.

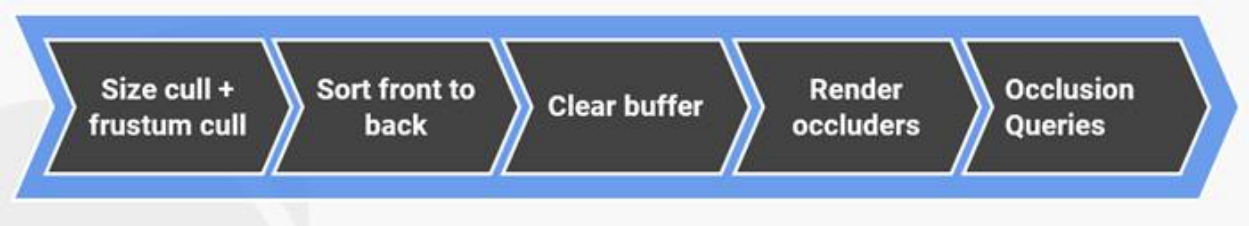

Figure 2: *Typical workflow for Masked Occlusion Culling.*

A typical workflow for integrating Masked Occlusion Culling into a game engine is shown in Figure 2. This workflow mirrors the traditional graphics pipeline and has been used in this format by developers—including Booming Games in their title *Conqueror's Blade**—with good results.

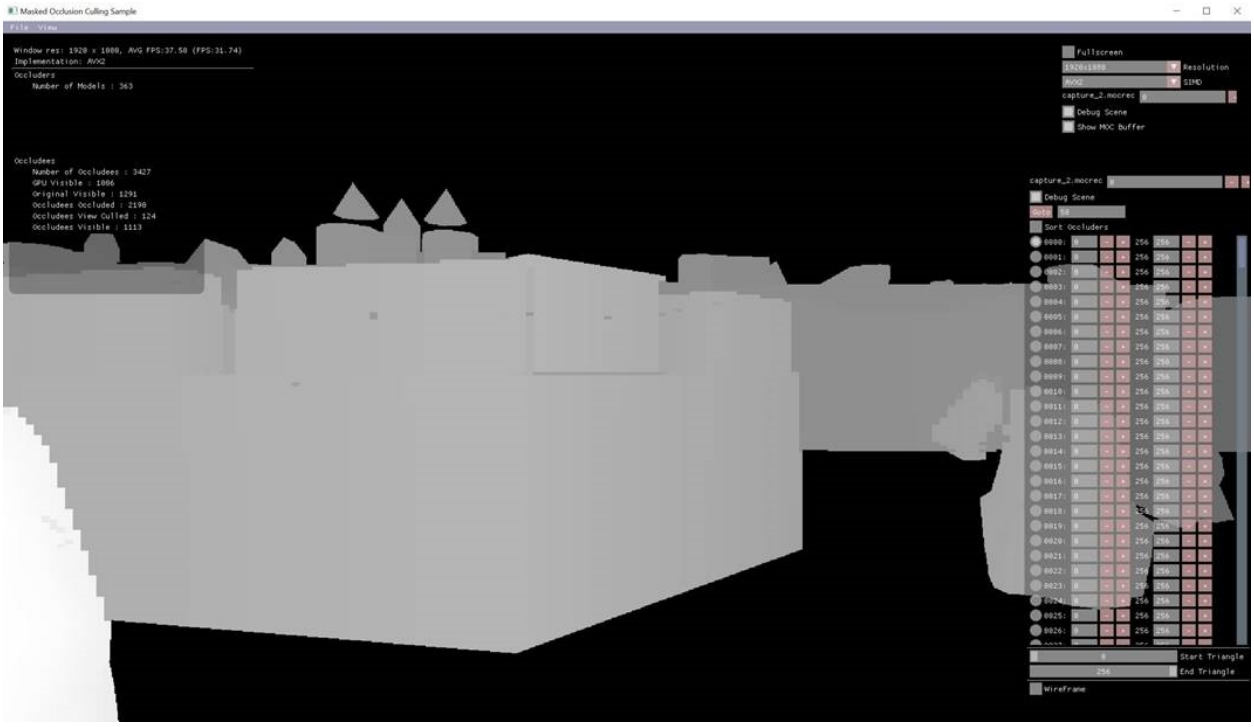

Figure 3: *Typical Masked Occlusion Culling buffer for* Conqueror's Blade*.*

However, some game workloads showed that one area where Masked Occlusion Culling proved less effective was when rendering very large meshes with significant depth overlap, as the overlap made it impossible to do accurate sorting. This proved particularly problematic for the updated [HAMM16] algorithm. Specifically, the issues manifested when rendering a mixture of foreground assets and terrain patches for expansive landscapes. A single terrain patch covered a very wide depth range and couldn't be sorted relative to the foreground occluders in an optimal order. These discontinuities are inherent in the Masked Software Occlusion HiZ buffer creation, as the current heuristics used for discarding layers while constructing the buffer did not have enough context regarding future geometry to keep the most important data. Without the full context of the incoming geometry, the heuristics have to take a conservative approach during depth selection, which increases the number of later occlusion queries that return visible. This, in turn, means the GPU has to render geometry that eventually is culled by the GPU, and never contributes to the overall scene.

