A Modern Approach to Software Rasterization

Zach Bethel
Taylor University
December 14, 2011

Abstract

This paper details the architecture of a multithreaded software rasterizer designed specifically for current and future generation multi-core processors. The pipeline rasterizes and shades four fragments in parallel using SSE instructions, and utilizes an extensive SIMD optimized math library for all transformations. By strategically utilizing the vector units widely available in modern desktop processors as well as multiple threads, performance is drastically higher than a fully serial implementation. Additionally, rendering order is preserved and a rich set of features is supported, including the ability to write custom vertex and fragment shaders in C++, set different render targets, and utilize z-buffering, backface culling and polygon clipping, as well as perspective correct texture mapping.

1 Introduction

The clear trend towards parallelism in the mainstream processor is once again bringing software rasterization to the forefront of active research. As fast as hardware pipelines are, they lack flexibility and stifle innovation. We are stuck with awkward, limiting high level shader languages to control our shading and strictly bound to triangle rasterization for the foreseeable future. As a result, most games look homogeneous and predictable. What if developers could control every aspect of the rendering pipeline? More interestingly, what other rendering algorithms such as Pixar’s REYES could be utilized? The future is bright for a software rendering revolution.

The obvious question, then, is whether the performance of a software-based renderer can compete with hardware versions. In this hardware generation, the answer to that question is a flat no. However, this paper details a threaded, SIMD optimized pipeline that is a step towards bridging this performance gap, providing a pipeline design that scales well to the number of physical cores on the CPU. Although the core pipeline functionality presented here is not innovative by any means, the primary goal was to maximize performance through parallelism—a factor that will play an increasing role in the years to come. Principles realized through this design would undoubtedly aid in the development of other software rendering pipelines.

2 Previous Work

Massively parallel software renderers designed for mainframe systems have existed for decades [7]. The pixel planes 5 system at the University of North Carolina, for instance, touted massive vector units dedicated to graphics processing [4]. On the other hand, at the time when mainstream software rasterizers were popular—before the advent of the GPU revolution—the typical home computer had only one processor. The Quake renderer by Id software is a prime example of a technically superior early rasterizer written completely in serial code. At some point, interest piqued towards integrating SSE technology into 3D graphics processing [6]. Later rasterizers, like Pixomatic 1.0 and 2.0 by Rad Game Tools, included limited data-level parallelism via SSE/MMX instructions.
More modern rasterizers employ both thread and data-level parallelism for geometry and fragment processing. Microsoft’s WARP architecture supports the full DirectX 10 feature set implemented with SSE instructions and scales very well to multiple threads. It is even able to outperform low-end integrated graphics cards in some cases. As far as DirectX 9.0 renderers go, SwiftShader by TransGaming Inc. claims that their solution can run Crysis at playable fram rates on the latest processors. Pixomatic 3.0 boasts full multi-threading support as well. Unfortunately, since these products are all proprietary, the details of their architectures are not released to the general public.

Interestingly, recent work has also been done to create a software pipeline which runs on the GPU using the CUDA software platform [5]. Like any software solution, this allows additional flexibility at the cost of speed. The architecture uses a similar binning algorithm to the implementation presented in this paper, however, their rasterization process is instead divided into two distinct coarse and fine stages.

Intel recently pushed the boundary of software rendering with their attempt at the Larrabee project, which boasted 16-wide SIMD vector units and many x86 cores in the hardware specification. Particularly intriguing was the fact that their rasterizer was expected to run completely in software, executing across many threads. The pipeline used a tile-based, heierarchical rasterizer which heavily utilized the vector units [8, 3]. Ultimately, the project did not meet expectations and was shut down in May of 2010; however, Intel recently announced a similar project called Knights Corner, a massively parallel chip that will supposedly exceed teraflop speeds.

