Software Rasterization of Large Triangles

Project in Visual Computing 1

Github repo

Liopas Evangelos Ioannis – 12433743

1. Introduction

The approach of rendering primitives using a software-based approach in place of traditional hardware-based rasterization has existed and been a subject of research for quite some time. One reason would be that programmers require more flexibility in which parts of the graphics pipeline are actually programmable, however with the traditional pipeline, complex as it is, introducing more liberties becomes increasingly difficult, something that is not a problem for software-based approaches. In addition, there exist certain edge cases where software rasterization might be the more efficient choice for specific types of primitives rather than going the hardware-based way, as also seen in the intrinsics of Unreal Engine 5 in 2021 [1].

For Splatshop, the entirety of the primitives is software rasterized. A high-level description of how the rendering is done is as follows:

i. Store the entire primitives' list in the global memory of the GPU
ii. Process the triangles in batches (groups of 32 triangles each) and assign each batch for processing in individual thread blocks
iii. Each thread block processes its assigned batch, renders it in the frame buffer, and then is scheduled to be assigned more work until we run out of triangles.

This approach, though conceptually simple, is generally able to yield good performance, even for complex scenes with higher primitive counts. The major weakness of this method, however, is when the scene introduces a substantial number of larger primitives for rendering, by large typically meaning of triangles covering an area of more than 4000 fragments.

The reason for this, is how work is distributed among blocks, hence the streaming multiprocessors of the GPU. Since each block is assigned a batch of triangles to work solely on its own, with however many threads an individual block is comprised of, larger triangles would mean higher levels of work imbalance between the blocks. If work is not distributed as evenly as possible between the blocks, it means that some of them will become the bottleneck of the rasterizer, driving the framerates down because of certain blocks having to singlehandedly work on those larger triangles, while other blocks stand idle. As a matter of fact, for very large primitives, the task becomes practically sequential, as a limited number of threads tries to render a much larger number of fragments, losing all benefits of parallelization that we can leverage with modern GPUs.

The problem of efficiently rasterizing those larger triangles is precisely the goal of this project, and for that reason, code related with rendering smaller triangles is not tampered with at all, for our purposes.

2. The algorithm

Solving the problem of larger triangles means, as a result, to achieve better load balancing between the GPU's streaming multiprocessors when those larger primitives need to be handled. While we can keep processing smaller primitives in batches, where each block is solely responsible for rendering and related calculations, for the larger ones the approach changes and tries to split the rendering work between multiple blocks. The idea is visible below:

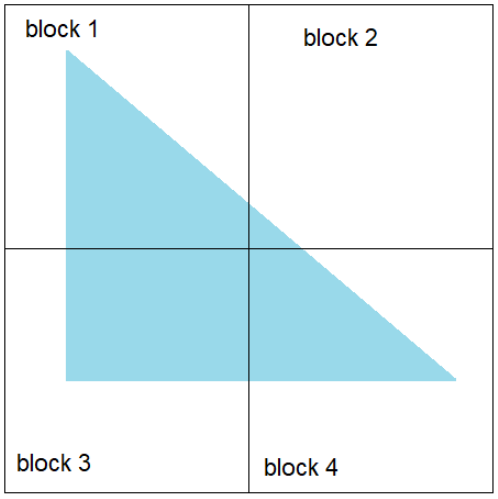

Figure 1. For larger triangles we employ "binning". We split screen space into equal size chunks called bins and determine the coverage of bins in screen space for each triangle. For each covered bin, a different block gets assigned work.

We now introduce "binning" specifically for large triangles [2]. We split the screen into bins of equal size, which specifically are 64x64 pixels in dimensions. Also, there is a lower level of areas which are called tiles and are 8x8 pixels wide, therefore each bin is comprised of 8x8 tiles. With the existence of bins and tiles we can construct a hierarchical tiled rasterizer [3] to split fragment shading work for bigger primitives more evenly across cores.

Bin Coverage – Coarse Test

Initially we perform a coarse test to estimate the area of the triangle based on the calculated bounded box, to verify if the triangle is big enough to go through the hierarchical pipeline with an area of 4000 fragments being set as the threshold. Then another very coarse and conservative test follows. We use again the bounding box of the triangle to estimate which bins of the screen might be at least partially covered, ending up with a list of bins that rendering might potentially happen. Of course, as figure 2 illustrates, such a coarse test will often produce bins with zero actual coverage against the triangle, but this is acceptable. These

bins will be discarded at a later stage, with the performance cost incurred by them considered minimal. This stage, we again feed triangles in batches to the thread blocks and we store bin lists globally to be fetched later for further processing.

