Reducing Shading on GPUs using Quad-Fragment Merging

Kayvon Fatahalian* Solomon Boulos* James Hegarty*
Kurt Akeley† William R. Mark‡ Henry Moreton§ Pat Hanrahan*

Abstract

Current GPUs perform a significant amount of redundant shading when surfaces are tessellated into small triangles. We address this inefficiency by augmenting the GPU pipeline to gather and merge rasterized fragments from adjacent triangles in a mesh. This approach has minimal impact on output image quality, is amenable to implementation in fixed-function hardware, and, when rendering pixel-sized triangles, requires only a small amount of buffering to reduce overall pipeline shading work by a factor of eight. We find that a fragment-shading pipeline with this optimization is competitive with the REYES pipeline approach of shading at micropolygon vertices and, in cases of complex occlusion, can perform up to two times less shading work.

CR Categories: I.3.1 [Computer Graphics]: Hardware architecture—Graphics processors

Keywords: GPU architecture, real-time rendering, micropolygons

1 Introduction

To create surfaces with fine geometric detail, artists increasingly produce dense polygon meshes. In addition, this year the Direct3D 11 standard has introduced a tessellation stage into the graphics pipeline. This stage samples surfaces to produce triangle meshes.

Unfortunately, current GPUs shade small triangles inefficiently. Rasterized triangles contribute fragments to each pixel they overlap. When triangles are small, many pixels contain multiple fragments due to partial overlap. Each of these fragments is shaded, leading to pixels being shaded redundantly. The severity of this "over-shade" problem is shown in Figure 1, where the same scene is rendered with decreasing polygon size. When the scene is tessellated into pixel-sized triangles, a GPU will shade each covered pixel more than eight times (once for each overlapping fragment). Given that shading is the major component of most rendering workloads, this significantly increases rendering cost.

We augment the GPU pipeline to reduce the amount of redundant shading performed when rendering small triangles, which we assume are generated by tessellation in a prior pipeline stage. To reduce shading, we merge fragments at the same pixel, but from different triangles, in a fixed-size buffer between the rasterization and shading stages. We merge fragments only if they satisfy a set of compatibility rules, including triangle adjacency, which ensure they can be adequately shaded by a single calculation. When processing sub-pixel area triangles, merging fragments reduces shading work by a factor of eight while maintaining image quality.

*Stanford University
†Microsoft Research
‡Intel Labs
§NVIDIA Corporation

ACM Reference Format
Fatahalian, K., Boulos, S., Hegarty, J., Akeley, K., Mark, W., Moreton, H., Hanrahan, P. 2010. Reducing Shading on GPUs using Quad-Fragment Merging. *ACM Trans. Graph. 29*, 4, Article 67 (July 2010), 8 pages. DOI = 10.1145/1778765.1778804 http://doi.acm.org/10.1145/1778765.1778804.

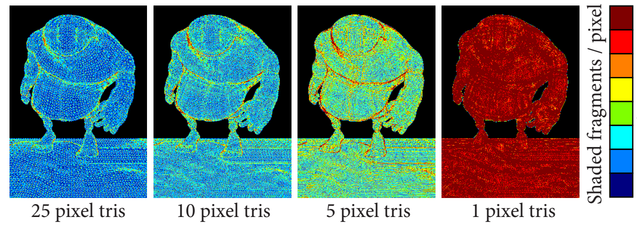

Figure 1: *Over-shade in a GPU pipeline increases as scene triangle size shrinks (images are colored according to the number of fragments shaded per pixel). When this scene is tessellated into pixel-sized triangles, each pixel is shaded more than eight times.*

2 Background

GPU pipelines [Blythe 2006], as well as off-line rendering architectures such as REYES [Cook et al. 1987], employ three important techniques that limit the number of shading computations needed to generate high-quality images.

- **Independent visibility and shading sampling densities**. Geometric screen coverage is sampled at a higher rate than shading to anti-alias edges without increasing shading work. It is sufficient to sample shading more sparsely than coverage because high-frequency content from textures is pre-filtered.
- **Derivatives via finite differencing**. Filter extents for texturing are computed via finite differencing of texture coordinates between neighboring shading samples. Sharing data between neighbors avoids re-computation when finite-difference derivatives are needed during shading.
- **Early occlusion culling**. Surface regions that are not visible to the camera due to occlusion are discarded from the rendering pipeline prior to shading.

We now describe how these techniques are implemented for both the GPU and REYES pipelines. These architectures have very similar structure, but differ notably in their approach to shading.

2.1 GPU Shading

GPU pipelines shade each triangle uniformly in screen space at a density of one shading sample per pixel. Visibility may be sampled at a higher rate than shading to reduce aliasing at triangle edges (multi-sample anti-aliasing [Akeley 1993]). The process of rasterizing and shading a triangle using 4× multi-sample anti-aliasing (MSAA) is illustrated in Figure 2. In this example, the rasterizer samples triangle coverage at four *multi-sample locations* within each pixel. Panel 1 of this figure shows a 4x4 pixel region of the screen. Multi-sample locations are shown as black dots. Panel 2 highlights the multi-sample locations covered by a triangle in red.

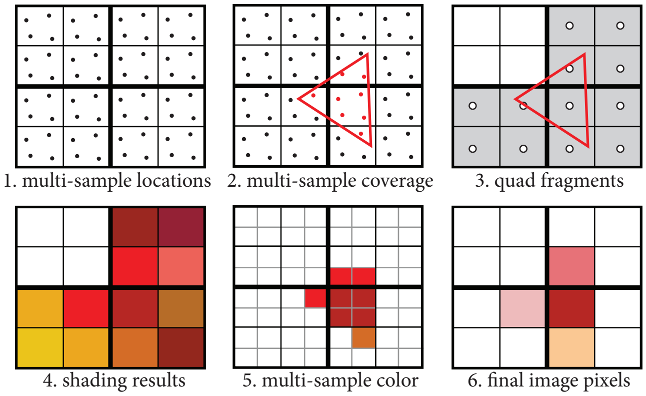

Figure 2: *Rendering a triangle to a 4x4 pixel screen region using 4× multi-sample anti-aliasing: The triangle's screen coverage is sampled four times per pixel (panels 1,2). Shading is sampled once per pixel at 2x2 pixel granularity (3,4). The results of visibility and shading computations are stored in the multi-sample buffer (5) and subsequently filtered to produce final image pixels (6).*

