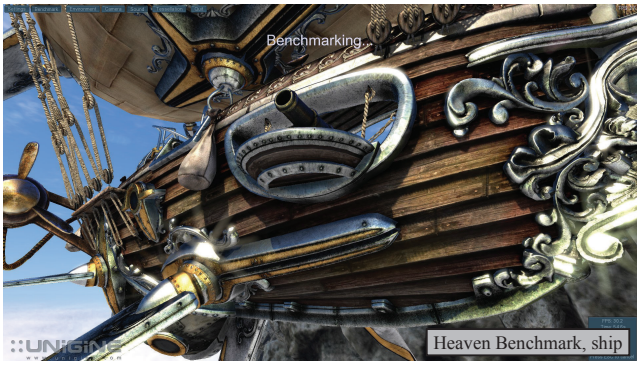
Masked Depth Culling for Graphics Hardware

Magnus Andersson[1,2] Jon Hasselgren[1] Tomas Akenine-Möller[1,2]

[1]Intel Corporation [2]Lund University



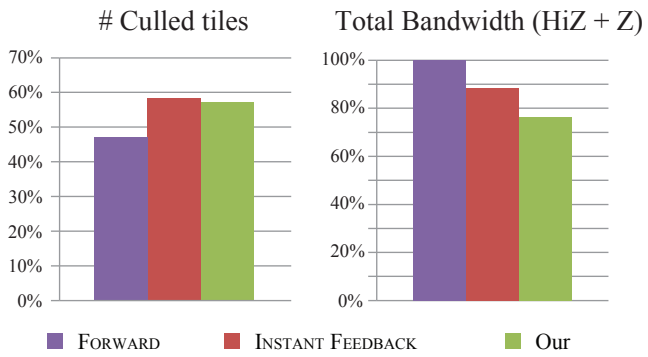


Figure 1: *Best case culling performance and bandwidth of our algorithm as compared to previous work. Our algorithm performs similarly to an idealized version of the more complicated feedback algorithm, while keeping the simplicity of the much less efficient forward culling approach. By compressing the depth representation, we show that we achieve significantly less bandwidth than competing algorithms.*

Hierarchical depth culling is an important optimization, which is present in all modern high performance graphics processors. We present a novel culling algorithm based on a layered depth representation, with a per-sample mask indicating which layer each sample belongs to. Our algorithm is feed forward in nature in contrast to previous work, which rely on a delayed feedback loop. It is simple to implement and has fewer constraints than competing algorithms, which makes it easier to load-balance a hardware architecture. Compared to previous work our algorithm performs very well, and it will often reach over 90% of the efficiency of an optimal culling oracle. Furthermore, we can reduce bandwidth by up to 16% by compressing the hierarchical depth buffer.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Hidden line/surface removal

Keywords: Occlusion Culling, Depth Buffer, Graphics Hardware

1 Introduction

Over 400 million graphics processors were sold in the notebook and desktop segments in 2014. In each of these GPUs, there is a highly optimized fixed-function hierarchical depth culling unit that uses occlusion culling techniques on a per-tile basis [Greene et al. 1993; Morein 2000]. Substantial engineering efforts have been spent fine-tuning such units, in order to minimize memory traffic to the depth buffer, which, in turn, improves performance and/or reduces power. Occlusion culling is integrated transparently in GPUs, i.e., most users enjoy its benefits without ever knowing it is there. The large number of units shipped each year and the performance/power benefits that comes with hierarchical depth culling makes it very important not only to increase efficiency, but also to make the implementation simpler and more robust to different use cases.

We present a novel culling algorithm that uses a layered depth representation with a selection mask that associates each sample to a layer. In our algorithm, culling and updating the representation is very inexpensive and simple, and unlike previous methods we compute accurate depth bounds without requiring an expensive feedback loop [Hasselgren and Akenine-Möller 2006]. Furthermore, we have greater freedom in choosing tile size since we do not require scanning the depth values of all samples in the backend. This, in turn, makes it easier to load-balance the graphics pipeline. See Figure 1 for an example of the culling potential of our algorithm.

2 Previous Work

Greene et al. presented a culling system based on a complete depth pyramid, with conservative $z_{\max}$-values at each level [1993]. However, while highly influential, it is not practical to keep the entire pyramid of depths updated at all times. Morein [2000] had a more practical approach, where the maximum depth, $z_{\max}$, was stored and computed per tile. If the conservatively estimated minimum depth of a triangle inside a tile is greater than the tile's $z_{\max}$, then the portion of the triangle overlapping the tile can be culled. In addition, it is also possible to store the minimum tile depth, $z_{\min}$, which is used to avoid depth reads. If the triangle's conservatively estimated maximum depth is smaller than $z_{\min}$ [Akenine-Möller and Ström 2003], the triangle can trivially overwrite the tile (assuming no alpha/stencil test etc), and the read operation can be skipped. From the literature [Hasselgren and Akenine-Möller 2006; Morein 2000], we deduce that $z_{\max}$ is typically computed from the per-sample depths in a tile, and must be passed to the hierarchical depth test using a feedback loop. Ideally, the $z_{\max}$-value of a tile should be recomputed and updated every time the sample with the maximum depth value is overwritten, but updates are typically less frequent in order to reduce computations. For example, $z_{\max}$ may be recomputed when a tile is evicted from the depth cache.