3 Pipeline Overview

Implementing a single-threaded software rasterizer is straightforward. A DirectX 9.0 style pipeline might consist of this series of sequential rendering stages: input assembly, vertex shading, culling/clipping of primitives, rasterization, and fragment shading. See Figure 3.1. On the other hand, adding multithreading and SIMD processing complicates the situation significantly. For one, we no longer have exclusive access to resources like the frame buffer; threads will need to share memory correctly while minimizing contention. Also, load balancing must be taken into account. For example, some areas of the scene may be more tightly packed than others. Threads need a way to handle unbalanced workloads effectively. Utilizing SIMD instructions requires reworking our data structures to use contiguous chunks of 16-byte aligned data, which effectively eliminates the traditional scanline rasterization algorithm. Finally, rendering order should be preserved in order to protect the integrity of the rendered scene.

The challenges of multithreading are addressed here by employing a sort-middle architecture [3, 5, 7]. Prior to rendering, the frame buffer is divided into a grid of tiles. Incoming primitives pass through the

Figure 3.1: A high level diagram of the rendering pipeline. The vertex and index buffer are passed to the input assembler, which composes the data into primitives and passes them to the vertex shader stage. Next, primitives that face away from the camera or exist outside the view frustum are culled, while primitives which intersect it are clipped. Those elements that pass this stage are rasterized, shaded, and drawn to the frame buffer.
Figure 3.2: A high level diagram of the software rasterization pipeline presented here. Threads receive batches of indices for vertex processing and binning. Once all the primitives are binned, threads switch over to tiles for rasterization and fragment processing work. Threads pull from the tile bins and rasterize primitives, emitting coverage masks to a buffer. When all threads are finished rasterizing, the work queue of tiles is traversed again, where fragments are shaded and written to the frame buffer.

vertex shading and clipping stages like normal, but then undergo a new process called binning. In this stage, transformed primitives are tested for intersection with each frame buffer tile in screen space; any primitive which passes the intersection test is added to a bin for that tile—where a bin is just a simple queue of indices. Once all the primitives are binned, the threads are assigned tiles for rasterization and fragment processing. This assignment is managed by a job queue holding only those tiles which contain triangles. The process is then repeated on the next stream of input. This design creates a natural synchronization point located in the center of the data-path where the threads transition from primitive to tile processing—hence the term sort-middle (see figure 3.2).

The first half of the datapath handles vertex and primitive-level processing. As input, each draw call to the rasterizer receives an array of vertices and a companion array of indices (our implementation only supports triangles, but one could certainly extend it to support other primitive types as well). In the main thread, if the input size exceeds a certain threshold, it is split into batches. The benefits to this are twofold: not only is there an upper bound on the size of internal buffers—resulting in a simpler implementation and less of a memory footprint, but we can optimize them for cache coherency. Thus, the pipeline is effectively run multiple times in sequence—once for each batch. Unlike in hardware where multiple stages of the datapath might be active at once—each working on a separate batch, all work threads instead focus on the current stage for the current batch. This is done to best utilize caches.

Once the batch of triangles is prepared, we immediately split it again into smaller, equal-sized batches—one for each thread. At this point, everything is ready; the work threads are signalled to begin processing. The datapath for primitives consists of vertex shading, backface culling, clipping, and finally binning. Additionally, threads store intermediate results in local buffers; the size requirements of which are related to the total batch size.

During the binning stage, a work queue of active
tiles is compiled; following the global synchronization, threads transition to processing jobs from this queue. The rasterizer emits coverage masks for each primitive in the bin of a given tile, which accumulate in a buffer. A coverage mask defines a set of screen space coordinates as well as an optional bit mask describing the state of each pixel; also, a flag is included to denote whether the element is a single quad of fragments, a block, or an entire tile. It is tempting to forgo this step and instead group the rasterizer and fragment shading into the same stage since they are related; however, locality in the cache is better utilized when threads can process fragment shaders exclusively without thrashing the cache with rasterization specific data.

Upon completion of this stage, the threads traverse the tiles a second time and use the previously computed coverage masks to shade four fragments at a time using SSE instructions. The pipeline then restarts the process at stage one with the next batch of primitives. The final result is a rendered scene. The next few sections will explore these pipeline stages in greater detail.