If any multi-sample location in a pixel is covered by the triangle, a shading computation must be performed for that pixel using information from the triangle. Shading inputs such as texture coordinates are interpolated from values stored at triangle vertices, using the location of the pixel center as the interpolation/sampling point [Kessenich 2009; Microsoft 2010]. Panel 3 shows the pixel centers as dots. If the pixel center lies outside the triangle, the shading inputs are extrapolated. Alternatively, GPUs permit shading inputs to be sampled at the covered multi-sample location which is closest to the pixel center (centroid sampling [Kessenich 2009; Microsoft 2010]). Centroid sampling avoids the need to extrapolate shading inputs, but results in a non-uniform screen-space sampling of shading near triangle edges.

The information needed to compute a triangle's shading at a pixel is encapsulated in a record called a *fragment*. This information consists of shading inputs, along with triangle coverage and depth information for each of the pixel's multi-sample locations. (For convenience, we say a fragment "covers" a multi-sample location if the triangle it was created from did). To support derivative estimates using finite differencing, rasterization generates fragments in 2x2 pixel blocks [Akeley 1993]. We refer to blocks of four fragments as *quad fragments*. Panel 3 shows the three quad fragments generated by rasterizing the triangle in gray. It also shows shading sample locations for each fragment (white dots). Notice that if the triangle covers any multi-sample location in a 2x2 pixel region, a quad fragment is generated at these pixels, and shading is computed at all four corresponding pixel centers. The results of shading each fragment are given by the colored pixels in panel 4.

Unique color and depth values are stored for each multi-sample in the frame buffer. After a fragment has been shaded, its color is stored in all multi-samples covered by the fragment (panel 5). Finally, at the end of the frame (after all rasterization and depth testing is complete) the colors of multi-samples within each pixel are filtered to produce a final rendered image (panel 6).

GPU shading is efficient when triangles are large. Most quad fragments are covered entirely by the triangle and the overhead of shading extra fragments near triangle edges is low. This overhead increases as triangle size shrinks. For example, the triangle in Figure 2 is approximately two pixels in area, but it causes a GPU to shade twelve fragments.

2.2 REYES Shading

Unlike GPUs, the REYES architecture [Cook et al. 1987] shades micropolygon vertices prior to rasterization. To shade approximately once per screen pixel, REYES must tessellate surfaces into micropolygons approximately one pixel in area. In REYES, tessellation produces a stream of *grids*. Although the term grid originally referred to a regular matrix of quadrilateral micropolygons [Apodaca and Gritz 2000], in modern REYES implementations, a grid is simply a collection of micropolygons with adjacency information.

Grids are the minimal granularity of shading work in REYES. Grid vertices are shaded together, permitting efficient data-parallel execution and computation of derivatives via finite differencing (adjacent vertices in a grid are known). Grids are also the granularity of culling: either an entire grid is discarded prior to shading, or all vertices in the grid are shaded. Thus there is tension between the need to make grid sizes large (to increase the data-parallel efficiency of shading computations and to reduce redundant shading at grid boundaries) and the desire to keep grids small for culling (to eliminate unnecessary shading of occluded surfaces).

It is simple for a REYES pipeline to sample visibility at a higher rate than shading because shading occurs prior to rasterization. During rasterization, surface color at shaded vertices is interpolated onto each visibility sample point covered by a micropolygon. Shading prior to rasterization is also fundamental to the REYES pipeline's support for advanced rendering effects such as motion and defocus blur. Incorporating these features into a GPU fragment shading pipeline remains an area of active research [Ragan-Kelley et al. 2010] and is not attempted in this work.

2.3 Evolving the GPU

Although REYES provides an efficient and proven solution for shading micropolygons, we have chosen to explore the option of evolving the GPU pipeline to support real-time micropolygon rendering. Our motivations included the following:

- Achieving real-time performance. Researchers have ported aspects of the REYES pipeline to GPUs [Wexler et al. 2005; Patney and Owens 2008] or even the full pipeline [Zhou et al. 2009]. However, REYES rendering performance is still far from meeting real-time requirements. Evolving the highly-optimized pipeline architecture of current GPUs (rather than porting REYES to run as a GPU compute application) seemed more likely to achieve our performance goals.

- Retaining fine-grain occlusion culling. GPUs shade after rasterization, allowing hierarchical depth algorithms [Greene et al. 1993] to efficiently cull occluded fragments at granularities approaching single pixels. We hoped to retain this efficiency.

- Maintaining development continuity. Evolving GPUs and their current rendering algorithms would allow current GPU-based applications to transition gradually toward micropolygon rendering, at all times trading off quality and performance to optimize user experience.

Current GPUs shade fragments immediately after rasterization, rather than postponing shading until all occlusions are resolved at the frame buffer. Such "deferred shading" [Deering et al. 1988] is a tempting optimization that offers the promise of executing exactly one shading operation per pixel. However, it is eschewed by GPU architects as a core pipeline mechanism because it interacts badly with multi-sample anti-aliasing (recall, pixels containing object silhouettes must be shaded more than once to obtain smooth edges).