To solve this problem, we [authors Leigh Davies and Filip Strugar] have added the functionality to merge multiple Masked Occlusion hierarchical depth buffers in the Masked Software Occlusion library. This allows the developer to utilize a strategy of subgrouping scene geometry and computing partial results buffers for each subgroup. Subgroups are chosen for their tighter dynamic range of depth values, as well as for geometry sorting behavior. A subgroup of foreground objects, and another subgroup of terrain objects, is a common situation. The partial occlusion results for such subgroups is merged later into a single hierarchical depth buffer. This merging of the partial buffers uses an extension of the existing discard heuristic for combining layers.

Previous Work

The Masked Software Occlusion rasterization algorithm is similar to any standard two-level hierarchical rasterizer [MM00]. The general flow of the rasterization pipeline is shown in Figure 4:

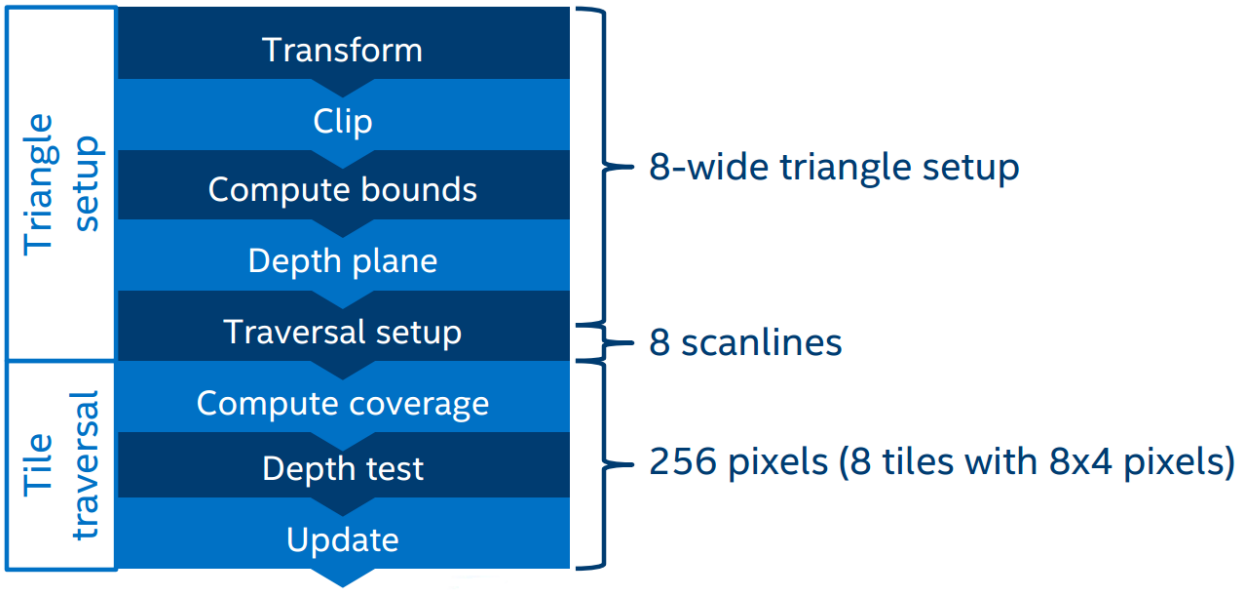

Figure 4: *Masked Occlusion Rasterization Pipeline, shown for Intel® Advanced Vector Extensions 2 (Intel® AVX2).*

Both the triangle setup and the tile traversal code have been heavily optimized to use SIMD, with the number of triangles and pixels that can be processed in parallel varying, depending on the flavor of SIMD being used. There are two main exceptions where the Masked Occlusion Culling algorithm differs from a standard software rasterizer, which are described below. First, rather than process a scanline at a time, it instead efficiently computes a coverage mask for an entire tile in parallel, using the triangle edges.