### 3.1 Vertex Processing

In the vertex processing stage, threads are each given local instances of working buffers, both to avoid contention and preserve order within the data. Among these buffers are a small post-transformation vertex cache, an intermediate vertex buffer for transformed primitives, and a face buffer containing cached interpolation data for the fragment shading stage. The small vertex cache is direct mapped and uses the vertex index as the cache index. Before a vertex is processed by the vertex shader, the cache is first checked for a previous instance of the vertex. In a typical grid of triangles, a vertex is shared by six other triangles on average. Thus, the cache takes advantage of temporal locality to avoid reprocessing the same vertex multiple times.

When processing triangles, three vertices are either passed through the vertex shader or pulled from the cache. The face is then culled or clipped against the view frustum. Triangles passing this test are appended to the intermediate vertex buffer. This buffer is unique in that it defines both the vertices and face information for the primitives; that is, there is no index buffer and thus vertices may be duplicated. On the bright side, only a handful of SSE load/store instructions are required to copy an entire vertex. This optimization is used wherever possible to increase bandwidth. Once again, finding the ideal vertex buffer size is highly dependent on the total batch size.

As an aside, there are special requirements for polygon clipping; the algorithm splits triangles against the six frustum planes into a maximum of five sub-triangles. Thus, to handle the maximum case, we multiply our total buffer size by a factor of 5 before allocating. Although typical workloads rarely get close to this upper bound, it is easy to see the benefit of knowing the maximum input size. Internal memory requirements could quickly grow out of hand otherwise.

Finally, when vertices are appended to the vertex buffer, face data is precalculated and cached in a face structure which links to these vertices. An additional face buffer holds these elements which are used later in the rasterization and fragment shading stages. Interpolation deltas are computed as a gradient vector for each varying attribute. This is easily done using the plane equation [7].

Load balancing concerns in this first half of the datapath are not as critical. For one, most of the workload in the vertex processing stage is thread independent and already quite balanced. Secondly, a majority of the heavy lifting occurs during the latter half with fragment processing. Thus, in this situation, distributing equal-sized batches to each thread still results in near-linear scaling.

In addition to achieving parallelism via threading, the vertex processing stage also utilizes SSE instructions wherever possible. This presents a particular challenge in that the typical interleaved vertex format is not organized in a SIMD-friendly fashion. As an example, we cannot easily represent the individual components of four vertices with four SSE registers; The data would need to be shuffled multiple times, and then shuffled back afterwards [6]. Figure 3.3 demonstrates this operation. The problem is made worse if we imagine using AVX registers which are eight el-
The interleaved nature of the vertex format makes a complete SIMD solution difficult for vertex shaders. A significant amount of time is spent shuffling the data for all four vertices into individual registers. As a result, a simpler approach is taken which optimizes general math within the vertex shader using SSE. Future versions of AVX supporting scatter/gather will change this.

Elements wide (and soon to be larger). One obvious solution to this is to store vertices in an SoA (structure of arrays) format, however doing so removes the spatial locality benefits of an interleaved vertex format. It also adds a level of awkwardness to the data structures.

Alternatively, most vertex processing requires matrices and four element vectors. Thus, in this implementation, the vertex shaders utilize a custom built SSE optimized math library for all operations. The classes in this library encapsulate SSE intrinsics through the use of operator overloading. In the future when loading/storing from arbitrary memory locations is supported by AVX and larger registers are standard, the former option will be the desired approach.

### 3.2 Binning

After a triangle passes the culling/clipping stage and is added to the intermediate vertex buffer, it undergoes the binning process. First, the bounding box of the triangle is computed and clamped to the frame buffer edges. Then, we loop through the tiles within the bounding box and perform half-space tests for all three edges the triangle against each tile [3, 7]. This allows us to test for three possible situations: either the tile is trivially rejected—in which case we ignore it and move on, it is trivially accepted—a special case that allows us to emit a single coverage mask for entire tile as an optimization, or it intersects one or more of the edges. Only in the last case is the polygon binned for rasterization. This hierarchy speeds up cases where we are dealing with large polygons.

When a bin receives its first triangle, that tile is added to the work queue. This optimization ensures that the threads spend less time fighting for empty tiles in later stages. A simple atomic increment is used to flag when a tile is no longer empty. Also, at this point in the pipeline, bins are still kept local to each thread. Not only does this eliminate contention, but it preserves rendering order. Each thread is assigned a unique index which it applies to a global array of bins to get its local bin. Once the threads are finished binning, all it takes to preserve rendering order is to iterate through the bins for each tile in sequence.