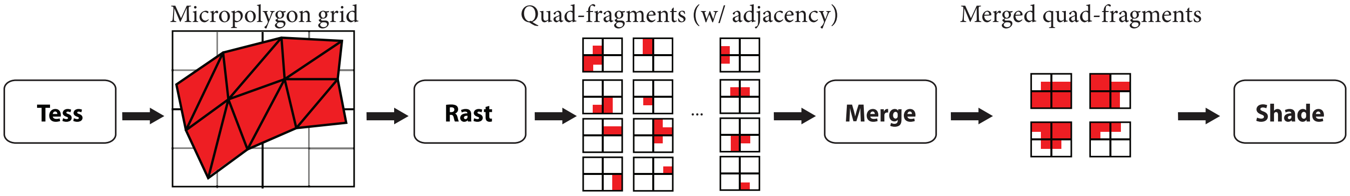

Figure 3: *Our quad-fragment merging pipeline: tessellation produces grids of triangles with adjacency. The triangles are rasterized to produce quad fragments. The merging unit buffers and combines rasterized quad fragments. Merged quad fragments are emitted for shading.*

```
quad_fragment {
  int           x, y;
  bool          facing;

  BITMASK       coverage;     // 4*MULTISAMPLES bits
  float         z[4][MULTISAMPLES];

  BITMASK       shade_coverage;            // 4 bits
  SHADER_INPUT  shade_input_data[4];
};

buffer_entry {
  quad_fragment frag;
  BITMASK       tri_mask;                // 512 bits
  BITMASK       adj_mask;                // 512 bits
};

bool can_merge(e1, e2) {
  return
      e1.frag.x == e2.frag.x &&
      e1.frag.y == e2.frag.y &&
      e1.frag.facing == e2.frag.facing &&
     (e1.frag.coverage & e2.frag.coverage) == 0 &&
     (e1.tri_mask & e2.adj_mask);
}

// merge quad-fragment in entry e2 into e1
void merge(e1, e2) {
  select_shading_inputs(e1.frag, e2.frag);
  copy_z(e1.frag, e2.frag);
  e1.frag.coverage |= e2.frag.coverage;
  e1.tri_mask |= e2.tri_mask;
  e1.adj_mask |= e2.adj_mask;
}
```

Figure 4: *Each merge-buffer entry (*`buffer_entry`*) contains a quad fragment and bitmasks enumerating the quad fragment's source triangles and the triangles that are adjacent to source triangles. Determining whether two buffer entries can be merged (*`can_merge`*) involves only a few bitwise operations.*

To support anti-aliasing, previous hardware implementations of deferred shading compute shading at each multi-sample location, rather than once per pixel [Molnar et al. 1992]. While some game engines implement deferred shading as a software layer running on GPUs, they do so by disabling multi-sample anti-aliasing and accepting the resulting loss in image quality. Direct multi-sample frame-buffer access in Direct3D 10.1 [Microsoft 2010] allows software approaches to provide limited multi-sample anti-aliasing support, but requires shading inputs to be computed and stored at all multi-sample locations and does not robustly support finite-difference derivatives in shaders. In light of these drawbacks, we do not attempt evolving the GPU pipeline toward deferred shading.

3 Quad-fragment Merging

To reduce the overall cost of shading micropolygons in a GPU pipeline, we propose merging rasterized quad fragments (with sparse multi-sample coverage) into a smaller number of quad fragments (with dense multi-sample coverage) *prior* to shading. Our technique, which we refer to as quad-fragment merging, is illustrated by the modified GPU pipeline in Figure 3.

In this pipeline, grids are generated from surface patches via tessellation. Our notion of a grid is similar to that of REYES. A grid is a group of triangles with adjacency information provided by the tessellator. A merging unit, which sits between rasterization and shading, buffers rasterized quad fragments from a grid. In the merging unit, quad fragments at the same screen location are identified and merged to reduce the total amount of pipeline shading work. To avoid aliasing or derivative artifacts, *only quad fragments from grid triangles sharing an edge are merged.* The output of the merging unit is a stream of quad fragments for shading. Unlike the quad fragments produced by rasterization, merged quad fragments contain shading inputs and multi-sample coverage information from multiple triangles.

The primary component of the merging unit is a fixed-size buffer that stores quad fragments for merging (the merge buffer). To facilitate the description of merging-unit behavior, we provide C-style definitions of a quad-fragment record (`quad_fragment`) and a merge-buffer entry (`buffer_entry`) in Figure 4. Quad fragments consist of coverage and depth information at each multi-sample location, as well as input data for shaders (`shade_input_data`). Shading inputs are defined by fragment shader signatures and include all interpolated attributes (e.g. texture coordinates, position, normal). Source triangle sidedness (`facing`) as well as its coverage of each fragment's shading sample location (`shade_coverage`) are also stored in the quad-fragment record.

Each merge-buffer entry contains a quad fragment and two bitmasks. The source triangle mask (`tri_mask`) enumerates triangles that have contributed to this quad fragment. Bit i in the mask is set if the entry's quad fragment was generated by rasterizing grid triangle i, or if quad fragments from triangle i have been merged into this entry. Bits in the adjacent triangle mask (`adj_mask`) are set for all triangles that share an edge with source triangles. In our implementation, these masks are sized to support grids of up to 512 triangles (512-bit masks), but could be sized differently to permit merging quad-fragments from larger or smaller groups of triangles.

When a quad fragment arrives as input from the rasterizer, the merging unit checks to see if the incoming quad fragment can be merged with an existing entry in the merge buffer. The merging unit first determines the set of quad fragments in the buffer with the same screen-space location and sidedness. In hardware, this can be implemented with a content addressable memory (CAM). If the set is non-empty, the merging unit then executes `can_merge` for the N most recently added quad fragments in the set (in our implementation $N = 2$). If `can_merge` returns true, the new quad fragment is merged. Otherwise, the quad fragment is placed in an available buffer entry. If no entries are available, a buffer entry is chosen for eviction (using a FIFO policy) and submitted to the shading system.

In the next two sections, we describe the conditions required to merge two quad fragments and the process of constructing a merged quad fragment in more detail.