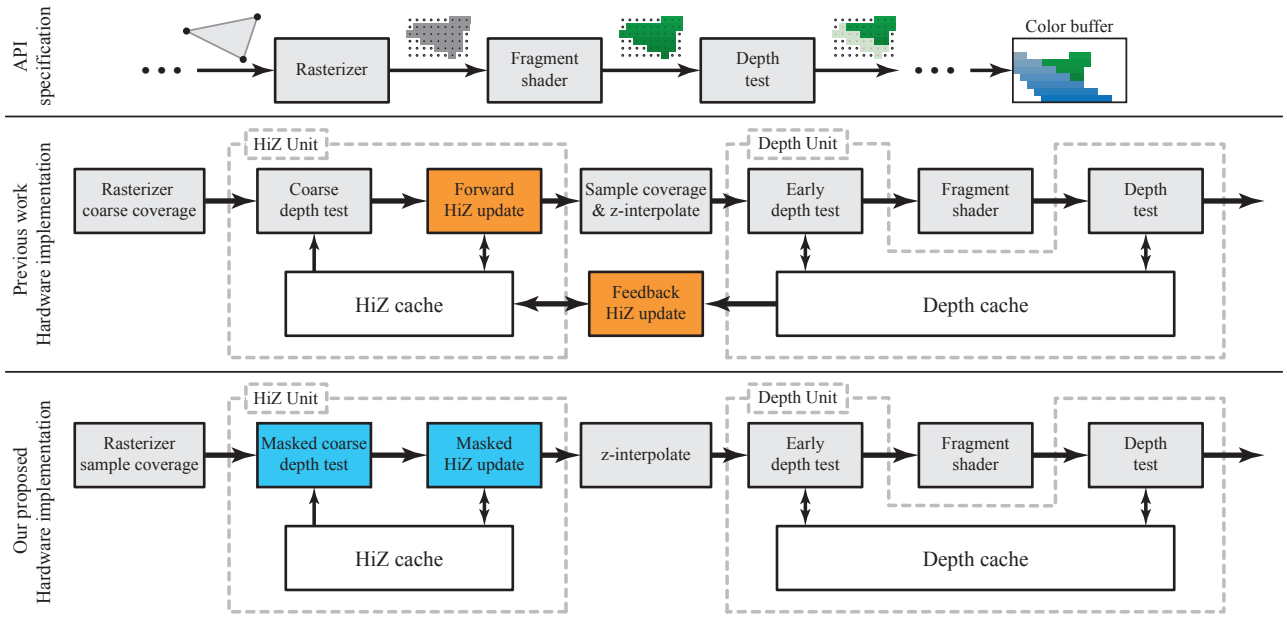


Figure 2: Top: *a simplified overview of the rasterizer, fragment shader, and depth units according the OpenGL and DirectX API specifications.* Middle: *a hardware architecture, according to previous work, featuring a HiZ culling unit and an early depth test. The purpose of these additional units is to improve performance through early occlusion culling. They are transparent to the programmer and can be implemented without changing the API.* Bottom: *our proposed architecture. Our novel HiZ culling algorithm completely removes the need for feedback HiZ updates by using a layered depth representation with a per-sample selection mask, which is efficient for culling and is easy to update.*

Occlusion queries count the number of fragments passing the depth test and can be used to cull entire objects [Bittner et al. 2004; Guthe et al. 2006; Mattausch et al. 2008; Staneker et al. 2003] using simple proxy geometry, such as a bounding box. A system for dynamic occlusion culling has been presented by Aila and Miettinen [2004], and in the gaming industry, it has proven useful to base occlusion queries on software rasterization to better load balance the CPU and GPU [Collin 2011]. Zhang et al. [1997] proposed using hierarchical occlusion maps for occlusion queries. Rather than basing the queries on the depth buffer, they use a full resolution, hierarchical coverage map, and store depth separately in a low resolution depth estimation buffer.

Our algorithm uses occluder merging inspired by the work of Jouppi and Chang [1999]. They propose an algorithm for low-cost anti-aliasing and transparency by storing low-precision depth plane equations. A fixed number of planes are stored per pixel and overflow is handled using a merge heuristic. Similarly, Beaudoin and Poulin [2004] extend MSAA [Akeley 1993] to use a hierarchical indexing structure to reference a small set of color and depth values per tile. To handle layer overflow they opted to reduce the sampling rate, rather than using a lossy merge heuristic.

Greene and Kass extended their earlier work [Greene et al. 1993] to include anti-aliasing with error bounds using interval arithmetic for the shaders and quad tree subdivision for visibility handling [1994]. Furthermore, Greene et al. [1996] used a BSP tree to hierarchically traverse the scene and the screen space using a pyramid of coverage masks for efficient anti-aliasing. In order to save pixel shading work for small triangles, Fatahalian et al. [2010] gather and merge quad-fragments from adjacent triangles, using an aggregate coverage mask. The purpose of their mask differs from ours in that they use it to avoid merging shading over geometric discontinuities, while our masked representation is a lossy, but conservative, approximation of the depth buffer.

3 Overview of Current Architectures

In order to contextualize our algorithm, we first describe a typical implementation of a GPU depth pipeline as presented by previous work [Hasselgren and Akenine-Möller 2006]. The top row of Figure 2 depicts the pipeline from a functional standpoint, as specified in most modern graphics APIs. Depth and color buffers are progressively updated as a sequence of triangles goes through the rendering pipeline. For each sample, the depth of the closest triangle is stored along with its color. The actual hardware pipeline typically differs from the API specifications for performance reasons, and a common implementation can be seen in the middle row of Figure 2. In the following, we will limit the discussion to a *less than* depth function to simplify the description, but the techniques generalize to all types of depth functions used in popular API's, such as DirectX, OpenGL, and presumably also in the coming Vulkan API.

Rasterizer We skip the geometry processing part of the graphics processor, and begin our discussion at the rasterizer unit. The rasterizer is responsible for determining which samples overlap a particular triangle. As an optimization, modern rasterizers typically work on *tiles*, which are groups of $w \times h \times d$ samples, where d is the number of samples per pixel. A conservative test is performed for each tile to determine if it is fully covered, is entirely outside the triangle, or partially overlap it. Per-sample coverage testing is only required for tiles partially overlapping the triangle. Once the sample coverage is computed, each fragment is shaded using the fragment shader, followed by the depth test which determines visibility. As can be seen in the middle row of Figure 2, a common optimization is to place the per-sample coverage test after the hierarchical z or *HiZ* unit. The rationale is that the HiZ may remove or cull tiles before the per-sample coverage test occurs, which improves the performance of that unit.

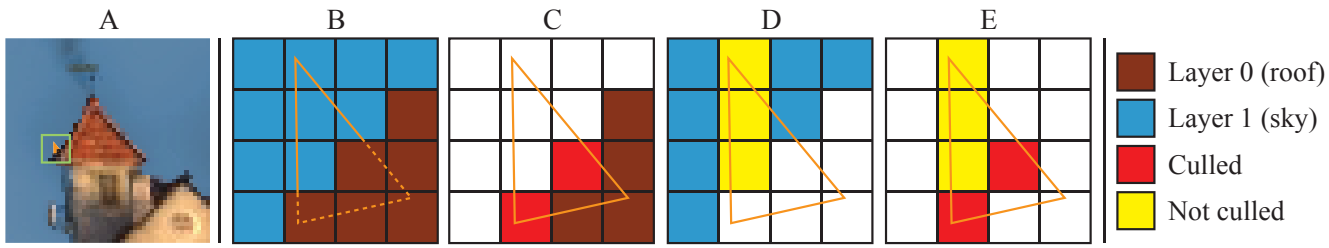


Figure 3: *A step by step illustration of how the fail-mask is calculated in our coarse depth test. A: The orange triangle is rasterized and forwarded to the HiZ unit. B: The coarse depth buffer already contains a tile with two layers, namely, the roof in the foreground (brown) and the sky background (blue). As indicated by the dashed lines, the triangle is occluded by the roof, while visible in front of the sky. C: The triangle is overlap tested against the depth of the roof layer and since $z^{tri}_{min} > z^{0}_{max}$, the result is the fail-mask indicated by the two red pixels. D: The triangle is overlap tested against the sky, but cannot be culled. Thus, the fail-mask for this layer is actually empty, but we indicate the three ambiguous pixels for clarity. E: The aggregate fail-mask. We cannot cull the triangle since the fail-mask does not contain all pixels covered by the triangle. However, it would be possible to skip the depth test for the two red pixels.*

HiZ Unit Depth testing may consume significant memory bandwidth and compute power [Aila et al. 2003]. For this reason, the hardware pipeline typically has a HiZ unit with the purpose of quickly discarding (culling) or accepting tiles using a *coarse depth test*, whenever the outcome of the depth test can be unambiguously determined for the entire group of samples. For this purpose, the HiZ unit maintains a conservative version of the depth buffer, referred to as the *coarse depth buffer*, which contains per-tile depth bounds, $[z^{tile}_{min}, z^{tile}_{max}]$.