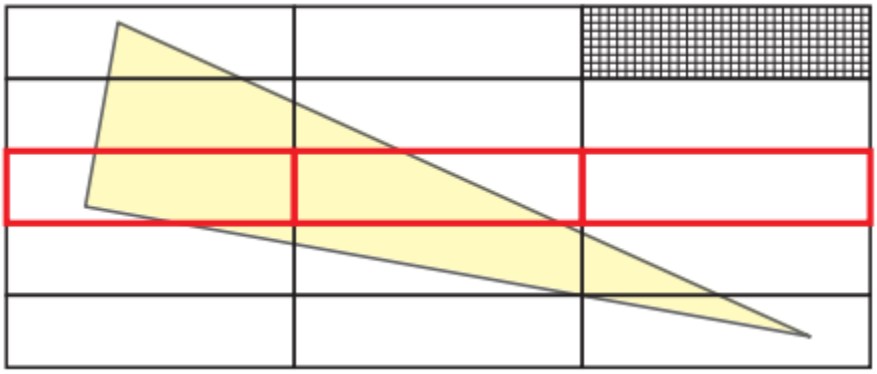

Figure 5: *An example triangle rasterized on an Intel® AVX2 capable processor. We traverse all 32 x 8 pixel tiles overlapped by the triangle's bounding box and compute a 256-bit coverage mask using simple bit operations and shifts.*

Since Intel AVX2 supports 8-wide SIMD with 32-bit precision, we use 32 x 8 as our tile size, as shown in Figure 5 (tile sizes will be different for Intel® Streaming SIMD Extensions 2 (Intel® SSE2/Intel SSE4.1/Intel® Advanced Vector Extensions 512 (Intel® AVX-512) implementations). This allows the algorithm to very efficiently compute coverage for 256 pixels in parallel.

The second difference is the hierarchical depth buffer representation, which decouples depth and coverage data, bypassing the need to store a full resolution depth buffer. The Masked Software Occlusion rasterization algorithm uses an inexpensive shuffle to rearrange the mask so that each SIMD-lane maps to a more well-formed 8 x 4 tile. For each 8 x 4 tile, the hierarchical depth buffer stores two floating-point depth values $Zmax^0$ and $Zmax^1$, and a 32-bit mask indicating which depth value each pixel is associated with. An example of a tile populated by two triangles using the Masked Occlusion algorithm can be found in Figure 6.

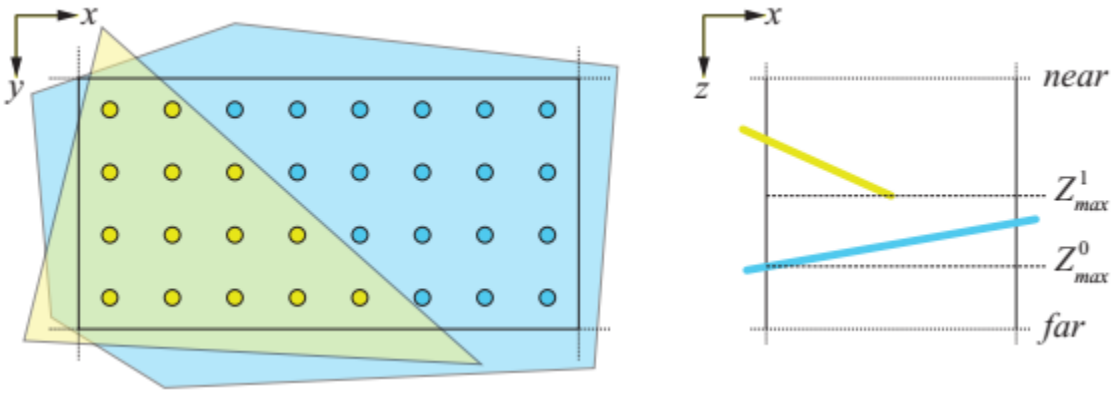

Figure 6: *In this example, an 8 x 4 pixel tile is first fully covered by a blue polygon, which is later partially covered by a yellow triangle. Left: our HiZ-representation seen in screen space, where each sample belongs either to $Zmax^0$ or $Zmax^1$. Right: along the depth axis (z), we see that the yellow triangle is closer than the blue polygon. All the yellow samples (left) are associated with $Zmax^1$ (working layer), while all blue samples are associated with $Zmax^0$ (reference layer).*

Limitation of a Single Hierarchical Depth Buffer

Given that we store only two depth values per tile, we require a method to conservatively update the representation each time a triangle is rasterized that—partially—covers a tile. Referring to the pseudo-code in Figure 7, we begin by assigning the $Zmax^0$ as the reference layer representing the furthest distance visible in the tile, and $Zmax^1$ value as the working layer that's partly covered by triangle data.

After determining triangle coverage, we update the working layer as $Zmax^1 = \max(Zmax^1, Zmax^{tri})$, where $Zmax^{tri}$ 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.

```
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
```

Figure 7: *Update tile pseudo code.*

In addition to the rules above, we need a heuristic for when to discard the working layer. This helps prevent the silhouettes of existing data in the buffer leaking through occluders that are rendered nearer the camera if the data is not submitted in a perfect front-to-back order, as illustrated in Figure 8. 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.