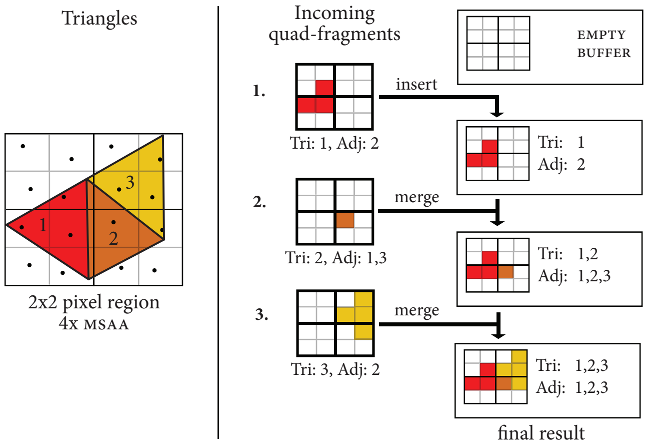

Figure 5: *Merging unit behavior on a stream of three quad fragments from adjacent grid triangles. The first arriving quad fragment is inserted into the merge buffer. The quad fragment from triangle 2 is merged with the buffered quad fragment from triangle 1 (these triangles share an edge). Last, the quad fragment from triangle 3 is also merged with the buffered quad fragment because triangles 2 and 3 share an edge.*

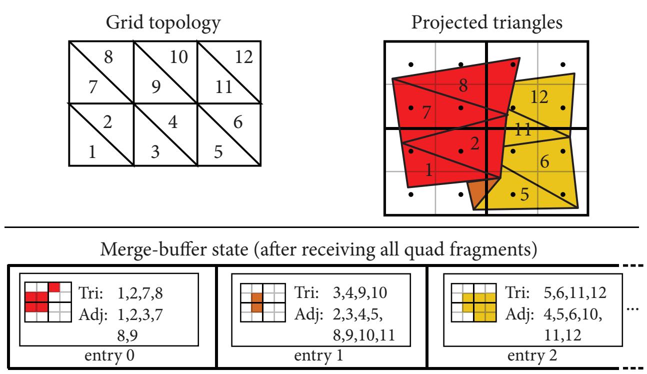

Figure 6: *Merging does not occur across the grid's silhouette edge because triangles in the red region of the surface do not share an edge with triangles colored in yellow. Grid triangles (3,4,9,10) are back-facing so their quad fragments cannot be merged with those of the red or yellow groups.*

3.1 Conditions for Merging

Two quad fragments may be merged if they meet four conditions:

1. They have the same screen-space location.
2. They do not cover the same multi-sample location (they do not occlude each other).
3. They have the same sidedness (either front or back-facing).
4. Their source triangles are connected via edges.

For a pair of quad fragments, these four conditions can be checked using simple bitwise operations (see function `can_merge`, Figure 4). Figure 5 shows merging unit behavior for a sequence of three triangles that fall in the same 2x2 pixel region (we label the triangles by the order they arrive at the merging unit). First, the quad fragment from triangle 1 is inserted into the merge buffer (there are no quad fragments to merge with). Next, the quad fragment from triangle 2 is merged with the quad fragment in the buffer because triangle 2 is in the entry's adjacent triangle mask (triangle 2 shares an edge with triangle 1). Finally, the quad fragment from triangle 3 merges with the buffered quad fragment (triangle 3 shares an edge with triangle 2). In this example, shading only the merged quad fragment, rather than each rasterized quad fragment, reduces shading work by three times.

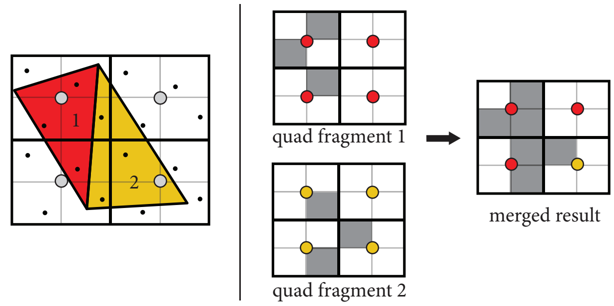

Figure 7: *Three cases for selecting shading inputs for a merged quad fragment (pixel centers are colored according to the quad fragment from which shading inputs are selected). Case 1: the top-left and bottom-right fragments receive shading inputs from the quad fragments that cover the pixel center. Case 2: the bottom-left fragment is assigned shading inputs from quad fragment 1, because triangle 1 covers the closest covered multi-sample location to the pixel center. Case 3: the top-right fragment is not covered. It receives inputs from quad fragment 1 because its horizontal neighbor (the top-left fragment) is covered by quad fragment 1.*

The conditions above prevent merging across many types of discontinuities, such as silhouettes or folds. In Figure 6, quad fragments for the red portion of the grid originate from triangles that do not share an edge with triangles in the yellow region. The quad fragments from these two groups of triangles are not merged. In the top right pixel, the red and yellow shading results will be blended together, resulting in an anti-aliased edge. Additionally, by shading these quad fragments separately, shader derivatives are representative of actual screen-space derivatives for each group of triangles. In this example, the occlusion and sidedness checks alone would not have prevented merging.

In many cases, the merging rules produce a single quad fragment for each grid that overlaps a 2x2 pixel region of the screen. Clearly, if grid geometry exhibits high-frequency detail (e.g. a bumpy surface tessellated into triangles much smaller than a pixel), shading it once per pixel may cause aliasing. In contrast to current GPUs, which perform extra shading in this case, quad-fragment merging limits shading costs and requires geometry to be properly pre-filtered to avoid aliasing.

3.2 Performing Merges

There are two steps required to merge quad fragments contained in buffer entries $e1$ and $e2$ (see function `merge`, Figure 4). First, shading inputs for each fragment in the merged quad fragment must be assigned. While quad fragments emitted by rasterization contain only a single source triangle, this is not true for merged quad fragments. Because of this, the shading input for each fragment in the merged quad fragment must be chosen from the two input quad fragments. Figure 7 illustrates the three selection cases:

1. The pixel center is covered (the top-left and bottom-right fragments). Shading inputs are selected from the quad fragment that covers the pixel center.
2. The pixel center is not covered, but multi-sample locations within the pixel are covered (the bottom-left fragment). Shad-

