

A⁴: Asynchronous Adaptive Anti-Aliasing using Shared Memory

Rasmus Barringer
Lund University

Tomas Akenine-Möller
Lund University and Intel Corporation

Abstract

Edge aliasing continues to be one of the most prominent problems in real-time graphics, e.g., in games. We present a novel algorithm that uses shared memory between the GPU and the CPU so that these two units can work in concert to solve the edge aliasing problem rapidly. Our system renders the scene as usual on the GPU with one sample per pixel. At the same time, our novel edge aliasing algorithm is executed asynchronously on the CPU. First, a sparse set of *important* pixels is created. This set may include pixels with geometric silhouette edges, discontinuities in the frame buffer, and pixels/polygons under user-guided artistic control. After that, the CPU runs our sparse rasterizer and fragment shader, which is parallel and SIMD:ified, and directly accesses shared resources (e.g., render targets created by the GPU). Our system can render a scene with shadow mapping with adaptive anti-aliasing with 16 samples per *important* pixel faster than the GPU with 8 samples per pixel using multi-sampling anti-aliasing. Since our system consists of an extensive code base, it will be released to the public for exploration and usage.

CR Categories: I.3.3 [Picture/Image Generation]: Antialiasing; I.3.7 [Three-Dimensional Graphics and Realism]: Color, shading, shadowing, and texture;

Keywords: visibility, anti-aliasing, shading, rasterization

Links:  DL  PDF

1 Introduction

Geometric aliasing is still one of the major challenges in real-time rendering, as noted by Andersson [2012] among others. Supersampling anti-aliasing (SSAA) is expensive both in terms of memory bandwidth usage, and in terms of fragment shading since each visibility sample is shaded individually. Multi-sampling anti-aliasing (MSAA) is less expensive, since the fragment shader is only executed once per pixel per primitive even though there are more visibility samples. Still, this incurs a lot of overhead in terms of rasterization, color & depth buffer memory bandwidth, and shading usually increases along triangle edges (see Section 2 for more information about this). At the same time, we note that many desktops and laptops have four or more CPU cores, and often, only a fraction of them are active during game play. A major goal of our research has been to develop an adaptive anti-aliasing (AA) algorithm where the CPU cores and GPU cores join forces to solve this problem using a shared memory architecture.

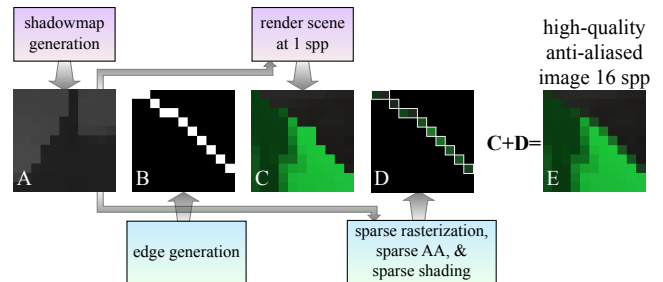


Figure 1: Our system generates high-quality edge anti-aliasing. Two render targets are rendered (top) with one sample per pixel (spp)—a shadow map and a rendering of the scene using the shadow map. In addition, our algorithm generate edges (bottom) and perform sparse rasterization, anti-aliasing (AA), and shading (bottom) for the edge pixels in the scene (but not in the shadow map). What is unique about our system is that all the squares in the middle are located in shared memory, the passes at the top are executed by the GPU, and the passes at the bottom are executed on the CPU, and together they generate a high-quality anti-aliased image (right) with 16 spp for the edges.

Already in 1977, Crow suggested to apply more expensive anti-aliasing techniques *only* to pixels being covered by the geometrical edges [Crow 1977b]. Algorithmic variants based on the same underlying idea have been proposed [Sander et al. 2001; Aila et al. 2003] after that, but there is still no practically useful algorithm that also generates high image quality with high performance. Based on Crow’s observation that only a sparse set of pixels needs high-quality edge anti-aliasing, we present a novel algorithm for solving the geometrical edge anti-aliasing problem.