### 3.3 Rasterization

At this point in the dataflow, threads are synchronized and all triangles in the current batch are binned to their respective tiles. Furthermore, a queue of tiles is now compiled and ready for processing. Using an atomic increment on the queue head, threads grab the next available tile. Each thread then has exclusive access to that tile, allowing the rasterization stage to run independently on each thread.

To begin, the algorithm iterates sequentially through the bins, preserving sorting order. Triangles are rasterized using the same half-space equation tests as in the binning stage. Compared with the traditional scanline algorithm, this method is far more parallel friendly, allowing entire blocks of pixels to be tested at once. Furthermore, it is elegant and easy to implement [3].

Just like in the binning stage, we find the bounding box of the triangle—clamped instead to the tile edges. Next, we loop through the area contained by the bounding box and test 8 x 8 pixel blocks using the half-space edge functions. Once again, this allows us to trivially accept or reject 64 pixels at once. If
a block is accepted, we simply emit a full mask for the entire block and move on. On the other hand, if a block intersects an edge of the triangle, we have to traverse into that block and test at the pixel level. Figure 3.4 demonstrates this hierarchy.

The beauty of this algorithm is apparent when we realize that four pixels can be tested simultaneously using SSE. Rather than keep scalar edge function gradients for individual pixels, we compute a vector of them. Evaluating the function for those four pixels and using SSE comparisons to compute a 4 bit mask grants us full data-level parallelism per thread.

The hierarchy evident in the rasterizer helps handle a diverse range of polygon sizes; it skips unnecessary work and avoids generating extraneous coverage masks. Without this hierarchy, a single triangle filling the screen at a 1920 x 1080 pixel resolution would generate 32,400 64 bit masks. Using 64 x 64 sized tiles, on the other hand, generates just over 500. This is one example of how software rendering allows us to optimize for the common as well as uncommon case [3].

Even with this optimization, there is a wide variance in the amount of possible coverage masks output by the rasterizer. In the worst case, we might have every triangle emitting the maximum number of block masks possible. This presents a challenge in allocating enough buffer space for an application. The problem is similar to how the operating system handles page tables: there is an upper bound in the memory requirement that we need to account for, but allocating for the worst case results in wasted space.

As a compromise, the mask buffer is composed of buffer segments; when the current segment is full, a new segment is allocated, allowing the buffer to grow as the application requires. The buffer threshold is checked and potentially resized after each triangle is rasterized. Thus, at a minimum, each segment must be able to hold the maximum number of coverage masks emittable by a triangle for any given tile (a number which relates to the tile size).

### 3.4 Fragment Processing

Threads having reached the end of the work queue loop around and traverse it once more—this time using the coverage mask buffer as input. Each thread iterates through the coverage masks in order. The loop is broken up to handle three distinct mask elements: entire tiles, entire blocks, and fragment quads with an accompanying mask. Entire blocks are handled essentially with a double for loop around the shaders and attribute interpolation. To optimize for locality within the cache, the tile codepath traverses 8 x 8 blocks of pixels using this same method. If a quad mask is encountered, the loop uses the 4 bit mask to determine individual fragment visibility.

Just as before, entire quads of fragments are processed in parallel. This includes both the shaders and attribute interpolation. Varying attributes are computed for the first row of fragment quads in the block using the gradients previously computed in the vertex processing stage. Since there are 8 pixels to a row, this results in two SSE registers per attribute; subsequent rows are computed by incrementing each attribute by the y coordinate of the gradient [7]. The number of active vertex attributes can be dynamic, based on the shader; thus, a for loop is required to traverse them properly. To reduce branching, the algorithm is structured to require only two of these loops per row: once to compute the final division by \( w \) (required for perspective correction) preceding the
shader call, and once to increment by the gradient in preparation for the next row. The column loop of the block is completely unrolled.