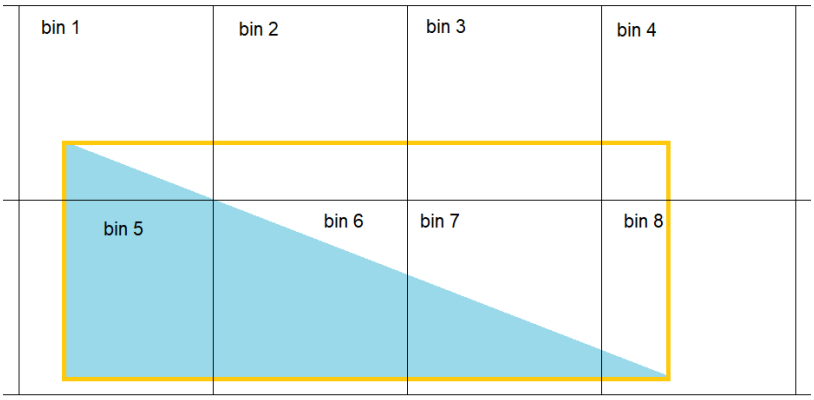

Figure 2. Conservatively estimate bin coverage against triangle using the triangle bounding box (orange). Notice that bins 3 and 4 will show as covered in that step and will be discarded in the following steps of the algorithm.

Tile Coverage – Finer Test

Once we have a bin list per primitive we move to the finer test where we process each bin individually. Each bin is comprised of 8x8 tiles, so we first construct a *tile coverage mask*. This is simply a bitmask of 64 bits where each bit is mapped to a specific tile. This allows us to cull out completely uncovered bins that passed the first test (if mask == 0) and save on calculation in the following stage of fragment shading by eliminating tiles that are found to be completely outside the triangle for partially covered bins [4].

To calculate the mask, we use the positive half-spaces of the triangle edges and take their union. This allows for quicker calculation of covered tiles, rather than testing each individually. We process each tile row within the bin, finding the span of the positive half spaces between the edges and doing so for each of the 8 tile rows will eventually produce a 64-bit tile mask. This test is also conservative as we extend the edges outward by half a tile width to avoid edge cases where tiles might not be included where they should, which resulted in striping.

The process is more easily understandable by looking at *Figure 3* below:

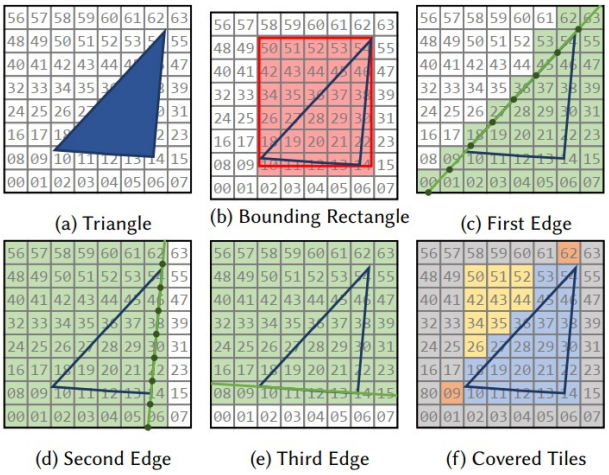

Figure 3. Tile coverage mask extraction for each bin by checking the positive half-spaces of the edges. As seen in (f) this allows to cull out tiles that show as covered from the bounding box while the test is conservative with the orange tiles also included [4].

Fragment Shading – Pixel-level Test

Once we have the tile coverage mask for the bins, we can move to the last stage of the algorithm which is fragment shading. Each tile showing as covered via the previous test is being handled by a single thread which performs a scanline-based check for each pixel. Since this is the most computationally intensive part of the process we try to limit the computations and memory accesses as much as possible and for that reason we avoid going through the entire scanline with the check being similar to what was shown for the tile coverage check (see Figure 3). The main difference is that the tests now are not conservative and are performed at pixel-level.

The implementation of the scanline rasterizer of the tiles is coarsely described by the following pseudocode:

```
for each tile_to_rasterize in parallel:
    tri = get_triangle_data(tile_to_rasterize)
    compute A, B, C = triangle_edge_coefficients(tri)
    precompute constants

    for y in tile_y_range:
        CY = B * y + C
        if row_outside_triangle(CY, A): continue

        compute lower, upper pixel x-bounds from CY and A
        clamp x-bounds within tile

        for x in x_range(lower, upper):
            bary = compute_barycentrics(x, y, A, B, tri)
            color = shade_pixel(triangle, bary)
            depth = interpolate_depth(tri, bary)
            pixel_value = pack_depth_color(depth, color)
            atomic_min(framebuffer[pixel_id(x, y)], pixel_value)
```

Figure 4. Scanline Rasterizer for tiles. For each scanline we find the range (xmin, xmax) and work on those fragments only.

Task Scheduling

The main idea for each of the algorithm's stages has been described above, however how these stages are scheduled for work in the GPU for more efficient utilization of its resources is a different problem. To get a faster rendering, two different ways of scheduling those tasks were tested:

- First, we tried a persistent kernel approach based on Kenzel et al. [4]. A persistent kernel (alternatively called "megakernel") runs for the entirety of the program and acts as a scheduler for each individual block. Each block can be assigned work in separation to the rest and can either handle rendering or bin processing much like these stages were described above, and they keep looking for work until the list of primitives to handle has been depleted.
  In addition, each block here is responsible for rendering specific areas of the screen (bins). If the hardware and the screen resolution allow for 1-to-1 correspondence between a block and a bin this can have the advantage of using a shared memory patch for local framebuffer writes limiting the global memory accesses since the global framebuffer will only need to be accessed once for each pixel at the very end of the process.
  So, the idea is: if a block has enough work on its bin queue, then it does fragment shading, otherwise it checks if there is more geometry to process and fetches a batch of primitives to cull out the large triangles and figure bin coverage, writing to the corresponding bin queues.
  Since each block is assigned specific bins for rasterization, this calls for block-specific bin queues in global memory where work is being put during the bin processing stage. This can be seen as a drawback because it necessitates multiple queues that demand atomic handling for their head and tail (one thread can read from its queue but all of them can dump bins to it), increasing memory consumption and algorithm complexity.