Our algorithm leverages idle CPU cores to perform anti-aliasing for a sparse set of pixels, while allowing the GPU to render the entire scene quickly using a single sample per pixel (spp). There are several merits to this approach. First, since accurate anti-aliasing is calculated for limited parts of the frame buffer, the workload is control flow divergent. A current CPU is thus a better fit than current GPUs to perform such calculations. Second, since anti-aliasing is decoupled from the GPU rendering pipeline, we can anti-alias only the most important pixels, which is often less than 5% in our experience, and in the worst case early-out in order to guarantee a certain frame rate. Third, our target architecture exploits shared memory between the CPU and GPU, which makes it possible for the sparse fragment shader evaluation done on the CPU to directly (without copy) access render targets already generated by the GPU, and also sparsely update the final image with high-quality anti-aliased pixels. Together this makes our algorithm extremely fast. A high-level illustration of our algorithm can be seen in Figure 1.

2 Previous Work

There is a wealth of literature on the topic of post-process screen-space anti-aliasing. The first technique in this area is called morphological anti-aliasing (MLAA) [Reshetov 2009], where the idea was to analyze the rendered image, detect edges, and cleverly filter over them with the goal of approximating an anti-aliased image. Since then, many intelligent edge-blur techniques have been proposed, and they are widely used in the game industry, since they are

fast and power-efficient. We refer the interested reader to the excellent SIGGRAPH course on this topic by Jimenez et al. [2011]. We note that these techniques are not robust to all kinds of scenarios.

Point sampling techniques, such as SSAA and MSAA (mentioned in Section 1), are commodity features in graphics architectures today. One potential disadvantage of MSAA is that shading along edges increases. For example, if a pixel is intersected by an edge (does not need to be a silhouette) shared by two triangles, and if there is at least one sample inside each triangle, then shading will be executed twice, for such pixels, and it will be done on a per 2×2 pixel basis. This can be particularly expensive for highly tessellated geometrical objects. Fatahalian et al. [2010] present a hardware-based approach to solve this, where fragments are gathered and merged in order to avoid unnecessary shading. In coverage-sampled AA (CSAA) [Young 2007], each pixel stores at most 2^n colors in a per-pixel palette, and each sample may point into this palette using an n -bit index. For high-quality visibility, the number of coverage samples per pixel is higher than the number of colors in the palette, e.g., a pixel may have 4 colors and 16 coverage samples. When more than 4 colors appear in a pixel, a heuristic is needed to reduce that set of colors down to 4 colors again. Note that these techniques, including SSAA, MSAA, and CSAA, are still rather brute-force approaches, since anti-aliasing is not directed only towards the pixels that are in need of it.

Carpenter’s A-buffer [1984] is another type of MSAA, where shading is decoupled from color and depth samples. Each rendered polygon stores a fragment per pixel, where a fragment consists of a coverage mask, i.e., a bitmask, as well as a color and some depth representation. A pixel can store any number of fragments in order to capture transparency effects etc. In addition, merging of fragments can be done in some situations for more efficient processing.

Crow [1977b] suggested that more effort should be spent on edges, and in particular on geometrical edges. For such pixels, analytical methods would be used to compute the area of coverage, and accumulated to the pixel color. However, this assumed non-overlapping polygons, and hence no handling of depth, which makes its usage less influential. Based on Crow’s observation, however, Sander et al. [2001] rendered the scene using 1 spp, and then overdrew the discontinuity edges with anti-aliased lines. This requires the lines to be rendered in back-to-front order for correct blending, and depth may not be resolved correctly on these lines, since the depth buffer is not multi-sampled. Aila et al. presented a hardware mechanism called the delay stream [2003]. One of their applications was adaptive AA on geometrical edges, also based on Crow’s observation. However, the delay stream is not yet implemented in any hardware to our knowledge. Our approach is also based on spending more AA efforts on geometrical edges, but our approach is based on introducing a sparse rasterizer & shader, and our method exploits shared memory to efficiently implement complex shading.

Greene et al. presented hierarchical Z-buffering [1993], where a quad tree of depths is maintained, and each node stores the maximum depths of its childrens’ depths. An object-space octree of the scene geometry is used for hierarchical culling against the Z-pyramid. Greene and Kass [1994] extended the previous method to render scenes with guaranteed error bounds. An octree of the geometry is traversed in front to back order, and occluded geometry is culled against the quadtree. Potentially visible polygons are inserted into the quadtree that contains conservative Z_{min} and Z_{max} of the geometry in the node. Finally, the hierarchical tiling algorithm for high-quality rasterization by Greene [1996] uses a coverage hierarchy. Classification of the polygon during traversal of the hierarchy is divided into inside, outside, or intersecting the nodes. Similarly, a quadtree node may be marked as fully covered, vacant, or active. Polygons are traversed in a strict front-to-back order us-

ing a BSP-tree with a Warnock-style [1969] tiler for rasterization.