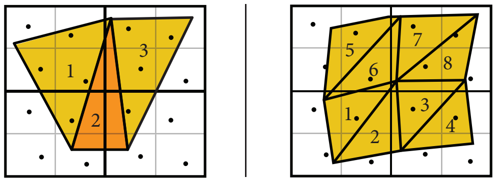

Figure 8: *Left: The pipeline rasterizer is modified to emit a quad fragment with an empty coverage mask when triangles fall within a 2x2 region but do not cover a multi-sample location (triangle 2). This quad fragment facilitates merging quad fragments from triangles 1 and 3. Right: The quad fragment from triangle 5 cannot merge with the quad fragment from 1,2,3, and 4 until after the arrival of the quad-fragment from triangle 6. Robustness to suboptimal triangle ordering is increased by attempting merges prior to evicting quad fragments from the merging unit.*

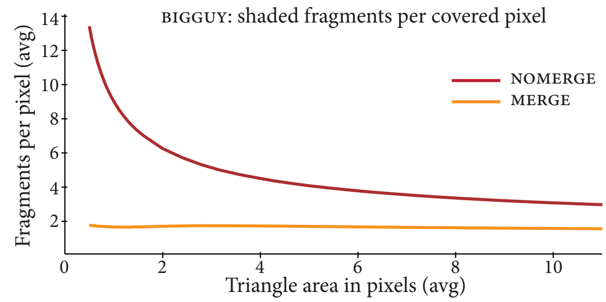

Figure 9: *When* BIGGUY *is tessellated into 0.5-pixel-area triangles,* NOMERGE *generates nearly 14 fragments per covered screen pixel. The* MERGE *pipeline shades only 1.8 fragments per pixel. Even when triangles are ten pixels in area,* MERGE *provides approximately a 2× reduction in shading work.*

ing inputs are selected from the quad fragment with the closest (relative to the pixel center) covered multi-sample location.

3. The pixel center is not covered, and no multi-sample locations are covered (the top-right fragment). Shading inputs are selected from the same quad fragment as a neighboring covered fragment (either the horizontal, vertical, or diagonal neighbor in order).

If the pixel center is not a multi-sample location, it is possible for both quad fragments to overlap the pixel center but have non-overlapping coverage masks. In this case, the shading input is chosen from the first quad fragment submitted to the merging unit.

Second, the coverage and topology bitmasks, as well as depth information, must be merged. The bitwise operations needed to combine the coverage and topology information are given in the pseudocode for `merge`. Merging the depth values is a data selection controlled by the coverage masks (recall that the merging rules prevent the coverage masks from having the same bit set). Once the merged coverage mask is determined, it is possible that the quad fragment is fully covered (i.e. that all multi-sample locations in the quad fragment are covered). In this case, the merging unit evicts the quad fragment, submitting it for shading immediately, because no more merges are possible.

3.3 Optimizations

Figure 8-left demonstrates a situation where point-sampled visibility information can cause merging to fail even when all the quad fragments ideally would be merged. Because triangle 2 does not cover a multi-sample location the rasterizer does not emit a quad fragment. This prevents the quad fragments from triangles 1 and 3 from being merged because they do not share an edge. To remedy this problem, we modify the rasterizer to emit quad fragments with empty coverage masks whenever a triangle overlaps a 2x2 pixel region, but does not cover a multi-sample location. In this example, the empty quad fragment from triangle 2 merges into the buffer, allowing all quad fragments to merge. Quad fragments with empty coverage masks are never emitted by the merging unit, and thus never introduce extra shading.

The grid in Figure 8-right presents another scenario where all rasterized quad fragments should be merged but are not. In this case, the order in which the triangles are submitted to the rasterizer causes a gap in the topology masks. Because triangle 5 is not connected to triangles 1 through 4, a new quad fragment must be created even though the grid is fully connected. While it may be possible to optimize triangle orderings for merging, this case can be handled easily in the merging unit. When a quad fragment is chosen for eviction, the merging unit attempts to merge the eviction candidate. If the quad fragment cannot be merged the quad fragment is submitted for shading. With this modification, the quad fragment for triangles 1 through 4 merges with the quad fragment from triangles 5 through 8 instead of being evicted from the buffer.

4 Evaluation

We evaluate the performance and image quality of quad-fragment merging using three software rendering pipelines. The first pipeline, NOMERGE, mimics the behavior of a current GPU by shading quad fragments from each triangle independently. The second, MERGE, also shades quad fragments, but implements quad-fragment merging as described in Section 3. The third pipeline, VERTEX, is an implementation of REYES that shades grid vertices.

We use the DiagSplit algorithm [Fisher et al. 2009] to tessellate input surface patches into grids of triangles. DiagSplit integrates into our pipelines by augmenting Direct3D 11's tessellation stage [Microsoft 2010] with an additional stage for adaptive patch splitting. Tessellated grids contain at most 512 triangles, allowing the merge unit to encode adjacency using bitmasks as discussed in Section 3. Grids are occlusion culled immediately following tessellation. As a result, all pipelines rasterize exactly the same triangles. Unlike VERTEX, both fragment shading pipelines can also occlusion cull individual quad fragments prior to shading.

Using the three pipelines, we rendered the eight scenes shown in Figure 10. PLANE is a basic test of merging behavior. SINEWAVE's camera position is chosen to create many grazing triangles and is a quality test for MERGE. FROG (high-frequency displacement) and BIGGUY are standalone characters. ZINKIA and ARMY provide moderate depth-complexity scenes. Finally, PTCLOUD and FURBALL exhibit fine-scale geometry and complex occlusion. All scenes are rendered at 1728x1080 resolution.