The Masked Occlusion update procedure is designed to guarantee that $Zmax^0 \geq Zmax^1$, 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. 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. If the working layer isn't covered, the algorithm has still honored the requirement to have a conservative representation of the depth buffer.

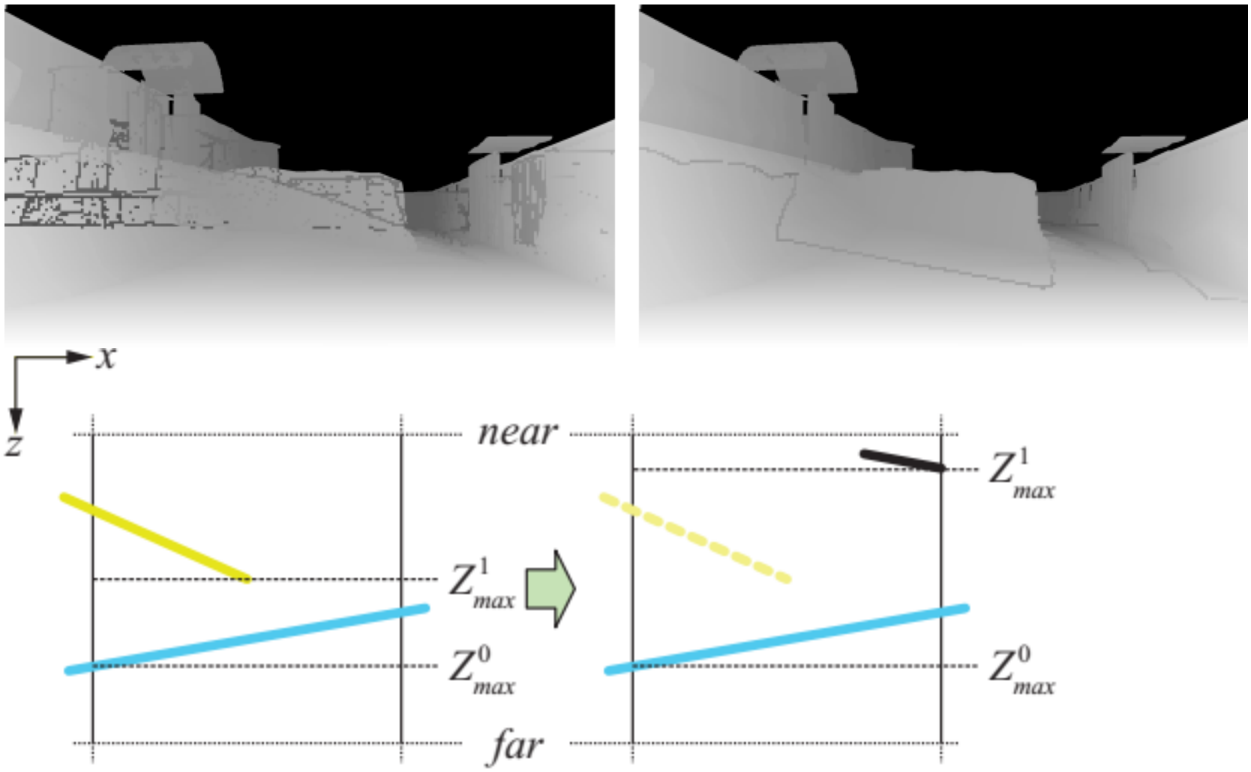

Figure 8: *Top: two visualizations of the hierarchical depth buffer. The left image is generated without using a heuristic for discarding layers. Note that the silhouettes of background objects leak through occluders, appearing as darker gray outlines on the lighter gray foreground objects. The right image uses our simple layer discard heuristic and retains nearly all the occlusion accuracy of a conventional depth buffer. Bottom: our discard heuristic applied to the sample tile from Figure 2. The black triangle discards the current working layer, and overwrites the Z^1max 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 Figure 9, the leaking of remaining silhouette edges through occluders happens because the heuristic for discarding working layers is not triggered, since the reference layer is a long way behind the working layer. In the problem case, the reference layer contains the clear value, resulting in a wide dynamic range to the depth values. The working layer is updated with the most conservative value from the working layer and the new triangle. This is in spite of the fact that consecutive triangles do eventually cover the entire tile, and the working layer could have used the nearer Zmax value from the incoming triangles.