Coarse depth test When a tile reaches the HiZ unit, the first step is to compute conservative bounds, $[z^{tri}_{min}, z^{tri}_{max}]$, of the depth of the incoming triangle within the tile [Akenine-Möller et al. 2008]. These bounds are then tested against the coarse depth buffer using interval overlap tests. For example, for a *less than* depth function, we can conclude that the per-sample depth test will fail for all samples if $z^{tri}_{min} \geq z^{tile}_{max}$ and pass if $z^{tri}_{max} < z^{tile}_{min}$. This leaves an ambiguous depth range, where the outcome of the per-sample depth test cannot be determined. Thus, the coarse depth test has one of three outcomes, namely, *fail*, *pass*, or *ambiguous*. Failed, i.e., *culled*, tiles are immediately thrown away and require no further processing. Passing and ambiguous tiles are sent down the pipeline for further processing, with the main difference being that ambiguous tiles must be fully depth tested in the depth unit, while trivially passing tiles can simply overwrite the contents of the depth buffer. This is a small difference, but depending on the architecture, write-only operations may result in lower bandwidth than the read-modify-write operation required for performing the full depth test [Akenine-Möller and Ström 2003].

Coarse depth buffer update As rendering progresses, the coarse depth buffer is continually updated. From previous work [Morein 2000; Akenine-Möller and Ström 2003], we note that z^{tile}_{min} and z^{tile}_{max} are updated separately in the pipeline using two different mechanisms, namely, a *forward* update located immediately after the coarse depth test and a *feedback* update located between the depth unit and the HiZ unit. This is illustrated in the middle row in Figure 2.

In the forward update stage, z^{tile}_{min} can be efficiently computed as $z^{tile}_{min} = \min(z^{tri}_{min}, z^{tile}_{min})$. The z^{tile}_{max}-value, however, can only be updated if *all* the samples in the tile are overwritten. If the coverage mask is fully set, then $z^{tile}_{max} = \min(z^{tri}_{max}, z^{tile}_{max})$. Unfortunately, this update scales very poorly when using smaller triangles since they are less likely to completely overlap a tile. An example of the irregular coverage resulting from solely using the forward stage can be viewed in Figure 9.

A better z^{tile}_{max} value is obtained using a feedback update. Here, a max-reduction on an entire tile of depth samples is performed in the depth unit and the result, $z^{feedback}_{max}$, is sent to the HiZ unit through the feedback mechanism, as depicted in Figure 2. To reduce the number of max-reductions performed, the feedback update typically occurs each time a tile is evicted from the depth cache. The feedback mechanism introduces a large delay, as the HiZ and depth units may be separated by hundreds of cycles in the hardware pipeline. Depending on the render state, we may also need to wait for the fragment shader to be executed and for the tile to be evicted from the cache before the update can occur. As a consequence, z^{tile}_{max} updates may lag behind, leading to decreased culling rates. Furthermore, the delay has non-obvious side effects, such as how to conservatively handle cases where the feedback message originated from a different GPU-state than is currently active.

Depth Unit Similar to the HiZ unit, the depth unit typically works on tiles of samples, but instead of performing a single test, each sample is individually tested against the value stored in the depth buffer. The size of a tile in the depth unit is typically correlated to the size of a cache line, which in turn is determined by how much data can be efficiently streamed to and from memory. It should be noted that the feedback mechanism creates a constraint between the tile sizes of the HiZ and depth unit. The max-reduction operation needs to be performed on the granularity of the tiles in the HiZ buffer. Therefore, it is important that all data required for the operation resides in the depth unit cache, and the easiest way to guarantee this is to couple the tile sizes of the coarse depth buffer and the regular depth buffer.

The most straightforward optimization in the depth unit is the *early depth test*, which takes advantage of that depth testing can be performed before fragment shading in many cases. This typically improves performance significantly as fragment shading is expensive and often a bottleneck. For the most part, it is safe to use the early depth test, but it must be disabled when, for example, the fragment shader alters the coverage through a `discard` operation, outputs a depth value, or writes to an unordered access view (UAV) resource.

4 Algorithm

It is challenging to accurately update the maximum depth of the tile without relying on the feedback mechanism. The key innovation in our algorithm is an efficient and accurate way to update the coarse depth buffer using *only* forward updates, which completely removes the need for the feedback mechanism, as illustrated in the bottom row of Figure 2. The updates are performed progressively

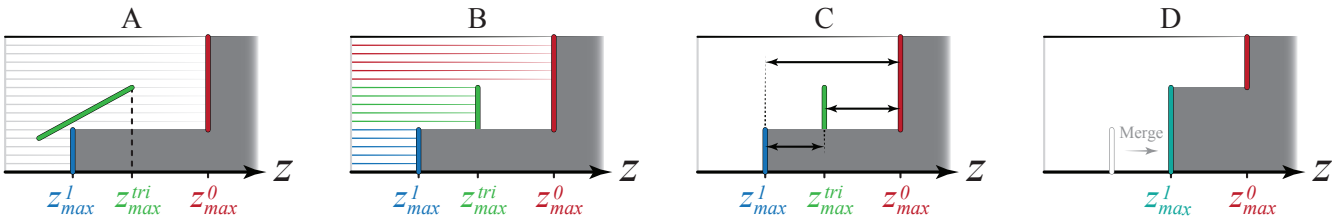


Figure 4: *An example of our occluder merge heuristic for a single tile shown in normalized device coordinates (NDC). The occluded region, as encoded by our algorithm, is shown in dark gray.* A: *a green, slanted primitive is rendered in front of two existing layers, illustrated with red and blue lines respectively.* B: *each sample is classified as belonging to one of the layers to create sample masks.* C: *for our merge heuristic, we find the closest pair of layers along the z-axis, which in this case is between z^1_{max} and z^{tri}_{max}.* D: *we select the maximum of these two depth values as the new z^1_{max} and fuse their sample masks to form a new selection mask.*

in a streaming fashion and only use information about the current triangle. No buffering of triangles or rendering history is required.

Without loss of generality, we limit the description of our algorithm to only two depth layers[1] per tile. Each tile has one z_{min} value and two z^i_{max} values. In addition, we store a *selection mask* of one bit per sample, which associates each sample with one of the two layers, i. We keep the z_{min} update strategy described Section 3, as it is simple and efficient. Each z^i_{max} must be greater or equal to all samples associated with that layer. We achieve this using a conservative merge of the incoming triangle and the layered representation. In the following, we describe the coarse depth test and update in detail.