4.1 Performance

4.1.1 Over-shade

Figure 9 plots the average number of fragments shaded by NOMERGE at each screen pixel and shows that over-shade in the NOMERGE pipeline is severe (pixels not covered by geometry do not factor into this average). For example, when BIGGUY is tessellated into 0.5-pixel-area triangle micropolygons and rendered at 16× multi-sampling, NOMERGE shades covered pixels nearly 14 times. Over-shade is notable even for small triangles covering a

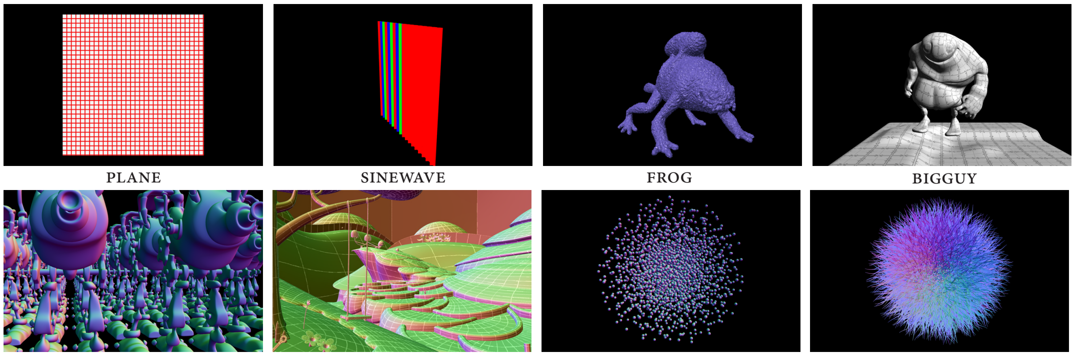

Figure 10: *Test scenes featuring high-frequency geometry (*SINEWAVE, FROG*), complex occlusion (*PTCLOUD, FURBALL*), grazing triangles (*SINEWAVE*), characters (*BIGGUY, ARMY*), and environments (*ZINKIA*).*

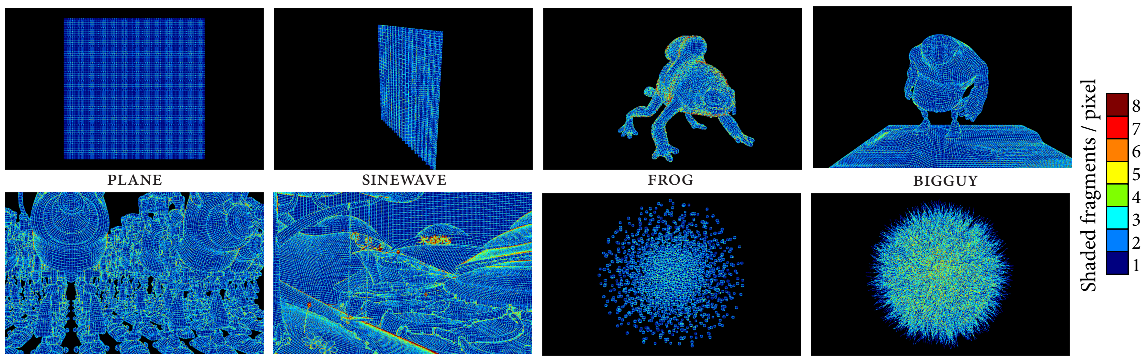

Figure 11: *Shaded fragments per pixel produced by* MERGE *(images colored according to the number of fragments shaded per pixel). Most scenes exhibit large regions of dark blue, indicating one shaded fragment per pixel.*

few screen pixels. Ten-pixel-area triangles still result in nearly four shaded fragments per pixel.

The MERGE pipeline (orange line) reduces the number of shaded fragments substantially. In many cases, all quad fragments at the same screen location are merged, so the amount of shading is independent of the size of scene triangles. Although our focus in this evaluation is on micropolygon-sized triangles, quad-fragment merging provides a significant reduction in shading work even when rendering small (but not necessarily sub-pixel) triangles.

On average, MERGE shades covered pixels approximately 1.8 times. This number falls short of the ideal one-fragment-per-pixel rate for three reasons: merging does not occur across grid boundaries, early occlusion culling in the pipeline is not perfect (regions of objects are shaded but ultimately occluded), and multiple fragments must be shaded in pixels containing object silhouettes. The images in Figure 11, which visualize the number of fragments shaded at each pixel (0.5-pixel-area triangles), show that MERGE indeed shades many image pixels exactly once. In these images dark blue pixels are shaded once, bright green pixels four times, and dark red pixels at least eight. A majority of pixels in these images are dark blue. Pixels near grid boundaries are shaded more than once because merging does not occur across grids. Shading also increases near object silhouettes because the screen-projection of grids becomes long and skinny (more pixels are near grid edges).

4.1.2 Comparison with NOMERGE

Figure 12 plots the benefit of quad-fragment merging as the size of the merge buffer is changed. Higher values on this graph indicate greater reduction in shading work (in comparison to NOMERGE). A 32-quad-fragment merge buffer reduces the number of shaded fragments by 8.1× (16× multi-sampling, 0.5-pixel-area triangles). A merge buffer of this size captures over 90% of the merges found by an "ideal" buffer of unbounded size. The benefits of merging decrease when multi-sampling is low. This result is not due to any change in the behavior of MERGE: the number of quad fragments generated by NOMERGE decreases at low multi-sampling because triangles are less likely to cover multi-sample locations.

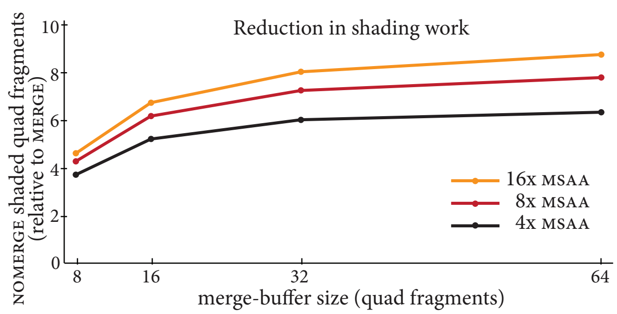

Figure 12: *Quad-fragment merging requires only a small amount of buffering to capture a high percentage of possible merges. A 32-quad-fragment buffer captures over 90% of the merges captured by a buffer of unbounded size.*

A 32-quad-fragment merge buffer constitutes only a small increase in the current storage requirements for a modern GPU shader core. High-end shader cores simultaneously shade more than 256 quad fragments and must already store shading inputs and shader intermediate values for these quad fragments. For the remainder of this evaluation, we configure MERGE to use a 32-quad-fragment merge buffer and render images using 16× multi-sampling.