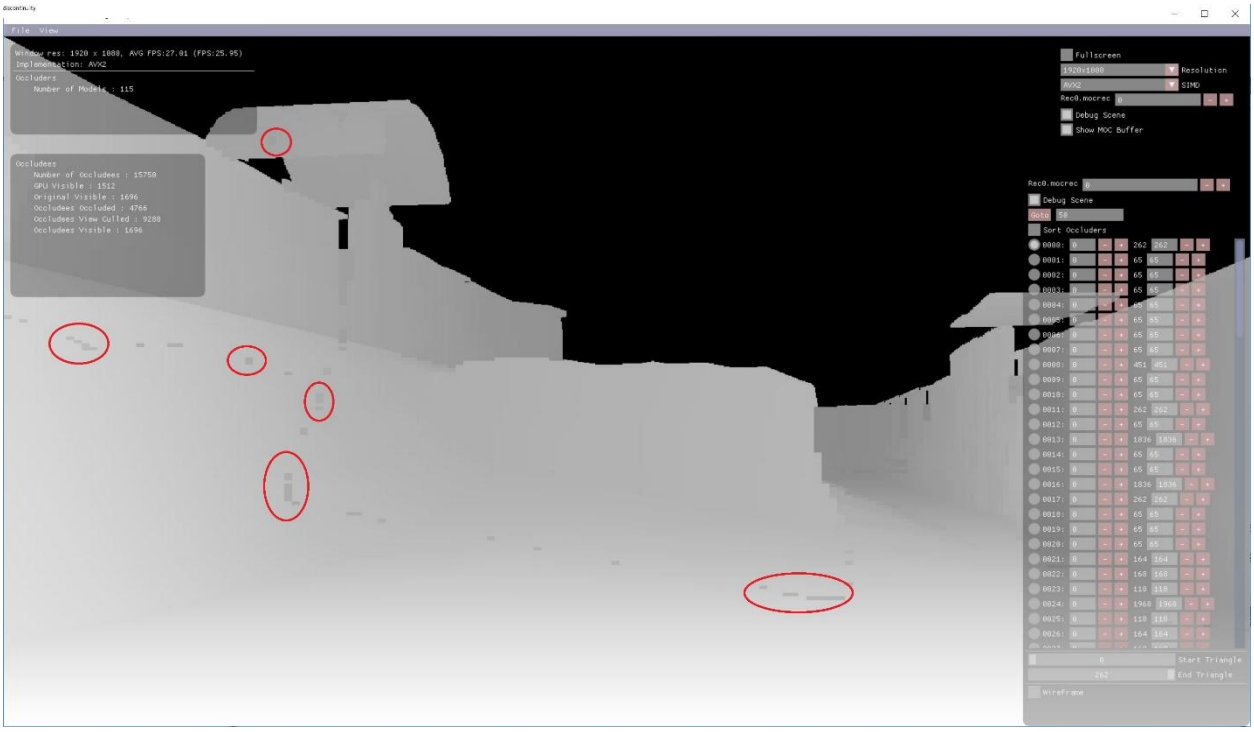

Figure 9: *Silhouette bleed-through in a well-sorted scene using only a single hierarchical depth buffer.*

Removing Silhouette Bleed-through on Terrain

The final effects of silhouette bleed-through at high resolution is shown in Figure 9. The most prominent case of silhouette bleed-through occurs when an object nearer to the viewer is rendered in a tile containing a distant object and the clear color. It is caused by the need to keep the most conservative value in the tile layer without context of what else may be rendered later. One potential solution would be to only merge data into a tile when we have the full data of the new mesh being added, but that would require being able to store more layers in the hierarchical depth buffer.

An alternative way to solve silhouette bleeding is to render the terrain and the foreground objects into their own hierarchical depth buffers, and render them individually. This significantly reduces discontinuities in the hierarchical depth buffers. The foreground objects have almost no bleeding, as the rough front-to-back sort is enough to ensure triangles are rendered in an order that is optimal for the Masked Occlusion discard algorithm. The landscape has some bleeding issues due to internal sorting of triangles within the single mesh, but these are much more localized. The separated hierarchical depth buffers are shown in Figure 10. That just leaves the problem of merging the terrain mesh and the existing foreground objects to generate a final hierarchical depth buffer that may be used for culling.

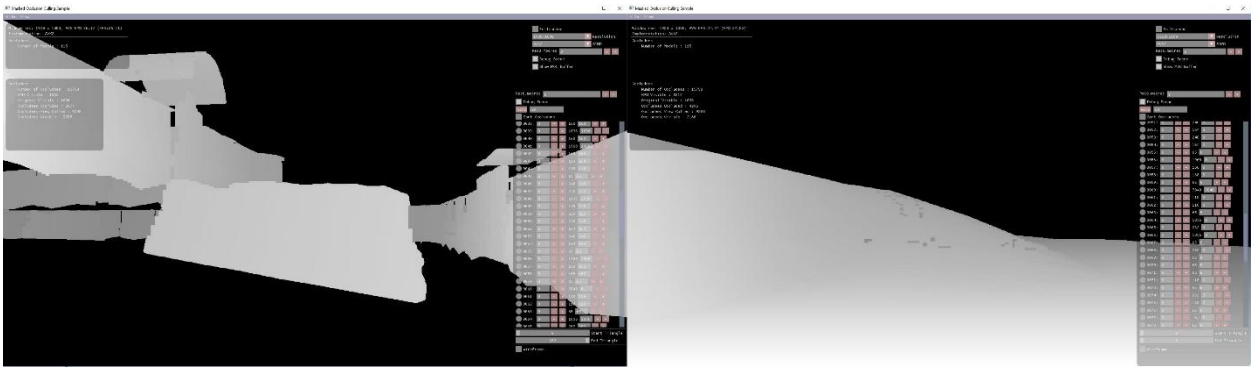

Figure 10: *Castle scene with terrain and foreground objects in separate buffers.*