Coarse depth test As described in Section 3, the triangle and its coverage mask is provided by the rasterizer, which enables us to compute z^{tri}_{min} and z^{tri}_{max} as before. Similar to how the coarse depth tests are performed for $z_{\min}/z_{\max}$-culling, we do interval overlap tests between $[z^{tri}_{min}, z^{tri}_{max}]$ and $[z_{min}, z^i_{max}]$ for each layer, as outlined in Figure 3. Aggregate per-sample pass- and fail-masks can be constructed from the triangle's coverage mask and the selection mask using simple bitwise operations. The exact depth test is only required for the samples that are not present in either of the pass- or fail-masks. Pseudo-code for how the coarse depth test is performed is given in Listing 1 in the appendix.

Coarse depth buffer update Unless all samples were culled by the coarse depth test, we must update the coarse depth buffer in a way that makes it conservatively bound the contents of the depth buffer. Updating the z_{min}-value is done as previously described in Section 3. The challenge lies in updating the z^i_{max} values and the selection mask. The incoming triangle forms a third depth layer, in addition to the, up to, two layers already populating the tile. We handle layer overflow by merging two of the layers using a heuristic, as shown in Figure 4 and described in detail below.

First, consider a single sample, S, which belongs to layer i and is also found to be overlapping the incoming triangle. With a *less than* depth test, we know that the depth of S after the depth test will be *at most* the minimum (closer) value of z^i_{max} and z^{tri}_{max}. Based on this observation, by comparing both z^i_{max}-values to z^{tri}_{max}, we can categorize which layer each sample belongs to – either its previous layer, i, or the incoming triangle layer. From this, we construct three non-overlapping *sample masks* signaling which of the three layers, z^0_{max}, z^1_{max}, and z^{tri}_{max}, each sample belongs to, as step B in Figure 4 exemplifies.

After categorizing the samples, if there are any layers that do not have samples associated with them (i.e., the sample mask is empty for a layer), the coarse buffer update is simple. Since the resulting number of layers is ≤ 2, the data will fit in our representation and we can simply write the populated layers to the coarse depth buffer. If there are samples in all three layers, we use a simple distance-based heuristic to select which layers should be merged. The underlying assumption is that triangles that have similar depth values are likely to be part of the same surface. As illustrated in step C in Figure 4, we first compute the distances between all of the layers as

$$d_{T0} = |z^{tri}_{max} - z^0_{max}|,$$
$$d_{T1} = |z^{tri}_{max} - z^1_{max}|,$$
$$d_{01} = |z^0_{max} - z^1_{max}|.$$

The shortest distance is then used to determine which merge operation is performed, as depicted in step D in Figure 4.

1. If d_{T0} is smallest then $z^0_{max} = \max(z^{tri}_{max}, z^0_{max})$.
2. If d_{T1} is smallest then $z^1_{max} = \max(z^{tri}_{max}, z^1_{max})$.
3. Otherwise $z^0_{max} = \max(z^0_{max}, z^1_{max})$ and $z^1_{max} = z^{tri}_{max}$.

The sample masks of the two closest layers are also merged (using simple bitwise operations) to produce the new selection mask. Pseudo-code for the update and merge functions can be found in Listing 3 and Listing 4 in the appendix.

Switching Depth Functions Our algorithm can easily handle depth function switches while rendering. For the *greater than* depth functions, tiles are instead represented by two z^i_{min} values and one z_{max} value. We store a single bit for each coarse depth buffer tile indicating which representation is currently used. If the tile does not match the current depth function we convert it before updating the coarse depth buffer. Conversion is performed by conservatively swapping the min and max values. For example, if the tile stored in the coarse depth buffer has two max layers, but the depth function is changed to *greater than*, we convert the tile by setting $z_{max} = \max(z^0_{max}, z^1_{max})$ and $z^0_{min} = z^1_{min} = z_{min}$ and clearing the selection mask. The conversion is quite crude and may lose a lot of culling information, but we have not found any workload where this has been an issue, as depth function changes are infrequent in most scenes. All standard OpenGL/DirectX depth functions can be handled using the *less than* or *greater than* representation.

Coarse depth test pipeline placement As can be seen in the bottom row of Figure 2, our coarse depth test is based on the coverage mask, which means that per-sample coverage testing must be moved to the rasterizer block. As previously mentioned, placing the per-sample coverage test behind the HiZ unit is beneficial, as

[1] All depth layers are disjoint and jointly cover the entire tile. This is not to be confused with layered depth images.

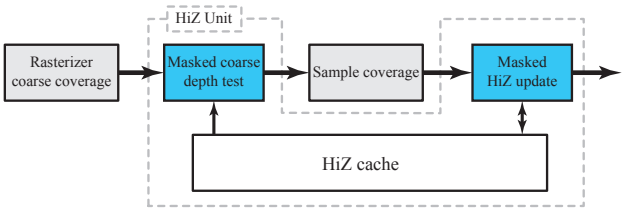


Figure 5: *It is possible to modify the coarse depth test to rely only on $z_{\min}/z_{\max}$, and not per-sample coverage results. Unlike our solution illustrated in Figure 2 (bottom), this alternative implementation does not incur the expense of per-sample coverage testing prior to HiZ, but can decrease efficiency as culling is performed on more conservative information.*

coverage testing may be skipped for culled tiles, reducing the load of this unit.

We can achieve a similar effect by using an alternate coarse depth test, where we perform an overlap test between the $[z^{tri}_{min}, z^{tri}_{max}]$ and $[z_{min}, \max(z^0_{max}, z^1_{max})]$ intervals, similar to the classic HiZ test. As shown in Figure 5, we may then place the per-sample coverage test between the coarse depth test and update, which results in similar load balancing to previous work. This version of the coarse depth test is less accurate, but most of the benefits of our algorithm comes from the accurate update. Pseudo-code for this alternative approach can be found in Listing 2 in the appendix. Compared to the results presented in Section 5, the culling rate of our algorithm decreases by $0.2 - 2.1$ percentage points and total bandwidth increases by $0.7-1.4$ percentage points. It is also possible to perform both versions of the coarse test, efficiently filtering most per-sample coverage tests while retaining the benefits of the accurate version.

4.1 Compression

In order to reduce coarse depth buffer bandwidth, we use a simple compression scheme when entries are evicted from the coarse depth buffer, similar to how regular depth buffer compression works [Hasselgren and Akenine-Möller 2006; Hasselgren et al. 2012].

Our compression scheme is inspired by zerotree encoding of wavelet coefficients [Shapiro 1993]. Each tile is first split into a set of *blocks*, each containing b samples. For each block, we store a single bit signaling whether its samples contain a mix of indices to both layers or if all samples belong to the same layer. If all indices are the same, only one additional bit is required to assign the entire block to the layer. A block containing indices to both layers require an explicit mask with b bits. With this scheme, the compressed selection mask cost, c, for the tile containing s samples is $c = 2\frac{s}{b} + (b-1)m$, where m is the number of blocks that need to be explicitly stored. The selection mask can be compressed without loss if $c \leq s$. Interestingly, we can limit m by performing lossy compression, without introducing artifacts in the rendered image. The coarse depth buffer representation is still valid (i.e., conservative w.r.t. the depth buffer) if we alter an index in the selection mask to use the farther of the two z^i_{max}-values. It is thereby possible to enforce a maximum value of m by forcing some blocks to use a single layer, instead of a mix of both.