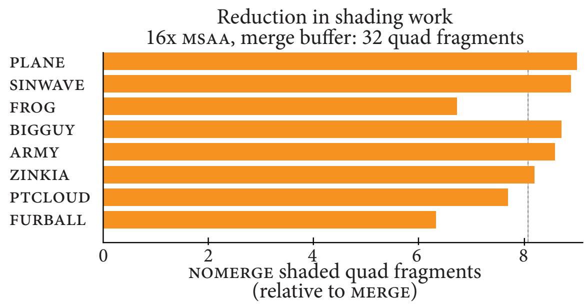

Figure 13: *On average,* MERGE *performs 8.1 times less shading work than* NOMERGE.

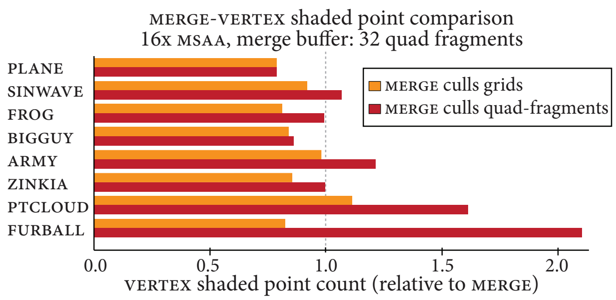

Figure 14: MERGE *shades approximately as many fragments as REYES (*VERTEX*) shades vertices. When fine-scale occlusion is present (*PTCLOUD, FURBALL*),* MERGE *shades over two times less than REYES because it culls quad fragments prior to shading.*

Figure 13 illustrates the benefit of MERGE on a scene-by-scene basis. The average reduction in shading across all scenes (8.1×) is shown as a vertical dotted line. The benefit of MERGE is the least for FROG and FURBALL because these scenes exhibit characteristics that limit opportunities for merging. FROG's high-frequency surface displacement causes triangles in the same grid to occlude each other, preventing merges due to rule 2. FURBALL's grids are long and skinny (hairs). Triangles on the borders of hairs have no neighbor to merge with.

4.1.3 Comparison With Vertex Shading

In Section 4.1.1, we measured that the MERGE pipeline significantly reduced shading work, but the absolute number of shaded fragments per pixel was greater than one (1.8). In a DiagSplit tessellation, grids with 0.5-pixel-area triangles correspond to a density of approximately one vertex per pixel.

We compare the number of fragments shaded by MERGE with the number of vertices shaded by VERTEX in Figure 14. Values greater than one indicate that MERGE shades fewer fragments than VERTEX shades vertices. On average, when both pipelines perform exactly the same occlusion culling operations (both only occlusion cull entire grids), MERGE shades 12% more than VERTEX (orange bars). This difference is explained in the top row of Figure 15 which shows a zoomed view of several grids from PLANE. Both VERTEX and MERGE over-shade pixels at grid boundaries. VERTEX over-shades because adjacent grids contain a vertex at these pixels. MERGE over-shades these pixels because quad fragments from different grids will not be merged. In MERGE the over shade occurs over a 2x2 pixel region, so over-shade occurs in more pixels than in VERTEX.

However, when quad fragments are occlusion culled by MERGE prior to shading (a common optimization in all modern GPUs), MERGE shades 17% less than VERTEX (red bars). The benefit of fine-granularity culling is particularly large in scenes, such as FURBALL and PTCLOUD, that exhibit fine-scale geometry. VERTEX performs more than two times as many shading computations as MERGE when rendering FURBALL. Culling shading work at grid granularity can be inefficient even when fine-scale geometry is not present, such as in the example from ARMY shown in the bottom left image of Figure 15.

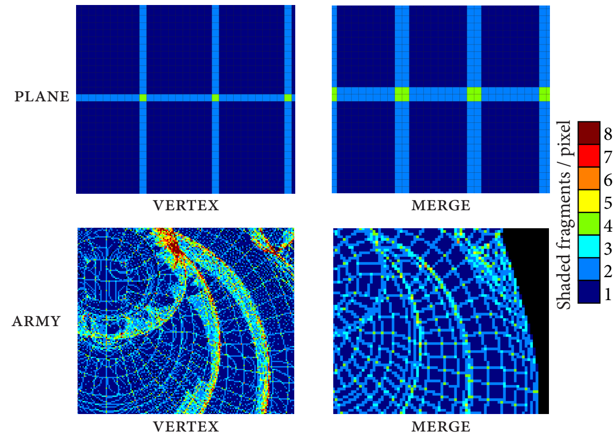

Figure 15: *Top: Quad-fragment shading yields more over-shade at grid boundaries than the REYES pipeline's vertex shading technique (*VERTEX*). Bottom: Quad-fragment occlusion culling eliminates parts of grids that are shaded by REYES.*

4.2 Quality

We rendered animations using all three pipelines and inspected the quality of the resulting images. Specifically, we looked for artifacts in the output of MERGE near silhouettes, as well as for texture level-of-detail errors that would result from inaccurate derivative calculations. Although the outputs of the three pipelines are different, the differences are subtle. For example, we observe that MERGE can produce less accurate shader derivatives near silhouettes of sharply curved surfaces. Still, in our tests, MERGE generates high-quality output that is comparable to the output of both NOMERGE and VERTEX. We refer the reader to the video accompanying this paper to compare the outputs generated by the three pipelines.

Our tests did show that MERGE can exacerbate artifacts caused by shading outside a triangle. Figure 16 highlights the contents of the multi-sample color buffer (top row) for one pixel of a rendering of FROG. The bottom row of the figure shows a portion of the final image surrounding this pixel. No triangle covers the pixel center, but the multi-sample location closest to the pixel center is covered by a nearly edge-on triangle. For this triangle, shading at the pixel center produces an inaccurate, bright white result. In NOMERGE, only the covered multi-sample is assigned this color, resulting in a subpixel error in the final image (bottom-left). By contrast, MERGE uses the erroneous shading result for all covered multi-samples in the pixel, producing a noticeable bright spot in the final image (bottom-center). Centroid sampling avoids these extrapolation errors, removing this artifact from both the multi-sample results (top-right) and the final image (bottom-right). Although centroid sampling is not commonly used in GPUs today, we find it to be a valuable technique for the MERGE pipeline. To ensure accurate shader derivatives when using centroid sampling, it is important to modify the finite difference calculations employed by current GPUs [Microsoft 2010] to account for the actual locations of shading sample points.