If a quad is encountered, the depth test and shaders are computed for all four fragments—even if only one is determined visible by the rasterizer. Invisible fragments are masked out by either the coverage mask or the depth test before the final write to the depth and frame buffers. As an optimization, the rasterizer will never output a coverage mask for a completely invisible quad. To assist with shading, another custom math library provides objects mimicking scalar floating-point numbers as well as 2 through 4 element floating-point vectors. In the background, each scalar object is really an SSE register holding values for four fragments; a quick look at the shader disassembly will reveal SSE instructions. This abstraction allows users the flexibility to write shaders without resorting to assembly language or compiler intrinsics, if they so desire. Finally, helper functions are included for common tasks like sampling from a texture map. Once all the fragments have been processed, the main thread returns to the batch processing loop and repeats for any additional batches. Threads synchronize and begin the process all over again for the next batch.

4 Performance Considerations

Performance in the pipeline depends greatly on how we handle intermediate data. For instance, tile resolution and batch size requires some fine tuning in order for the system to reach optimal speeds. As tiles get smaller, fragment workloads become more balanced; areas with more triangles touch more tiles, reducing cases where a single tile might hold a majority of the triangles for the batch. However, as we decrease the workload within each tile, thread contention and preparation overhead grows. Specifically, the binning and rasterization stages must traverse and process more tiles. Experimentation suggested either 64 x 64 or 32 x 32 pixels to be the optimal tile size, depending on the dimensions of the viewport (sizes ranging from 800 x 600 to 1920 x 1080 were tested).

The largest factor in maximizing parallelism is the batch size. Finding the optimal value here is straightforward: we want the biggest buffer possible that will still fit in the level 2 cache. If the buffer is too small, threads will spend more time preparing work and less time processing it. This will drastically reduce overall parallelism. On the other hand, a buffer larger than our cache brings no noticeable performance improvement. Although we may have more parallel work per batch, the penalty of fetching data not local to the processor core negates this benefit—at the additional cost of unnecessarily large internal buffers. Figure 4.1 demonstrates the effect of batch size on performance. For example, when testing on an Intel Core i7-2720 quad-core, hyperthreaded processor with 256 KB level 2 caches, the ideal batch size was around 15,000 total vertices—an amount just larger than the cache. Keeping as much of our working data in the cache as possible is critical to reaching peak performance.

Furthermore, exploiting temporal locality is a key optimization. It is this reason that rasterization and fragment processing are disjoint stages. The level 1 cache of each processor is a mere 32 KB in size; thus, it is important that we are careful to reuse data as much as possible to reduce fetch latency. Particularly, rapid switching between rasterization and fragment shading effectively thrashes the L1. A better approach is to localize those operations to their own stages, allowing working variables like interpolants, edge functions, and color buffer lines to stay in fast memory as long as possible.

Finally, the fragment shader loop is extremely tight, making the cost of a branch misprediction particularly expensive. To help remedy this, all nonessential conditionals are removed from the inner loop. Instead, function templates are used to cache all permutations of branching codepaths at compile time. This not an ideal solution, but it works for a simple rasterizer. A better solution would be to use a JIT compiler or assembler to dynamically create the code for fragment processing based on the current state [1, 2]. One specific limitation of templates is that it is difficult to handle integer states—like the number of active varying attributes. Thus, for loops are used only where absolutely necessary, as
described in the previous section.

Total parallelism is extremely difficult to achieve in modern graphics pipelines. Inevitably, the work threads do end up idling. For instance, we have an unavoidable synchronization point after the binning stage. Threads cannot begin processing tiles until all primitives are processed; there is no way around this. Even though workloads are fairly balanced, they are not completely. In this implementation, dynamic load balancing techniques were considered, but ultimately a uniform workload was used for its simplicity and ease of implementation. The situation is complicated by the requirement of preserving rendering order.

Also, with fragment processing, there are common cases when a quad of fragments contains invisible pixels. In the worst case, only one pixel is rendered, while the other three are masked out. This is a common issue found also in hardware pipelines, where groups of lockstepped threads are forced to process an entire batch of fragments in parallel even if only a single fragment is visible. That is a situation that is extremely difficult if not impossible to avoid, especially when using vector units.