Furthermore, we decrease the precision of z_{min} and z^i_{max}. There is a variety of possible options depending on the bit budget available, the depth buffer target format, and the expected distribution of depth samples. We have opted to use a simple reduced precision float with fewer exponent and mantissa bits. In addition, we only use negative exponents and no sign bit, limiting the representable range to $[0, 1]$.

5 Results

When comparing different culling algorithms, there are two main quantities that are of interest – memory bandwidth usage and throughput. Bandwidth is primarily consumed by the depth unit when reading and updating the depth buffer, and to a lesser extent by the HiZ unit for maintaining the coarse depth buffer. The number of per-sample tests the depth unit has to perform depends on the amount of tiles culled (failed) by the coarse depth test, and consequently a higher culling rate leads to better throughput. Depending on the system and the expected workloads, these quantities must be balanced against each other for maximum performance. We evaluated five different pipeline configurations listed below with regards to bandwidth and culling rates (i.e., the percentage of tiles culled by the coarse depth test):

- ORACLE - The HiZ unit replicates the exact depth buffer and performs per-sample depth tests. For each sample, there is no ambiguous outcome, only pass or fail. A tile is only classified as ambiguous if it contains both samples passing and failing the depth test. This pipeline is only used to get an upper bound on possible cull rates.
- FORWARD - Only the forward update unit for $z_{\min}/z_{\max}$-culling is enabled. A feedback unit with infinite delay will act as a forward-only pipeline, which makes this configuration a lower bound on the culling rate such a design can achieve.
- INSTANT FEEDBACK - Both the forward and the feedback HiZ update mechanisms are used. z_{max}-updates are triggered directly on depth buffer updates (i.e., as early and as often as possible), and there is no feedback delay, which gives us an upper (albeit unrealistic) bound on how well a forward/feedback-design can perform.
- ZMASK - Our proposed feed forward algorithm.
- PACKED ZMASK - Our algorithm tailored to minimize bandwidth. This variant requires an additional post-HiZ cache compression stage, as described in Section 4.1.

Our results are based on a C++ hardware simulator, which models the system on a functional level. We use a 32 kB depth cache and a 16 kB HiZ cache, both using a cache line size of 64 B (unless otherwise stated) and both with a least recently used (LRU) replacement policy. For our main results, found in Figure 6, we allocate a coarse depth buffer which amounts to an overhead of 4 bits per depth sample for the FORWARD, INSTANT FEEDBACK, and ZMASK configurations. With this storage, we can keep z^{tile}_{min} and z^{tile}_{max} in a 32 bit format each at a 16 sample granularity for FORWARD and INSTANT FEEDBACK (this corresponds to 4×4 pixel tiles for single sample targets and 2×2 pixels for $4\times$ multi-sample targets). For ZMASK we store one z_{min} and two z^i_{max} at a 32 sample granularity, using 32 bits for each entry, as well as a 32 bit per-sample selection mask (corresponding to 8×4 pixel tiles for single sample targets and 4×2 pixel tiles for $4\times$ MSAA targets). All configurations use the common *fast clears* [Morein 2000] optimization to ensure that the results are not biased by how the different algorithms handle clear values.

The PACKED ZMASK is similar to the ZMASK algorithm, but uses larger tiles (16×8 pixel and 8×4 pixel tiles for single and $4\times$ MSAA targets respectively), and compresses them as they are evicted from the HiZ cache. In the cache, each tile occupies $128 + 3 \cdot 32 = 224$ b, or 28 B of memory. Since we do not want to alter the memory transaction size of 64 B, we group 4 tiles to a common cache line of 112 B, which is compressed down to 64 B on eviction. To achieve this level of compression, we use a reduced precision float format with 4 exponent bits and 11 mantissa bits for the z_{min} and z^i_{max} values, one bit to encode the test direction (see

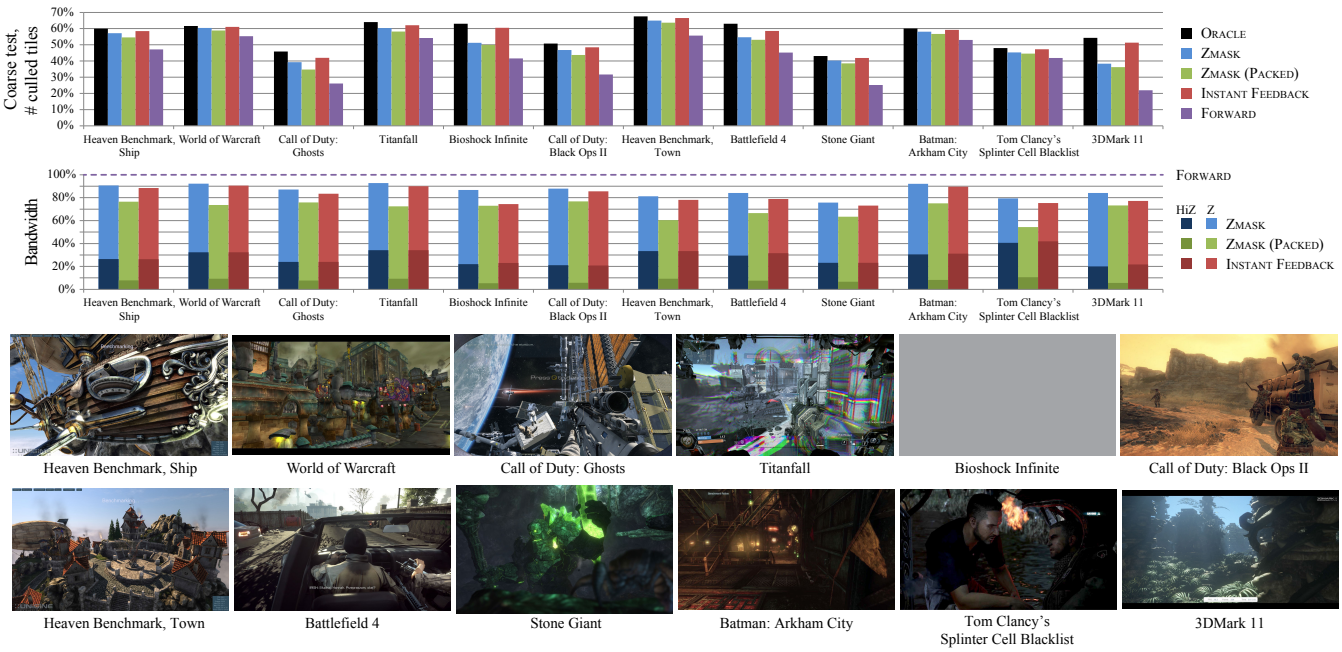


Figure 6: Top: *the percentage of incoming 16 sample tiles (i.e.* 2×2 *pixels for* $4\times$ *MSAA targets and* 4×4 *pixels for single sample targets) that were culled by the coarse depth test for the different algorithms. A higher number means that less work is pushed through the pipeline.* Bottom: *the simulated depth buffer (Z) and HiZ bandwidth, normalized to the* FORWARD *algorithm. The darker bars indicate the HiZ bandwidth and the lighter bars is the depth buffer bandwidth. Note that even though the* PACKED ZMASK *culls fewer tiles and has a higher depth buffer bandwidth, the total bandwidth is still very low due to the HiZ bandwidth savings. Note that the screen shots are generated using our hardware simulator. While we strive to make it feature complete according to the latest OpenGL/DirectX specifications, some visual differences may occur compared to commercial GPUs and production drivers.*[2]

Section 4), and the remaining 82 bits are spent on storing the selection mask using the compression format described in Section 4.1. Thus, when using the PACKED ZMASK pipeline, each tile occupies 1.75 bits per sample while in the cache, and reading or writing the tile from/to memory uses 1 bit of bandwidth per sample. The tile size was selected empirically by finding the best balance between depth buffer bandwidth and HiZ bandwidth, as described in Figure 8.