3 Algorithm Overview

Our algorithm is divided into two parts, where one is executed on the graphics processing unit (GPU), and the other on the CPU cores. The idea is to take any application, e.g., a game, and render the entire scene, as usual on the GPU, using either OpenGL or DirectX, at one sample per pixel (spp). The fragment shaders used for rendering can include all conventional rendering techniques, such as local illumination, texture mapping, environment mapping, screen-space ambient occlusion, G-buffer passes for deferred rendering, light accumulation, shadow mapping, shadow volumes, etc. Most rendering engines consists of several passes, where each pass may generate one or more render targets (RTs). The user of our algorithm can choose to apply high-quality anti-aliasing to any subset of the render targets, and we call these *high-quality render targets* (HQRTs). The high-level idea of our approach is to first render to the RTs and the HQRTs as usual using the GPU with one spp, and then let the CPU refine the pixels, in the HQRTs, that are in need of high-quality anti-aliasing.

While the GPU is no more occupied than usual, our novel contribution is to run our adaptive anti-aliasing algorithm asynchronously on the CPU cores for the HQRTs. Our CPU rendering pipeline consists of three stages, namely, *i*) silhouette edge detection, *ii*) sparse rasterization, and *iii*) sparse anti-aliasing. The number of pixels containing silhouette edges in an image is relatively small (often less than 5% in our experience), and more expensive, higher-quality anti-aliasing will be applied to only this sparse set of *important* pixels. The fragment shading for the sparse set of pixels is done on the CPU, where the render targets are accessed via shared memory. This makes our CPU shading extremely efficient. Finally, the high-quality versions of the sparse set of important pixels are written back to the HQRT, also located in shared memory.

In our experience, it is often sufficient to let only the final render pass (before any post-processing techniques, such as tone mapping, etc, are applied) render to an HQRT and let all other render targets be rendered at one spp on the GPU. This makes the impact of our algorithm very small. An illustration of such a setup is shown in Figure 2, where the final render pass uses an HQRT. In this setup, the fragment shader in the final render pass also accesses all previously generated render targets. Note that this is just an example—there are essentially an endless number of variations possible.

4 GPU Rendering Pipeline

The task of the GPU is simply to render all RTs and all HQRTs as usual with only 1 spp. As we will see in Section 5, the RTs and HQRTs may be accessed from the fragment shader running sparsely on the CPU, and in addition, the HQRTs will be updated sparsely with high-quality anti-aliased pixels. Hence, the RTs and HQRTs need to be shared between the CPU and the GPU. In the worst case, sharing resources means copying data from the GPU to the CPU every frame, and in the best case, it simply means reading from the same memory pointer as the GPU pipeline uses. The latter is a reality when working with a shared memory architecture, such as the Intel Ivy Bridge [Piazza 2012] with an integrated graphics processor. Therefore, our algorithm is designed around an architecture that can share the address space between the CPU and the GPU.

Even without shared memory, a duplication of static vertex buffers and textures generally works well. However, rendering gets substantially less efficient when there is a frequent use of render targets. We elaborate on this in Section 5.3.2.

