Pixel Merge Unit

Rahul Sathe[1] and Tomas Akenine-Möller[1,2]

[1]Intel Corporation [2]Lund University

Figure 1: *Left: a frame from Heaven 3.0 running in our simulator, i.e., a ground truth image. Center: using our pixel merge unit, the resulting image is nearly identical to the ground truth but with about 12% less shading. Right: scaled difference (*$100\times$*) between the left and the center. With higher tessellation settings for this frame, the reduction in shading goes up to about 15%.*

Abstract

Multi-sample anti-aliasing is a popular technique for reducing geometric aliasing (jagged edges) and is supported in all modern graphics processors. With multi-sampling anti-aliasing, visibility and depth are sampled more than once per pixel, while shading is done only once per pixel per primitive. Although this significantly reduces the appearance of jagged edges around object boundaries, the image quality improvement in non-silhouette regions is hardly noticeable. We propose a hardware unit, called the pixel merge unit, which is located just after the early depth test unit but before the pixel shader. Our unit attempts to reduce the shading rate to once per pixel per group of connected primitives covering a pixel using a novel merging strategy. We demonstrate up to 15% reduction in pixel shader executions. Given the simple implementation that we propose, this is a substantial reduction.

1. Introduction

For high-performance graphics, it is essential to innovate in the space of more efficient hardware algorithms for each new generation of graphics processors. However, the same holds for mobile phones and tablets, where more and more performance is desired at the same or less power usage. Interestingly, power reductions are also crucial for discrete and integrated GPUs. Since a large portion of the power in a graphics processor is due to pixel shading [Poo12], a reduction here will, more or less, translates directly to power reductions and performance improvements.

To provide high image quality at a relatively low cost, multi-sampling anti-aliasing (MSAA) [Ake93] performs shading computations only once per pixel per primitive, while visibility and depth are computed at a higher rate. Within a pixel, MSAA uses the same calculated color for all the samples covered by that primitive. The pixel shader units are followed by the output merger (OM) unit. The OM is responsible for blending and it works on a per-sample basis. After the main shading pass, a resolve pass averages the colors at every sample within a pixel and displays the result.

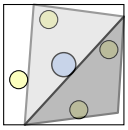
MSAA improvements are highly noticeable around the object silhouettes, where typically there are color discontinuities in addition to depth discontinuities. However, even though MSAA produces different colors for neighboring primitives for internal edges (not silhouettes), typical color variation is minimal and hardly noticeable after the resolve pass. We exploit this key observation by proposing a unit, located before the pixel shader stage, that merges partially covered fragments within the pixel boundary with the fragments from connected neighboring primitive(s). In other words, pixel shading of partially covered fragments is deferred until their neighbors arrive and the pixel becomes fully covered, in which case, our pixel merge unit saves a considerable amount of work.

2. Previous Work

In this section, we will only mention the most important work that is relevant to our research. Supersampling [FGH*85] associates one visibility sample with one shading sample, and computes the shading at every sample location within a pixel. Although this generates the best-looking images, it is also the most expensive technique.

MSAA [Ake93] is a less expensive version, as described in the introduction. More recently, Direct3D 11.1 has introduced target independent rasterization as a means to achieve the image quality of MSAA without forcing color and depth buffers to have high resolutions. Jouppi and Chang [JC99] presented the Z^3 algorithm, which samples visibility at a higher rate and allocates a small list of fixed number of fragments per pixel. When the list overflows, fragments are merged using a heuristic to minimize artifacts. The per-pixel list can be replaced with a dynamic linked-list [LK00], much like a real A-buffer [Car84]. Kerzner and Salvi [KS14] proposed a software solution to reduce the G-buffer generation bandwidth by lossy compression of the *surfaces* covering a pixel. Fatahalian et al. [FBH*10] proposed a quad fragment merging (QFM) unit that merges sparsely covered quad fragments rasterized from a single primitive into a single densely covered quad fragment before shading. While QFM is the algorithm that is most similar to our research, it has some key differences, which are discussed in the end of Section 3.

3. Algorithm



Before our algorithm is described, let us define some common terms. A *sample* is a location within a pixel where visibility and depth/stencil are evaluated. A *fragment* is defined as a portion of a triangle with non-zero sample coverage within a pixel, and a *quad fragment* (QF) is a 2×2-block corresponding to a fragment. The core idea of our pixel merge strategy is to let the fragments that overlap with a pixel center share its shading computed at that pixel center to neighboring fragments that do not overlap with that pixel center. A simple example is shown to the right with a single pixel with four (yellow) visibility samples and a single shading sample point (blue) in the center of the pixel. The left triangle overlaps the pixel center, and so it computes the shading there. The next triangle shares an edge with the previous triangle, and it overlaps two visibility samples but not the pixel center. Normally, the second triangle would compute new shading at the center of the pixel, but our algorithm shares the pixel shading from the first triangle with the second for these two samples. Note that our algorithm is only applicable when MSAA is enabled. With MSAA disabled, only the first triangle would have gotten shaded in the figure above.