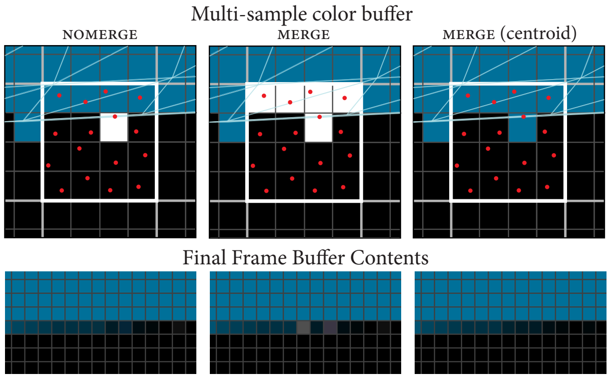

Figure 16: *Sampling shading outside a grazing triangle can produce artifacts in both* NOMERGE *(left column) and* MERGE *(center column). Shading artifacts from the grazing triangle are more noticeable in* MERGE *because they are applied to all multi-sample locations in the pixel. Centroid sampling corrects these artifacts (right column).*

5 Discussion

We have presented an evolution of GPU architecture that supports efficient shading of surfaces represented by micropolygons and small triangles. For micropolygons with an area of half a pixel, our approach reduces the number of fragments shaded by more than a factor of eight relative to a modern GPU. Often, a quad-fragment merging pipeline performs a similar amount of shading work as the REYES pipeline's vertex-shading technique. In cases of complex occlusion, it performs less than half the shading work of REYES.

The advent of efficient GPU tessellation makes the shading of small triangles (a few pixels in area) increasingly important, even if micropolygon-resolution surfaces are not required. Quad-fragment merging, unlike REYES, addresses this intermediate workload. The need to efficiently shade small triangles combined with the low cost and evolutionary nature of a quad-fragment merging approach should facilitate its incorporation into mainstream GPUs.

Quad-fragment merging requires a description of triangle connectivity. Our implementation obtains connectivity from the tessellation unit. Alternatively, connectivity can be provided through indexed-vertex-buffer formats, allowing triangle meshes to be rendered using quad-fragment merging without the need for pipeline tessellation.

This study of shading efficiency adds to our recent work on tessellation [Fisher et al. 2009] and rasterization [Fatahalian et al. 2009] that aims to design a GPU optimized for micropolygon rendering. Despite significant advances toward this goal, many interesting questions remain. For example, it is not obvious whether fragment or vertex-based shading techniques are preferred under different scene workloads or rendering conditions. Also, combining quad-fragment merging with recent attempts to incorporate motion and defocus blur in a modern GPU pipeline [Ragan-Kelley et al. 2010] should be immediately explored.

Acknowledgments

Support for this research was provided by the Stanford Pervasive Parallelism Laboratory funded by Oracle, NVIDIA, IBM, NEC, AMD, and Intel, the NSF Graduate Research Fellowship Program, and an Intel Larrabee Research Grant. We would also like to thank Edward Luong and Jonathan Ragan-Kelley for valuable conversations.

References

AKELEY, K. 1993. RealityEngine graphics. In *Proceedings of SIGGRAPH 93*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 109–116.

APODACA, A. A., AND GRITZ, L. 2000. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann.

BLYTHE, D. 2006. The Direct3D 10 system. *ACM Transactions on Graphics 25*, 3 (Aug), 724–734.

COOK, R., CARPENTER, L., AND CATMULL, E. 1987. The Reyes image rendering architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, ACM, vol. 27, 95–102.

DEERING, M., WINNER, S., SCHEDIWY, B., DUFFY, C., AND HUNT, N. 1988. The triangle processor and normal vector shader: a VLSI system for high performance graphics. In *Computer Graphics (Proceedings of SIGGRAPH 88)*, ACM, vol. 22, 21–30.

FATAHALIAN, K., LUONG, E., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. 2009. Data-parallel rasterization of micropolygons with defocus and motion blur. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, ACM, 59–68.

FISHER, M., FATAHALIAN, K., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. 2009. DiagSplit: parallel, crack-free, adaptive tessellation for micropolygon rendering. *ACM Transactions on Graphics 28*, 5, 1–10.

GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical z-buffer visibility. In *Proceedings of SIGGRAPH 93*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 231–238.

KESSENICH, J., 2009. The OpenGL Shading Language Specification, language version 1.5.

MICROSOFT, 2010. Windows DirectX graphics documentation. `http://msdn.microsoft.com/en-us/library/ee663301` [Online; accessed 27-April-2010].

MOLNAR, S., EYLES, J., AND POULTON, J. 1992. PixelFlow: high-speed rendering using image composition. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, ACM, vol. 26, 231–240.

PATNEY, A., AND OWENS, J. D. 2008. Real-time Reyes-style adaptive surface subdivision. *ACM Transactions on Graphics 27*, 5, 1–8.

RAGAN-KELLEY, J., LEHTINEN, J., CHEN, J., DOGGETT, M., AND DURAND, F. 2010. Decoupled sampling for real-time graphics pipelines. *MIT Computer Science and Artificial Intelligence Laboratory Technical Report Series, MIT-CSAIL-TR-2010-015*.

WEXLER, D., GRITZ, L., ENDERTON, E., AND RICE, J. 2005. GPU-accelerated high-quality hidden surface removal. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, ACM, 7–14.

ZHOU, K., HOU, Q., REN, Z., GONG, M., SUN, X., AND GUO, B. 2009. RenderAnts: interactive reyes rendering on gpus. *ACM Transactions on Graphics 28*, 5, 1–11.