Our main results are shown in Figure 6, where we present the coarse culling rates and the bandwidth consumed by each pipeline. The culling rates have been normalized to 16 sample tiles for the ZMASK and PACKED ZMASK algorithm to simplify comparison. Since the coarse depth test pass rates are very similar between the algorithms, we focus solely on the number of culled tiles (i.e., tiles where the depth test unambiguously fails) as a measure of throughput. The test suite contains a number of traces from modern games, with a variety of different render state combinations, and includes $1 - 4\times$ MSAA buffers as well as auxiliary targets such as shadow maps. As can be seen from the results, our algorithm is often close to the ORACLE in terms of culling efficiency. The FORWARD pipeline is always considerably less efficient than all of the alternatives. On average, we retain 90% of the rejection rate of the ORACLE pipeline, with the more difficult cases typically being scenes with a greater amount of alpha tested geometry, or scenes that output depth in the fragment shader. Note that ZMASK uses about 14% less total bandwidth compared to FORWARD, while INSTANT FEEDBACK uses about 18% less than FORWARD. It should be noted, however, that INSTANT FEEDBACK is idealized and impractical to implement. Increasing the pipeline delay reveals the weakness of the feedback algorithm, as we will show later in this section. By increasing the tile size and enabling compression using the PACKED ZMASK configuration, we lower the number of culled tiles and depth buffer bandwidth increases as a result. However, since the HiZ bandwidth is reduced, total bandwidth consumption is approximately 30% less than FORWARD on average, which is a substantial reduction.

The culling numbers presented for the ORACLE algorithm include tiles where the coverage is modified by the fragment shader. Alpha tested billboards, for examples, can have large, fully transparent portions that are discarded in the fragment shader.

Feedback Delay As previously mentioned, the performance of the feedback algorithm depends on how long delay can be expected in the pipeline. We opted to implement the feedback mechanism on depth buffer updates, rather than cache evicts, and simulate delay by introducing a FIFO-queue when feeding the messages back. This allows us to delay the messages by an arbitrary number of processed tiles and study how increased delay affects system performance.

Figure 7 shows how the number of culled tiles and total simulated depth buffer bandwidth (for both the coarse and exact depth buffers) are affected by an increased delay compared to the ZMASK ap-

[2] *Heaven Benchmark* screenshots courtesy of UNIGINE Corp. *World of Warcraft*, *Call of Duty®: Ghosts* and *Call of Duty®: Black Ops II* screenshots courtesy of Activision Blizzard, Inc. *Titanfall* screenshot courtesy of Respawn Entertainment. *Bioshock Infinite* Screenshot Courtesy of Irrational Games and Take-Two Interactive Software, Inc. *Battlefield 4*, © 2013 Electronic Arts Inc. Battlefield and Battlefield 4 are trademarks of EA Digital Illusions CE AB. Image from *Stone Giant* demo, courtesy of BitSquid. *Batman: Arkham City* screenshot courtesy of Rocksteady Studios. *Tom Clancy's Splinter Cell Blacklist* screenshot courtesy of Ubisoft. *3DMark 11* screenshot courtesy of Futuremark.

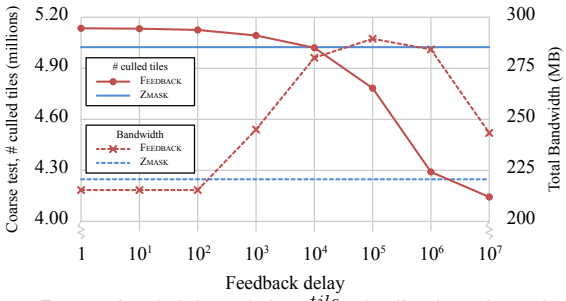


Figure 7: *Artificial delay of the z^{tile}_{max} feedback updates for the* Heaven Benchmark, Ship *scene. Here, each unit corresponds to delaying the feedback update by one processed tile. At a delay of 10^3 tiles, we see that the coarse depth test rejection rate starts to drop, while the bandwidth rapidly increases. While part of the increasing bandwidth is explained by reduced culling efficiency, the lion's share is due to cache behavior. As the delay increases, the tiles referenced by the feedback messages may already have been evicted from the HiZ cache, which may lead to cache thrashing and increased bandwidth. This creates an intricate relation between the system delay, which may depend on depth buffer cache size and shader execution, and the size of the HiZ cache. At 10^7 steps, the delay is so large that the rendering finishes before any of the feedback updates occur, leaving only the forward updates (i.e., equivalent of running the* FORWARD *algorithm). Since there are no delayed updates at this point, there are also no cache conflicts, which explains the bandwidth reduction.*

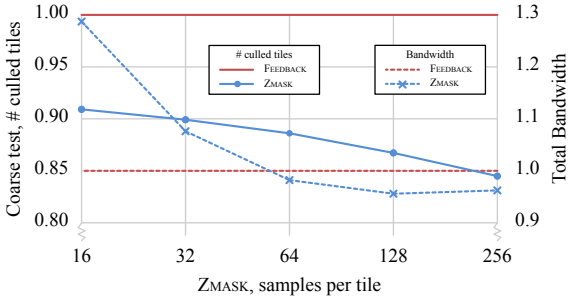


Figure 8: *The graph illustrates how bandwidth and culling rates are affected by varying the tile size for* ZMASK. *The numbers are combined over all of our test scenes, normalized to the* INSTANT FEEDBACK *algorithm with fixed* 16 *sample tiles. To simplify implementation, the cache line and memory transaction size match the footprint of a single tile. Although depth buffer bandwidth increases due to the lowered cull rate, the decrease in HiZ memory traffic counter this effect. In our setup, the lowest combined bandwidth occurs around* 128 *samples, which is the tile size selected for the* PACKED ZMASK *algorithm.*

proach. As expected, the number of culled tiles decreases with increasing delay. Similarly, total bandwidth usage increases significantly for larger delays. When the delay becomes sufficiently large, the coarse depth buffer entries may already have been evicted from the HiZ cache before a feedback message is received. Consequently, the data must be read back into the cache in order to perform the feedback update, and the HiZ and feedback mechanisms will compete for which data should be resident in the cache. Therefore, it is important to balance the size of the HiZ cache and the depth unit cache based on the expected delay of the system. Alternatively, feedback updates of tiles not resident in the HiZ cache could be discarded, which avoids thrashing at the cost of reducing culling rates even further.