Finally, the total parallelism of the application is heavily dependent on the amount of serial code performed between renderings, as determined by Amdahl’s Law. The test application used in the results section spent very little time in between draw calls. A typical game, on the other hand, has other subsystems demanding attention each frame. The best approach to solving this with the current design is to offload the rendering to another thread and utilize a command buffer to process work. This would allow the rendering thread to farm out data to the work threads without ceasing, maximizing total parallelism each frame.

5 Results

For a simple scene consisting of roughly 66,000 total rendered triangles, complete with per pixel lighting and shadow mapping, performance scaled very well to the number of hardware threads on the CPU. In particular, 8 threads increases performance on an Intel quad-core, hyperthreaded processor to 4 times the speed of a single thread. With respect to the amount of serial work in the demo, this figure exceeds the theoretical maximum. Hyperthreading is responsible for that. Activating only 4 threads yields a speedup up 3.3. Figure 5.2 shows the performance based on the number of work threads. Once the number of logical threads exceeds the number of available hardware threads, performance slowly declines. The cost of context switching is clearly to blame for this behavior.

Implementing SSE into the pipeline more than doubles the performance of each individual thread. This means that a fully serial implementation is more than 8 times slower on a quad core processor than our pipeline. As an interesting side note, profiling data determined that vertex processing was roughly 25% of the total workload, while rasterization and fragment shading dominated the remaining 75%. This is a key reason why those stages remained the primary focus of SIMD and load balancing optimizations. The fragment processing loop—not including the shader—required a whopping 34% of the total processor time; with the the color/depth buffer fetch/store, early-z rejection test, and attribute interpolation loops attributing most to the workload. Memory bandwidth
Figure 5.1: The demo consists of a 33,000 triangle statue model, rendered in two passes to create the shadow mapping effect. A 1024 x 1024 pixel render target was used for the shadow buffer. A single, directional light shows the full per-pixel lighting complete with specular reflection. The primary render pass required two texture fetches per pixel. Below, the tile floor demonstrates the perspective correct texture mapping. The primary screen resolution used for testing was 1920 x 1080 pixels.

is definitely the limiting factor of the modern CPU for software rendering. Hyperthreading allows more hardware threads in flight at once which helps hide this latency, but it is still quite evident.

6 Future Work

Although the architecture behind the rasterizer is sound, improvements definitely could be made. A big limitation of the current design is that it requires the user to implement a command buffer themselves in cases where the application needs to do more than just submit draw calls. A better method might be to allow the main thread to submit work to the background threads asynchronously, without participating in the rendering process. This technique is precisely how hardware accelerated SDKs like OpenGL submit work to the GPU.

Although this implementation uses simple, even-sized batches for all vertex processing, a more rigorous load balancing scheme might improve parallelism even further by reducing the synchronization penalty at the pipeline midpoint. As the number of processor cores increases into the double digits, the design might scale better by activating multiple pipeline stages rather than just one—allowing threads to process back-to-back batches in parallel.

Finally, with the addition of AVX registers which are expected to soon reach 1024 bits wide, the role of data-level parallelism in software rendering is more prominent than ever. Threads will need to process bigger blocks of fragments at once to keep up. Unfortunately, unless extensions are added allowing vectors to reference arbitrary memory locations, the chances of achieving anything close to a true linear speedup with current designs are not good. Vertex shading in particular would become more awkward to implement—forcing us to jump to a SoA format after all. Time will certainly tell.

7 Conclusion

In the age of parallelism, software rasterization is once again reaching a point of commercial feasibility. The idea of writing a high-performance custom graphics pipeline running in software is an appealing concept that would gain the mainstream graphics market much needed flexibility. Shaders could be written in
straight C++, and even have access to neighboring elements and general memory. In some cases, techniques that are not even possible on the current generation GPU might become trivial. Additionally, this step could help bridge the gap between the movie and gaming industry technologies. Clearly, the future is still bright for software rendering.

In the context of real-time software rasterization, only recently have fully parallel implementations surfaced within commercial products like WARP and SwiftShader[2, 1]. Even then, due to their proprietary nature, internal architectural details are not revealed to the general public. Although the architecture of the pipeline detailed in this paper is not novel per se, it is intended as an open contribution towards achieving true high-performance of software rasterization on current and next-generation hardware.

References