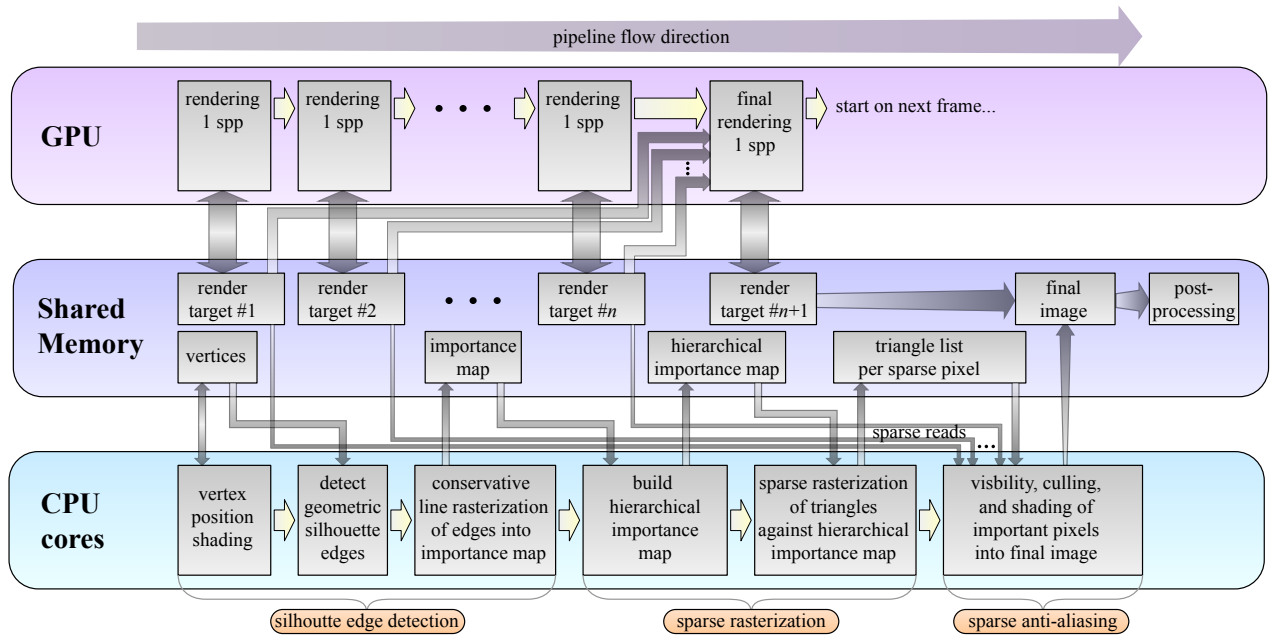


Figure 2: In this example, we assume that n render targets, with one sample per pixel (spp), are rendered by the GPU, and that for the final rendering on the GPU, all render targets are read and used in the fragment shader for the final image. One or more post-processing passes may follow. The work of the GPU is shown at the top, the work done by the CPU cores is shown at the bottom, and the shared memory, which glues together the work done by the CPU and the GPU, is shown in the middle. Note that the CPU rendering pipeline consists of three stages, namely, silhouette edge detection, sparse rasterization, and sparse anti-aliasing. The CPU pipeline works on a sparse set of important pixels (see Figure 1) that are in need of high-quality anti-aliasing. One reason that our algorithm is fast is that it can exploit the render targets created rapidly by the GPU, and sparsely read from these render targets directly from the CPU, thereby avoiding the often expensive copy from GPU to CPU. In the same way, the CPU can update a sparse set of pixels directly into the final image since the memory is shared.

5 CPU Rendering Pipeline

As mentioned in Section 3, the purpose of the CPU rendering pipeline is to detect a sparse set of *important* pixels that contain geometrical silhouette edges, and then sparsely apply a high-quality anti-aliasing algorithm to only these pixels. Our CPU pipeline is heavily optimized for rasterizing triangles to this sparse set of *important* pixels, and in this section, we describe the algorithmic side of our approach, while the implementation details, which depends on the target architecture, are described in Section 6. The three stages of our CPU rendering pipeline are silhouette edge detection, sparse rasterization, and sparse anti-aliasing, and these stages are executed for all HQRs. The following subsections contain descriptions of these stages.

Similar to the GPU pipeline, the input to our pipeline is organized into *draw calls*. Each draw call contains information about a group of vertices with the same format and rendering state, connected into triangles by an implicit relationship, or explicitly by a list of indices. The following subsections will refer to vertices and triangles in the context of all draw calls.

5.1 Silhouette Edge Detection

Here, we describe a straightforward silhouette edge detection mechanism, illustrated by the three leftmost gray boxes at the bottom in Figure 2, used in our algorithm. First, the clip-space positions for all vertices are computed. Note that we avoid computing other per-vertex attributes, and defer them until they are possibly needed. As will be seen later in Section 5.3, the vertex attributes are only computed sparsely for the vertices that are accessed. Next, the triangle edges are tested to determine whether they are geometrical silhouette edges. The usual convention is to treat an edge as a silhouette

if the edge is shared by one front-facing and one back-facing triangle for closed models, or if the triangle edge is only connected to a single front-facing triangle [Crow 1977a; Bergeron 1986]. As a side effect, we mark edges as silhouettes whenever there is a difference in attributes on the edge, since the vertices are then distinct in the vertex buffer. For example, all edges of the quadrilaterals of a cube will be marked as silhouettes because the normals of the quadrilaterals differ.