Our pixel merge unit (PMU) is located after the early Z-test unit, but before the pixel shading unit. This can be seen in Figure 2, where the PMU has been added to a standard graphics pipeline. We impose the following restrictions on when it is possible to share shading between two triangles, where one triangle has computed shading at a certain pixel center, and the second has not:

1. triangles must be facing the same way,
2. the triangles' coverage may not overlap, and
3. the triangles must share an edge.

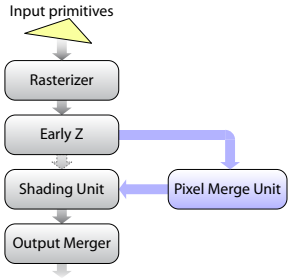


Figure 2: *The gray units are parts in a standard graphics pipeline. Our additions are marked in light blue, and the connection between the Early Z and the (pixel) Shading Unit has been grayed out, since it is replaced by the new, blue connections.*

Adjacency, i.e., whether two triangles share an edge, is tracked using a pair of vertex identifiers. For tessellated triangles within a patch, one can use the *uv*-coordinates. Alternatively, a bitmask, similar to the one proposed by Fatahalian et al. [FBH*10], could be used. However, that would require the hardware tessellator to output the adjacency mask. To track the adjacency across patches, one can use corner identifiers [SFS14]. A longer example of our algorithm in action is illustrated in Figure 3.

We disable the PMU if the pixel shader has any of the following:

1. discard instruction(s),
2. writes to unordered access views (UAVs),
3. writes to the depth,
4. alpha to coverage, or
5. uses the system generated primitive id or the coverage.

The reason for this is that the PMU is located before the pixel shader, and the pixel shaders above may change the outcome of whether the fragment passes or not, and in the case of UAVs, it may not possible to know (statically) where the output is written. For example, if the pixel shader writes a custom depth, then the PMU cannot know whether the fragment passes or not. Also, the coverage or the primitive ids are different in the pixel shader if the merge is successful and can result in very different shading.

Discussion Our algorithm is similar to quad fragment merging (QFM) [FBH*10] in the criteria used for merging, but there are some important differences. The primary goal of QFM was to avoid over-shading caused by the helper pixels for micropolygons, whereas our primary goal is to avoid shading along the internal edges for MSAA. We merge fragments within the pixel boundary, while QFM merges quad fragments over a 2×2 pixel region. Our algorithm never picks up shading samples from more than one primitive, while QFM can have shading samples from up to four primitives. So QFM requires shader cores to have large number of registers because it must be possible to shade multiple primitives in *one* SIMD batch, which

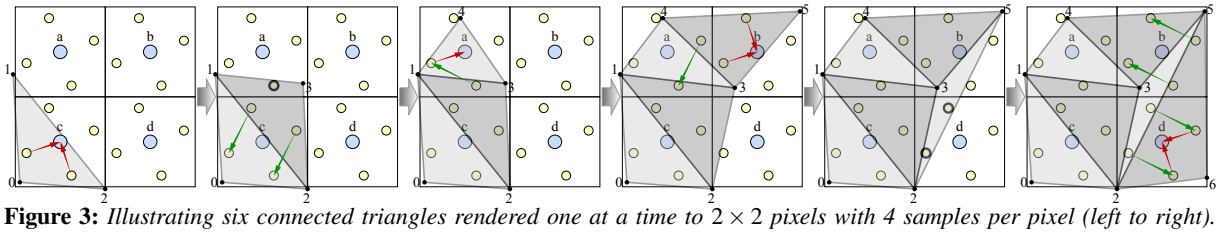


Figure 3: *Illustrating six connected triangles rendered one at a time to 2×2 pixels with 4 samples per pixel (left to right). Red arrows means that shading is computed at the center of a pixel (light blue circles), and is stored in the corresponding light yellow samples. Green arrows indicate that samples "steal" their color from another triangle (where the arrow head points on the sample being stolen). In the second figure (from left), one yellow sample is marked bold, indicating that it has not yet obtained any color. The third figure shows how the shading is computed in the center of the pixel, and then one yellow sample obtains that color (red arrow), and the bold sample from figure 2 then obtains that color as well (green arrow). Note that the two yellow samples in figure five are also marked as not yet having obtained a color. In this entire example, only four quads will be shaded with the PMU, which should be compared to six without, i.e., a 33% reduction.*

the PMU does not need. Our logic to decide what shading samples to use is simple, i.e., always use the pixel center (covered or extrapolated). The QFM logic is substantially more complicated. In their algorithm, derivatives for all the surviving fragments can change after the merge, which can result in incorrect mip-map level being used. In contrast, for our algorithm the surviving fragments have correct derivatives, only the coverage is different. However, the most significant difference is that QFM targets micropolygons, while we target medium- and large-sized triangles too.