Merging the Buffers

The new Merge function added to the Masked Occlusion API does just that, taking a second hierarchical depth buffer and merging the data onto the first depth buffer. The merging algorithm works at the same 8 x 4 tile basis and uses the same SIMD instructions as the merging heuristic used for rasterization, allowing multiple subtiles to be processed in parallel. The merge code has been implemented for both the original merge algorithm [AHAM15] and the updated version [HAAM16]. The flow of the merge algorithm is described in Figure 11.

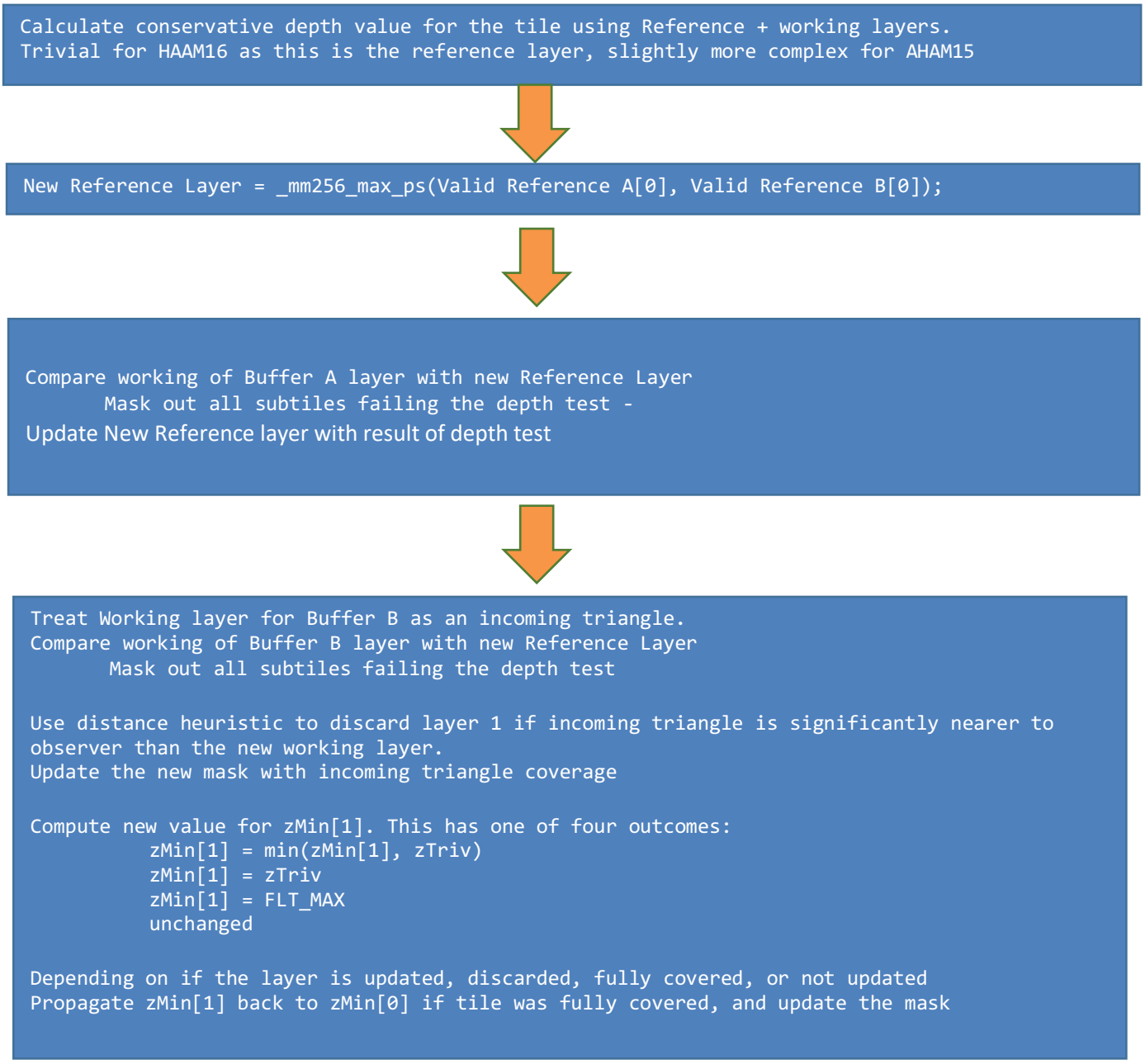

Figure 11: *Flow chart of the new HI-Z merge algorithm*.