- Then, a two-stage approach was attempted. Here, the idea is simpler than the persistent kernel implementation. We have the triangle coverage – bin assignment stage and the fragment shading stage much like before however these stages are performed sequentially in two different kernels.

First, we process all the primitives, with each block taking in a batch, and for the larger triangles performing again the coarse bin coverage test. However, this time there exists just this one global bin queue which has been decided to be arbitrarily large so we can assume that an overflow is impossible.
Then, the second kernel performs the rasterization with each block taking in a batch of bins from the global queue to process at once. This is where the tile coverage masks are computed, proceeded by fragment shading as explained in the previous paragraphs. Blocks continue to fetch bins from the queue until they are depleted.

3. Benchmarks

After testing both techniques for a few use cases featuring large triangles we got the following results. By large triangles we refer to triangles in the 4,000 to 40,000 fragments in area range and anything above that is referred to as “very large”.
Below we specify some benchmark where we test for a certain number of these “large” or “very large” primitives. We specifically test for triangles with area/fragment count in the range of 4,000 to 40,000 which we defined as “large” above. We also test for the extreme case where the screen space is dominated by very large triangles. The exact dimensions of those triangles do not matter (eg if they are close to uniform or very sharp) as long as their fragment count is within those ranges. We test for a small number of large triangles (less than 10) a higher number (more than 30) of those, that puts more stress on the rasterizer and then for very large triangles which is a condition satisfied on extreme close-ups of the camera to the geometry.

Kernel	Large Triangles (<10)		Large Triangles (>30)		Very Large Triangles	
Persistent Kernel		0.991 ms		1.521 ms		2.126 ms
Store Bins (stage 1)	0.22 ms	0.966 ms total	0.22 ms	1.26 ms total	0.22 ms	1.577 ms total
Rasterize Bins (stage 2)	0.746 ms		1.041 ms		1.357 ms	

Benchmarks performed on a 1070Ti GPU. We observe generally better performance for the two-stage approach. Evidently, not assigning specific bins to specific blocks leads to better load distribution among the cores. This is easily visible for higher numbers of “large” triangles or in the “very large” triangles case where work is allocated more evenly with a global queue allowing for the blocks to pull work dynamically in a first-come, first-serve manner.

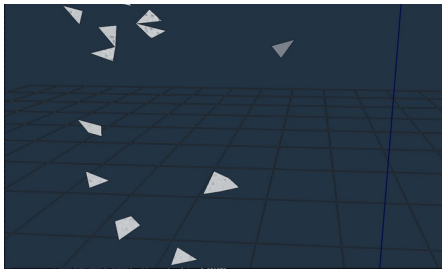

Figure 5. Use case of <10 large triangles

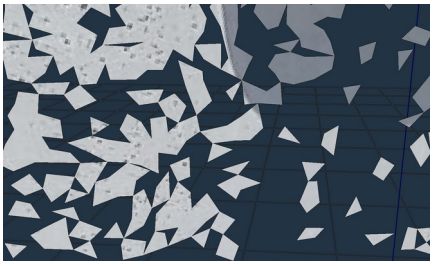

Figure 6. Use case of >30 large triangles

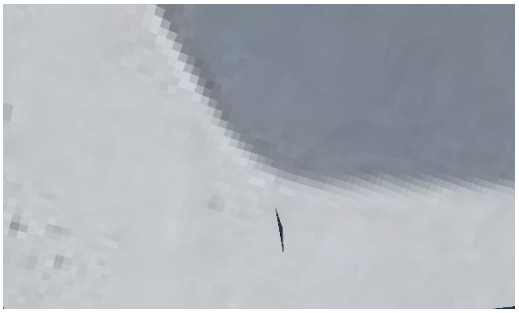

Figure 7. Use case with only very large triangles in extremely zoomed in view.

4. References

[1] A Deep Dive into Nanite Virtualized Geometry, https://youtu.be/eviSykqSUUw?si=p_Pnf_y5kRNfHtaR&t=2372

[2] Optimizing the basic rasterizer, https://fgiesen.wordpress.com/2013/02/10/optimizing-the-basic-rasterizer

[3] Laine, Samuli, and Tero Karras. "High-performance software rasterization on GPUs." proceedings of the acm siggraph symposium on high performance graphics. 2011.

[4] Kenzel, Michael, et al. "A high-performance software graphics pipeline architecture for the GPU." ACM Transactions on Graphics (TOG) 37.4 (2018): 1-15.