It should be noted that the delays of pipelining, shading, and caching in a real system causes culling efficiency and bandwidth to scale in a much more intricate way, and Figure 7 should only be seen as indicative of the trend. In our system, we observe a significant bandwidth impact when using an evict-based feedback strategy. If we halve the HiZ cache size to 8 kB, while keeping depth cache constant, HiZ bandwidth increase by about 10% to 45%, depending on the scene.

Tile size As our algorithm is of feed forward nature, it allows us to easily decouple tile sizes of the coarse and exact depth buffers, and this gives flexibility to chose whatever tile size gives the best trade-off between culling efficiency and coarse depth buffer bandwidth. In Figure 8, we show how our algorithm scales when varying tile size. As a reference, we also include baseline results for the INSTANT FEEDBACK pipeline.

Multiple depth layers Our algorithm can be extended to use more depth layers and we have observed some improvement in culling rates when using three or four layers, as shown in Figure 9. However, we feel that the improvement is not significant enough to motivate the added complexity. Increasing the number of layers requires more coarse depth buffer storage, as we must store additional z^i_{max} values and use more bits per sample to store the selection mask. It also complicates layer merging as the number of ways that we can merge layers scales $\mathcal{O}(n^2)$.

Stochastic motion blur As a proof of concept, we plugged the ZMASK algorithm in to our existing framework which simulates a stochastic rasterization pipeline in hardware to render images with motion blur. The framework uses the TZSLICE algorithm [Akenine-Möller et al. 2007] to perform $z_{\max}$-culling, which is the most bandwidth efficient variant of the more general tz-pyramid [Boulos et al. 2010], according to Munkberg et al. [2011]. Updates to the coarse depth buffer is done in the same way as the INSTANT FEEDBACK algorithm. For TZSLICE, we used one 32-bit float z_{max}-value for each slice of $4 \times 4 \times 1$ ($w \times h \times time$) samples. For ZMASK, we used $4 \times 4 \times 4$ tiles (64 samples) with two layers. Thus, both of these configurations use a 2 bit overhead per sample of HiZ data. As can be seen in Figure 10, even without any algorithmic modifications, ZMASK compared very well to TZSLICE. Note that these are early experiments and we believe that there is a lot more potential for improvement in this area.

Limitations While our algorithm handles even complex geometry very well, the main drawback relative to the feedback approach is that we cannot handle alpha testing, pixel shader discards, or pixel shader depth writes as accurately. While this could be an issue in extreme cases, none of our test applications seem to rely on alpha tested geometry for the main occluders (although we present an abundance of examples using alpha testing). Note that the feedback approach is also affected by alpha testing, as it implies deferring the feedback update until after the shader has been executed. It should also be noted that our algorithm could be used in conjunction with feedback updates.

6 Conclusions

We have proposed a novel $z_{\min}/z_{\max}$-culling algorithm, which we believe is an interesting and competitive alternative to the traditional feedback update mechanism. Our algorithm has similar performance to that of an *ideal* feedback architecture (without delay),

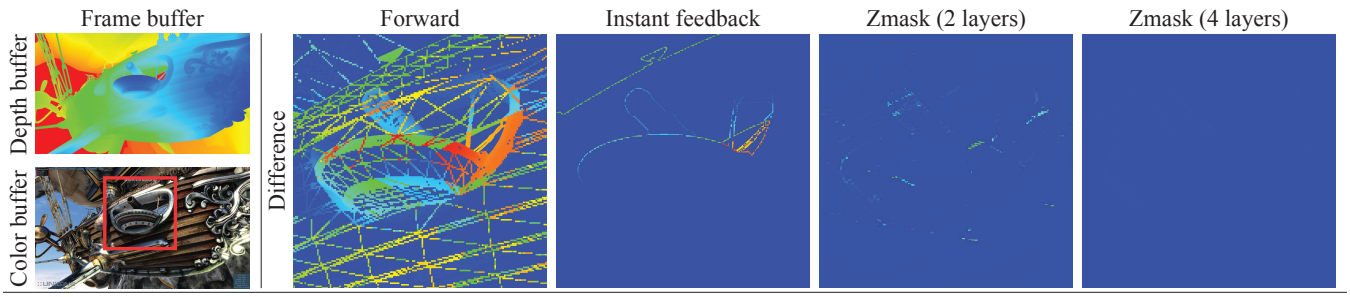


Figure 9: *The impact of multiple depth layers on the quality of the coarse depth buffer. The cropped images show the difference between the z_{max}-value of the coarse depth buffer and the exact depth buffer. The forward pipeline has massive leakage along triangle edges, and silhouette edge shows up in* INSTANT FEEDBACK *as only one z_{max}-value is stored per tile. The layer merging inaccuracies visible in the two layer version of our algorithm almost entirely disappears with four layers. However, as can be seen in the table, two layers perform well enough not to motivate the extra complexity of adding additional layers.*

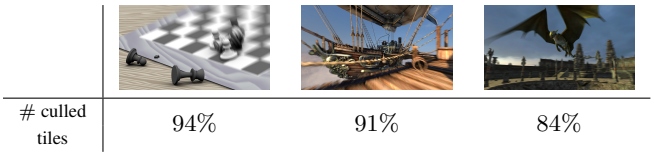


Figure 10: *Three scenes with stochastic motion blur. We count the number of $4 \times 4 \times 1$ tiles culled with* ZMASK *compared to a* TZSLICE *baseline.*

but retains the benefits of a strict feed forward pipeline. This means that implementation and validation is simplified as we do not need to consider or handle hazards that may occur from the feedback delay. Furthermore, as we decouple the tile sizes of the coarse and regular depth buffers, we have great freedom in choosing tile sizes and bit layout for the coarse depth buffer entries. This makes it easy to load-balance a hardware system and simplifies the re-design cycle if memory bus width or cache line size changes. Thus, we believe our approach is a cost-efficient and flexible solution that is suitable for current and future GPUs.

Acknowledgements

We would like thank the anonymous reviewers for their feedback. We also thank David Blythe and Mike Dwyer for supporting this research. Tomas Akenine-Möller is a Royal Swedish Academy of Sciences Research Fellow supported by a grant from the Knut and Alice Wallenberg foundation. Thanks to the following people for helping us with permissions for the screenshots in this paper: Beth Thomas at UNIGINE Corp., Christer Ericson at Activision, Abbie at Respawn Entertainment, Naty Hoffman at 2K Games, Martin Lindell and Johan Andersson at EA Digital Illusions CE, Tobias Persson at Bitsquid, Kelly Ekins at WB Games Montréal Inc., Guy Perkins at Rocksteady Studios, Heather Steele at Ubisoft and James Gallagher at Futuremark.