4. Implementation

The PMU is implemented as a finite-sized FIFO merge buffer where every entry corresponds to a quad containing a partially covered fragment, along with the other information about the fragment as shown below:

```
1   #define MAX_EDGES 8
2   struct Edge { unsigned int index1, index2; }
3
4   struct PMUEntry {
5     unsigned short  x, y;     // pixel location
6     unsigned int coverage[4]; // coverage
7     bool facing;              // front or back
8     Edge outerEdges[4][MAX_EDGES];
9     ShadingInput s;           // Shading Data
10  }
11
12  // Merge coverage only within a pixel (p)
13  Merge(PMUEntry e1, const PMUEntry e2, uint p) {
14    e1.coverage[p] |= e2.coverage[p];
15    MergeOuterEdges( e1, e2, p);
16  }
17
18  // For non-tesellated triangles
19  MergeOuterEdges(PMUEntry e1, PMUEntry e2, uint p) {
20    foreach (ed1 in e1.outerEdges[p])
21      foreach (ed2 in e2.outerEdges[p])
22        if (ed1.index1 == ed2.index2 &&
23            ed1.index2 == ed2.index1) {
24          RemoveEdge(e1, ed1, p);
25          RemoveEdge(e2, ed2, p);
26        }
27    AddEdges(e1, e2, p) // Adds e2 edges to e1
28  }
```

Only the quads with partially covered fragments enter into the PMU along with the information listed in *PMUEntry*. *ShadingInput* contains attribute data at the three triangle vertices along with the barycentric coordinates at the pixel location. The vertex data can be stored in a separate reference counted buffer to reduce storage requirements further.

When the rasterizer generates the quads (fully or partially covered), the PMU first checks whether the incoming quads overlap with any quads in the PMU and evicts those in the order they had entered the buffer. This ensures that all the fragments always get rendered in the triangle submission order for any given pixel. We restrict our merge to fragments within a pixel. If no fragment in the quad is partially covered (even though the quad itself may be partially covered), there will never be any merge. We do not buffer such quads in the PMU. If the conditions (listed in Section 3) for the merge are met, we perform the merge using the pseudocode listed in *Merge*, such that the fragment covering the pixel center (or the sample closest to the pixel center) is the one that survives. Note that, a single quad fragment can result in multiple merges, one for each fragment in the quad fragment. After the merge, if all its fragments become fully covered or empty, there is no reason to wait for more merges. We mark such a quad fragment for eviction and override the FIFO policy. The merge buffer is flushed after every instance of the draw call.

5. Results

We have evaluated our pixel merge unit (PMU) using a functional simulator of the D3D11 pipeline, augmented with the PMU. The scenes we have used are a set of representative games and benchmarks. Heaven 3.0 has different tessellation settings, namely, moderate (HeavenM), normal (HeavenN), and extreme (HeavenE).

The goal of our technique is to reduce shading along internal edges to once per pixel per group of connected primitives. We measure the total number of partially covered quad fragments that get rendered, with and without our technique for different finite-sized merge buffers. We call the ratio of partially covered quads that we save to the total number of partially covered quads, the *efficiency* of our scheme. This

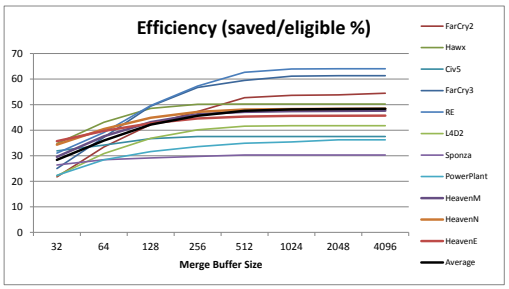


Figure 4: *Efficiency measures the percentage reduction in the number of quads relative to the number of partially covered quads. Note that the partially covered quads also contains the silhouette pixels which never get merged. Colored lines show the PMU efficiency for individual titles and the bold black line indicates the average.*