All silhouette edges are then clipped against the canonical clip-space volume. The silhouette edges, which possibly have been clipped, are then conservatively rasterized into an image, called the *importance map*, with the same resolution as the HQR. Note that we use the term *important* to indicate that a more sophisticated anti-aliasing algorithm should be applied to those pixels. Hence, it suffices with a single bit per pixel in the importance map, where zero indicates no further need of anti-aliasing. For conservative line rasterization, we use the two-dimensional version of Amanatides and Woo’s DDA algorithm [1987].

Although geometrical silhouette edges are unable to capture aliasing from intersecting triangles, a trivial extension can allow additional important pixels to be specified from, e.g., discontinuities in the color buffer, where there is no discontinuity in depth. Another possibility is to allow an artist to paint importance on objects and rasterize those to the importance map. Yet another is simply to mark an entire triangle or object as important. This may be useful when the sampling rate needs to be locally higher for an entire object. We refer to these additions to the importance map as *manual additions*. Manual additions represent a special case because some optimizations cannot be applied when performing shading culling, as described in Section 5.3.1. In particular, we cannot assume that there are no intersecting triangles that are in need of AA.

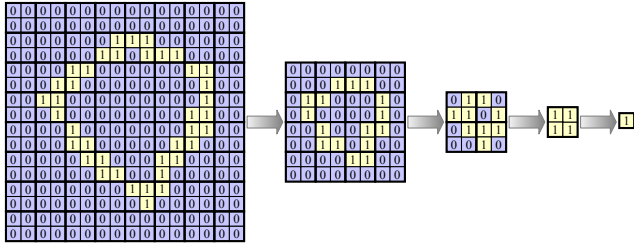


Figure 3: The input from the edge detection phase is an importance map (left), with one bit per pixel indicating whether there is at least one silhouette edge overlapping the pixel. Here, the resolution is 16×16 , but in reality, it may be 2048×1024 , for example. Tree construction is done bottom up (illustrated as left to right in the figure). The bit of a parent is the logical OR of all children’s bits.

5.2 Sparse Rasterization

This subsection describes how our sparse rasterizer, corresponding to the fourth and fifth gray boxes at the bottom in Figure 2, works. Early rejection of candidate triangles is one of the most important design principles of our CPU rasterizer, which makes this stage extremely important.

The sparse rasterization stage commences by creating a quadtree of the importance map from the edge detection phase (Section 5.1). Recall that the importance map contains one bit per pixel, and the bit is set to one if there is at least one silhouette edge overlapping the pixel. We call the resulting quadtree a *hierarchical importance map* (HIM), which is a full tree since the input is the entire image. The HIM will be used in the sparse rasterizer to stop traversal if a triangle does not overlap any pixels that are important. In our case, the bit of a parent is set to one if at least one children bit is one. An example is shown in Figure 3. Note that we have chosen a quadtree, but any hierarchical tree would work.

Next, we describe our sparse rasterization algorithm. The purpose is to find, for each pixel, a list of triangles that may possibly affect the final pixel color. Note that it does not suffice to loop over triangles with silhouette edges, because a triangle without a silhouette edge may still be visible within an important pixel, and can therefore affect the final color of the pixel. A typical example is a background triangle, and a silhouette edge cutting through a pixel. Therefore, *all* triangles are rasterized against the HIM in this stage.

In theory, sparse rasterization is simply the process where a triangle is traversed against the quadtree of the HIM. Traversal continues down into a child node if the triangle overlaps with the spatial extents of the child node, and the corresponding bit in the HIM is set to one, i.e., there is at least one important pixel in the subtree of the child node. The overlap test between a quad and a triangle can be done with an efficient tile test using only additions [Pineda 1988; Akenine-Möller and Aila 2005]. In the following, we will use the term *tile* to denote a $2^n \times 2^n$ region of pixels, where $n \geq 0$, and n can be adjusted for different performance trade-offs. When a node, whose size is equal to the tile size, is reached during traversal, a pointer to that triangle is added to the tile’s triangle list so that the sparse anti-aliasing procedure (Section 5.3) knows which triangles to process in the important pixels. This means that a triangle list will be created for each tile, and these triangles are a superset of the triangles that may affect the final color of the pixels in that tile. In practice, the entire traversal can be done in a more efficient manner, depending on the target architecture. Our current implementation is described in Section 6.