In practice, we found that for a scene like the castle, we only need to use two layers; with minor modifications, the code could combine multiple buffers. The final merged buffer is shown in Figure 12. The original silhouette issues have been solved completely.

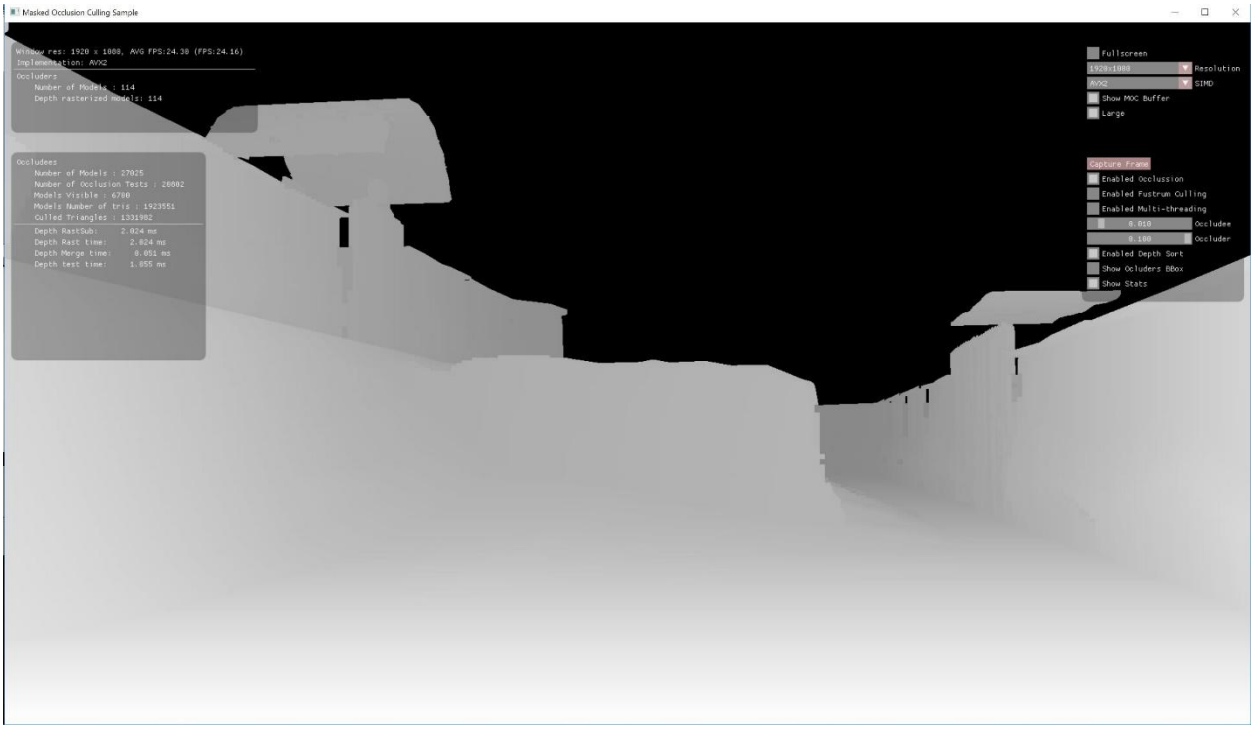

Figure 12: *Visual representation of the final hierarchical depth buffer created from the merged foreground and terrain buffers.*

The merging of buffers does not solve bleed issues local to an individual buffer—such as the internal issues on the terrain—but it does ensure they don't propagate forward if covered by data in another set of geometry. In the case of the castle, they are covered by the foreground objects.

Table 1 shows the performance of the merge algorithm at different resolutions; performance scales with resolution, while the code for merging the HI-Z data uses the same set of SIMD instructions used for the rest of the MOC algorithm. The data presented here was generated using Intel AVX2, thereby processing 8 subtiles in parallel, but this can be expanded with Intel AVX-512 to up to 16 subtiles.

	Terrain Rasterization Time (ms)	Foreground Rasterization Time (ms)	Total Rasterization Time (ms)	Merge Time (ms)	% Time Spent in Merge
640 x 400	0.19	0.652	0.842	0.008	0.9%
1280 x 800	0.26	0.772	1.068	0.027	2.5%
1920 x 1080	0.31	0.834	1.144	0.051	4.4%

Table 1: *Merge cost relative to resolution; performance measured on Intel® Core™ i7-6950X processor.*

The merge function skips tiles that don't require a merge, as only one buffer has valid data in the tile as an optimization. Although there is a fixed cost for traversing the data set to check this, the low memory footprint for the Masked Occlusion hierarchical depth buffer is the primary reason for the performance of the merge. At 1920 x 1080 resolution, the screen consists of 64800 (8 x 4) subtiles that require only 12 bytes of storage per subtile. The merge function only needs to read 760 KB, compared to over 8.1 MB for a traditional 32-bit depth buffer. Additionally, by using Intel AVX2, we are able to process eight sub-tiles in parallel. The timing in Table 1 refers to single-threaded performance. Experiments on threading

the merge showed only minimal benefits due to hitting memory limitations. A 1080p buffer can be merged in under 0.5 percent of a frame on a single core on a PC title running at 60 frames per second.