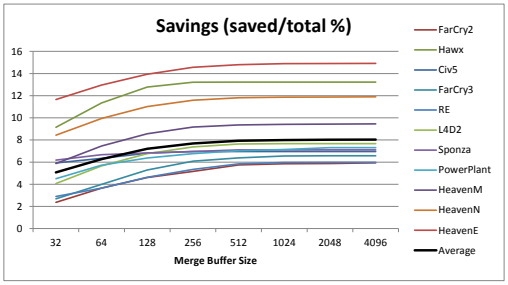


Figure 5: *The percentage reduction in the number of quads relative to the total number of quads that get shaded as a function of number of entries in PMU. Colored lines indicate the savings for individual titles and the bold black line indicates the average reduction in pixel shading.*

metric is conservative in that, we also count the true silhouette quad fragments in our baseline. The silhouette quad fragments must be shaded once per pixel per primitive. As seen in Figure 4, the efficiency increases (up to 64%) with larger merge buffer sizes, as expected. In Figure 5, the savings in percentage of total number of quads dispatched are shown. Both the metrics start to level off when the merge buffer has 512 entries. As expected, for scenes with smaller triangles, we see larger savings because there are more edge quads than internal quads. This is reflected in larger savings for Hawx, for example. Similarly, we see savings increase for Heaven as the tessellation rate increases from moderate to extreme. On average, our PMU provides 8% savings for 512 entries.

Typically, hardware vendors employ some kind of compression scheme for MSAA color buffers [RHAM07]. A side benefit of our technique is that it tends to generate render targets that compress better. This is because all samples in the pixels along the internal edges, where merges were successful, have the same color. This directly translates to bandwidth reduction. However, we did not measure this bandwidth reduction, as it correlates to the shading reduction. Also, in general, our image quality is very high, e.g., 48.57 dB for Heaven with extreme levels of tessellation.

Our algorithm also extends nicely to coarse pixel shading (CPS) [VST*14], which is a generalization of MSAA. For a given buffer size and a screen resolution, the pixel shader execution savings increase when using CPS. For the PowerPlant scene with a merge buffer size of 64, for example, the savings increase from 5.72% to 7.42% using CPS.

We have presented a simple hardware unit, the PMU, to avoid unnecessary shading along internal edges. On average, the PMU provides 8% savings in pixel shader work, but up to 15% for some workloads. This is substantial, given the simplicity of the unit, and will directly translate to increased performance and lower power usage.

Acknowledgements Thanks to David Blythe and the Advanced Rendering Technology group at Intel. Tomas Akenine-Möller is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg foundation.

References

[Ake93] AKELEY K.: RealityEngine Graphics. In Proceedings of SIGGRAPH 93 (1993), ACM, pp. 109–116. 1, 2

[Car84] CARPENTER L.: The A-buffer, an Antialiased Hidden Surface Method. In Computer Graphics (Proceedings of SIGGRAPH 84) (1984), vol. 18, ACM, pp. 103–108. 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. 2

[FGH*85] FUCHS H., GOLDFEATHER J., HULTQUIST J. P., SPACH S., AUSTIN J. D., BROOKS JR. F. P., EYLES J. G., POULTON J.: Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes. In Computer Graphics (Proceedings of SIGGRAPH 85) (1985), vol. 19, ACM, pp. 111–120. 1

[JC99] JOUPPI N. P., CHANG C.-F.: Z^3: An Economical Hardware Technique for High-Quality Antialiasing and Transparency. In Graphics Hardware (1999), pp. 85–93. 2

[KS14] KERZNER E., SALVI M.: Streaming G-Buffer Compression for Multi-Sample Anti-Aliasing. In High Performance Graphics (2014), pp. 1–7. 2

[LK00] LEE J.-A., KIM L.-S.: Single-pass Full-screen Hardware Accelerated Antialiasing. In Graphics Hardware (2000), ACM, pp. 67–75. 2

[Poo12] POOL J.: Energy-Precision Tradeoffs in the Graphics Pipeline. PhD thesis, 2012. 1

[RHAM07] RASMUSSON J., HASSELGREN J., AKENINE-MÖLLER T.: Exact and Error-Bounded Approximate Color Buffer Compression and Decompression. In Graphics Hardware (2007), pp. 41–48. 4

[SFS14] SATHE R., FOLEY T., SALVI M.: Post-Tessellation Geometry Caches. In Eurographics 2012 – Short Papers (2014), pp. 57–60. 2

[VST*14] VAIDYANATHAN K., SALVI M., TOTH R., FOLEY T., AKENINE-MÖLLER T., NILSSON J., MUNKBERG J., HASSELGREN J., SUGIHARA M., CLARBERG P., JANCZAK T., LEFOHN A.: Coarse Pixel Shading. In High Performance Graphics (2014), pp. 9–18. 4