Occlusion culling is very important to performance, and our ap-

proach is similar to z_{max} -culling used in graphics processors [Morcin 2000; Greene et al. 1993], where conservative tests are used. For each tile, a z_{max}^{tile} -value is first initialized to ∞ . If an opaque triangle is processed, and it covers the entire tile, the triangle’s maximum depth, z_{max}^{tri} , inside the tile is computed as the maximum of the triangle depths at the four corners of the tile. If $z_{max}^{tri} < z_{max}^{tile}$ then the maximum depth of the tile is updated: $z_{max}^{tile} = z_{max}^{tri}$. Note that all triangles behind z_{max}^{tile} can be safely culled for opaque geometry. For all triangles that are processed during sparse rasterization, a z_{min}^{tri} -value is also computed as follows. The z_{min}^{verts} is computed as the minimum of the triangle vertices, and z_{min}^{corner} is computed as the minimum of the depths of the triangle plane at the four corners of the tile. A tight, conservative estimation of the minimum depth of the triangle in that tile is then $z_{min}^{tri} = \max(z_{min}^{verts}, z_{min}^{corner})$ [Akenine-Möller et al. 2008]. This z_{min}^{tri} -value is stored with the triangle for that tile. All incoming triangles are culled against the current z_{max}^{tile} , i.e., if $z_{min}^{tri} > z_{max}^{tile}$ then the triangle is *not* added to the triangle list of the tile.

Note that our traversal and data structure (HIM) resemble the work of Greene et al. [1993; 1994; 1996]. However, in our case, the quadtree is built once and used for the remainder of the process, while their quadtrees are updated continuously during rendering of the scene. In addition, our data structure represents a *sparse* set of pixels in need of anti-aliasing, and we do not require a strict front-to-back traversal of all triangles [Greene 1996].

5.3 Sparse Anti-Aliasing

This subsection describes how our sparse visibility computations, culling, and shading are done (rightmost gray box at the bottom in Figure 2). The purpose of this stage is to efficiently resolve visibility among the triangles of a tile and ultimately calculate the color of each important pixel within. All triangles within a tile are processed in submission order. This stage defines sequence points where the blend mode, z -mode, or other pixel back-end state changes. All triangles contained within the same sequence points are treated as a continuous *triangle sequence*, even if triangles belong to different draw calls.

Our current approach to high-quality anti-aliasing is simply adaptive multi-sampled anti-aliasing (MSAA), where several visibility samples are used per pixel and triangle, but only a single shading sample. We have found 16 visibility samples per pixel to be sufficient for most scenes, but this can be specified on a per-pixel level, and can even vary across the image. First, inside testing is performed against the edge equations of a triangle, and for each visible sample, a depth value is computed and a triangle pointer is stored.

Shading is delayed to a point where it is absolutely necessary, generally reducing the amount of shading performed. In the case of opaque geometry, the resolve defers shading within the tile until a triangle sequence with a transparent blend mode starts, or until all triangles within the tile have been processed. In the case of transparent geometry, fragments are shaded for a small group of triangles at a time, e.g., 4 or 8. They are then blended with the tile’s samples in submission order. More details about the shading pipeline can be found in Section 5.3.2. However, first, we describe a number of different culling methods that we use before shading is done.

5.3.1 Culling

Depending on the rendering state of the triangle sequence, different optimizations can be applied. In the common case, where a triangle sequence represents opaque geometry with depth testing enabled, all optimizations are active. First, all triangles whose $z_{min}^{tri} > z_{max}^{tile}$ are removed from the triangle list of the tile. The remaining trian-

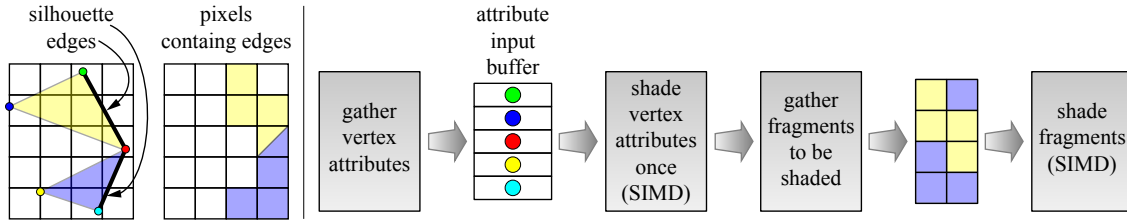


Figure 4: Left: two triangles belonging to the same draw call request that some fragments be shaded since they have silhouette edges in several pixels. Note that one pixel needs to shade both the yellow and the blue triangle. Right: the shading pipeline. First, the vertex attributes are gathered into an input buffer, and then these are shaded in parallel using SIMD execution. Next, the fragments to be shaded are gathered, and finally, these fragments are shaded (which includes attribute interpolation). Note that in reality, more triangles are usually present. For example, one triangle would usually exist between the blue and yellow triangle—we excluded it to make the figure clearer.

gles are reordered to maximize occlusion culling within each pixel. In our implementation, the triangles are sorted based on z_{min}^{tri} .