References

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

AILA, T., MIETTINEN, V., AND NORDLUND, P. 2003. Delay Streams for Graphics Hardware. *ACM Transactions on Graphics 22*, 3, 792–800.

AKELEY, K. 1993. RealityEngine Graphics. In *Proceedings of SIGGRAPH*, 109–116.

AKENINE-MÖLLER, T., AND STRÖM, J. 2003. Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Transactions on Graphics 22*, 3, 801–808.

AKENINE-MÖLLER, T., MUNKBERG, J., AND HASSELGREN, J. 2007. Stochastic Rasterization using Time-Continuous Triangles. In *Graphics Hardware*, 7–16.

AKENINE-MÖLLER, T., HAINES, E., AND HOFFMAN, N. 2008. *Real-Time Rendering*, 3rd ed. AK Peters Ltd.

BEAUDOIN, P., AND POULIN, P. 2004. Compressed Multisampling for Efficient Hardware Edge Antialiasing. In *Graphics Interface*, 169–176.

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

BOULOS, S., LUONG, E., FATAHALIAN, K., MORETON, H., AND HANRAHAN, P. 2010. Space-Time Hierarchical Occlusion Culling for Micropolygon Rendering with Motion Blur. In *High Performance Graphics*, 11–18.

COLLIN, D., 2011. Culling the Battle Field. Game Developer's Conference.

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

GREENE, N., AND KASS, M. 1994. Error-bounded Antialiased Rendering of Complex Environments. In *Proceedings of SIGGRAPH*, 59–66.

GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical Z-Buffer Visibility. In *Proceedings of SIGGRAPH*, 231–238.

GREENE, N. 1996. Hierarchical Polygon Tiling with Coverage Masks. In *Proceedings of SIGGRAPH*, 65–74.

GUTHE, M., BALÁZS, Á., AND KLEIN, R. 2006. Near Optimal Hierarchical Culling: Performance Driven Use of Hardware Occlusion Queries. In *Eurographics Symposium on Rendering*, 207–214.

HASSELGREN, J., AND AKENINE-MÖLLER, T. 2006. Efficient Depth Buffer Compression. In *Graphics Hardware*, 103–110.

HASSELGREN, J., ANDERSSON, M., NILSSON, J., AND AKENINE-MÖLLER, T. 2012. A Compressed Depth Cache. *Journal of Computer Graphics Techniques 1*, 1, 101–118.

JOUPPI, N. P., AND CHANG, C.-F. 1999. Z^3: An Economical Hardware Technique for High-Quality Antialiasing and Transparency. In *Graphics Hardware*, 85–93.

MATTAUSCH, O., BITTNER, J., AND WIMMER, M. 2008. CHC++: Coherent Hierarchical Culling Revisited. *Computer Graphics Forum 27*, 2, 221–230.

MOREIN, S. 2000. ATI Radeon HyperZ Technology. In *Graphics Hardware, Hot3D Proceedings*.

MUNKBERG, J., CLARBERG, P., HASSELGREN, J., TOTH, R., SUGIHARA, M., AND AKENINE-MÖLLER, T. 2011. Hierarchical Stochastic Motion Blur Rasterization. In *High Performance Graphics*, 107–118.

SHAPIRO, J. 1993. Embedded Image Coding using Zerotrees of Wavelet Coefficients. *IEEE Transactions on Signal Processing 41*, 12, 3445–3462.

STANEKER, D., BARTZ, D., AND MEISSNER, M. 2003. Improving Occlusion Query Efficiency with Occupancy Maps. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 111–118.

ZHANG, H., MANOCHA, D., HUDSON, T., AND HOFF, III, K. E. 1997. Visibility Culling Using Hierarchical Occlusion Maps. *Proceedings of SIGGRAPH*, 77–88.

Appendix

Coarse depth test The coarse depth test produces two per-sample masks – a pass mask and a fail mask. The remainder of the samples must to be tested using the regular per-sample depth test.

Listing 1: *Perform coarse depth test.*

```
function coarseZTest(tile, tri)
    failMask0 = tri.zMin >= tile.zMax[0]
        ? tri.rastMask & ~tile.mask : 0
    failMask1 = tri.zMin >= tile.zMax[1]
        ? tri.rastMask & tile.mask : 0
    failMask = failMask0 | failMask1
    passMask = tri.zMax < tile.zMin ? tri.rastMask : 0
    return [passMask, failMask]
```

An alternative version of the coarse test may be performed before the per-sample coverage test. Contrasting the version above, this test does not account for coverage. We still observe good cull rates, as the accurate update is the key to the performance of our algorithm.

Listing 2: *Perform coarse depth test without coverage mask.*

```
function coarseZTest_noMask(tile, tri)
    if tile.mask == 0:
        maxOfMax = tile.zMax[0]
    else if tile.mask == ~0:
        maxOfMax = tile.zMax[1]
    else:
        maxOfMax = max(tile.zMax[0], tile.zMax[1])
    fail = tri.zMin >= maxOfMax
    pass = tri.zMax < tile.zMin
    return [pass, fail]
```

Updating the coarse buffer The coarse buffer is trivially updated if any of the layers are overwritten, while the heuristic-based merge function is called to resolve complicated multi layered situations.

Listing 3: *Update coarse depth buffer.*

```
function coarseZUpdate(tile, tri)
    triMask0 = tri.zMax < tile.zMax[0]
        ? tri.rastMask & ~tile.mask : 0
    triMask1 = tri.zMax < tile.zMax[1]
        ? tri.rastMask & tile.mask : 0

    triMask = triMask0 | triMask1
    layer0Mask = ~tile.mask & ~triMask
    layer1Mask = tile.mask & ~triMask

    if triMask != 0:
        if layer0Mask == 0:
            // Layer 0 is empty and is replaced
            tile.zMax[0] = tri.zMax
            tile.mask = ~triMask
        else if layer1Mask == 0:
            // Layer 1 is empty and is replaced
            tile.zMax[1] = tri.zMax:
            tile.mask = triMask
        else:
            // All layers contain samples, merge
            merge(tile, tri, triMask)
```

Merging depth layers The merge function reduces three layers to two and updates the selection mask.

Listing 4: *Merging heuristic*

```
function mergeClosest(tile, tri, triMask)
    dist0 = abs(tri.zMax - tile.zMax[0])
    dist1 = abs(tri.zMax - tile.zMax[1])
    dist2 = abs(tile.zMax[0] - tile.zMax[1])
    if dist0 < dist1 && dist1 < dist2:
        // Merge triangle layer with layer 0
        tile.zMax[0] = max(tile.zMax[0], tri.zMax)
        tile.mask = tile.mask & ~triMask
    else if dist1 < dist2:
        // Merge triangle layer with layer 1
        tile.zMax[1] = max(tile.zMax[1], tri.zMax)
        tile.mask = tile.mask | triMask
    else:
        // Merge layer 0 and 1
        tile.zMax[0] = max(tile.zMax[0], tile.zMax[1])
        tile.zMax[1] = tri.zMax
        tile.mask = triMask
```