Using Buffer Merging to Parallelize Hierarchical Depth Buffer Creation

The Masked Occlusion library already provides a method for parallelizing the creation of the hierarchical depth buffer. By using a system of tiled rendering, where incoming meshes are sent to a job system and rendered in parallel, the resulting triangle output of the geometry transformation stage is stored into binned lists representing screen-space tiles and then processed by the job system with each thread, rasterizing the geometry of one screen-space tile. The merging of hierarchical depth buffers offers an alternative approach that works on a much coarser grain and doesn't require the geometry to be stored in a temporary structure. A typical setup is shown in Figure 13. The task system schedules two occluder rendering tasks initially, and creates a merge task that is triggered when both render tasks are complete.

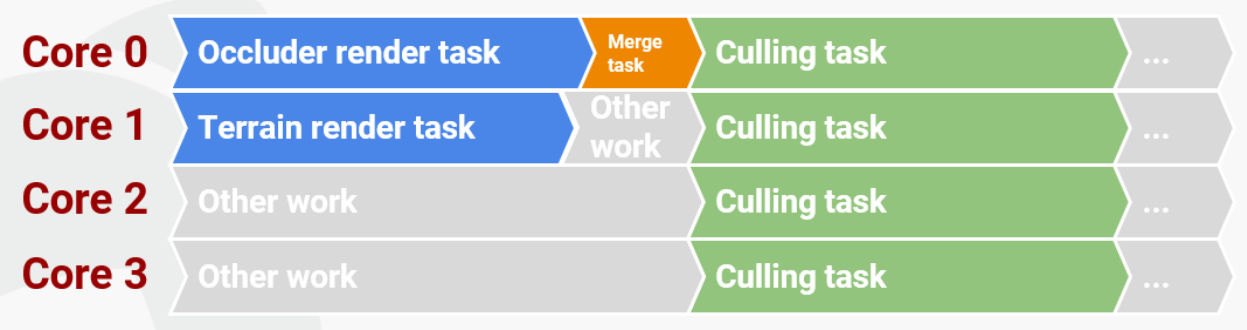

Figure 13: *Potential multithreading setup for Masked Occlusion Culling.*

Once the merge task is complete, the rendering application's actual occlusion queries can be issued on as many threads as required. In theory, the foreground occluder render tasks could be further subdivided. That may be advantageous if the amount of work between them is very uneven, if the cost of each additional task would be an extra merge task, and if the merge function could be further modified to manage multiple buffers (if required). One side benefit of threading in this way is the removal of the memory required and the associated bandwidth saving of not having to write out the geometry between transform and rasterization passes. Overall, this approach of splitting a scene into two or more Masked Occlusion Culling buffers, and using independent threads to process them, allows for a functional threading paradigm that is suitable for engines that do not support the fine-grained tasking system required for the Intel tile-binning algorithm.

Conclusion

Our extensions to Masked Software Occlusion Culling results in a flexible solution that increases the final accuracy of the depth buffer for scenes that cannot be sufficiently presorted without compromising performance of the original Masked Occlusion algorithm. The merge time in our approach scales linearly with resolution and is independent of the geometric complexity in the scene. In test cases, our approach represented only a small part of the total time required for the culling system. A final benefit is that our

partial results subgrouping enables new thread-level parallelism opportunities. These improvements produce robust, fast, and accurate CPU-based geometry culling. As a whole, the improvements significantly reduce the barrier to entry for a game engine to adopt Masked Software Occlusion Culling, freeing GPU rendering resources to render richer game experiences.

References

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

[FBH∗10] K. Fatahalian, S. Boulos, J. Hegarty, K. Akeley, W. R. Mark, H. Moreton, P. Hanrahan: *Reducing Shading on GPUs using Quad-Fragment Merging*. ACM Transactions on Graphics, 29, 4 (2010), 67:1–67:8. 4

[HAAM16] M. Andersson, J. Hasselgren, T. Akenine-Möller: *MaskedSoftwareOcclusionCulling*.

[MM00] J. Mccormack, R. Mcnamara: *Tiled Polygon Traversal UsingHalf-PlaneEdgeFunctions. InGraphicsHardware* (2000), pp.15– 21. 2