To allow for occlusion culling at the pixel level, a z_{max}^{pixel} -value, representing the maximum depth of the samples in a pixel, is maintained. Once all samples in the pixel have been covered by opaque geometry, the value is used for occlusion culling. When a triangle in a triangle sequence is found with $z_{min}^{tri} > z_{max}^{pixel}$, the rest of the triangle sequence can be discarded since the triangles are sorted along z_{min}^{tri} . This scheme allows for efficient occlusion culling even when using highly tessellated models.

For opaque geometry, there are also a number of methods to avoid unnecessary shading, and hence can be seen as shading culling techniques. For example, if a single fragment is to be shaded for an important pixel, CPU shading is omitted since it would result in the same shading as already generated by the GPU. As a consequence, the pixel is discarded. Similarly, if there are two fragments that belong to two front-facing triangles *sharing* an edge, which obviously is not a silhouette edge, the corresponding important pixel is discarded. Finally, if there are multiple fragments, but none of the fragments belong to a triangle with a silhouette edge, then there cannot be any visible silhouette edge within the pixel. Hence, the important pixel is discarded. The last optimization assumes that there are no intersecting triangles in need of AA. Therefore, it cannot be applied if the important pixel is a manual addition.

Note that there is a small risk of increased aliasing when the optimizations from the previous paragraph are used. Since fragments are created from discrete samples, there is a possibility that a small triangle ends up between samples when our algorithm is used. The GPU pipeline, on the other hand, might sample this small triangle, and hence produce an aliased pixel, which is different from the resulting pixel from our high-quality rasterizer. The risk of this happening can be reduced to numerical differences by specifying one of our sample points to coincide with the sample location used by the GPU. However, this issue is ultimately a sampling problem that can be solved by increasing the sample rate of our CPU pipeline.

5.3.2 Shading

Self-contained fragment shaders, such as those using static textures and uniforms, are trivial to implement in the CPU pipeline. More advanced shaders, in particular those that use render targets (RTs), may require a more elaborate solution, depending on the underlying architecture. When memory is shared between the GPU and the CPU, RTs (created by the GPU pipeline) are directly accessible to the CPU pipeline. However, when the CPU and the GPU are separated by a relatively slow memory bus, issuing transfers of all RTs from the GPU to the CPU each frame, is prohibitively expensive. One solution is to evaluate RTs lazily when they are accessed by a

CPU shader. However, even though the evaluation becomes sparse, it would become expensive with many RTs. It is also unclear how lazy evaluation could be implemented in a feed-forward rasterizer. For these reasons, we focus on shared memory architectures in this paper, and we note that with a shared memory architecture many more different flavors of an algorithm are possible and can become efficient.

When possible, shading is performed for all deferred samples within the tile simultaneously. This enables SIMD execution over fragments from different important pixels and will thus have better efficiency than traditional quad rendering in our sparse setup. Initially, the samples within each pixel are compacted into per-pixel per-triangle fragments with a sample mask. These fragments are then sorted based on draw call, triangle, and pixel, in that order. All fragments belonging to the same draw call are processed simultaneously so that a single shader invocation can be performed for all draw call fragments. First, all vertex attributes for the draw call triangles are fetched into an input buffer. Then, all attributes are shaded using a single attribute shader invocation. Note that the attributes for each triangle are shaded exactly once within a tile. The resulting shaded attributes are then interpolated to the individual fragments. Finally, the interpolated attributes are used as input to a single fragment shader invocation that calculates the color of each fragment. The resulting colors are then distributed to the individual samples and they are marked as shaded. An example of our shading pipeline is shown in Figure 4. When all triangles have been processed in a tile, the final pixel color is computed as the mean of all the samples' color, which is the last step of our pipeline.

Next, we describe how derivatives are computed in our pipeline. A common way to compute attribute derivatives for, e.g., texture mip map selection, is to render quad fragments and estimate the derivatives as the differences between the samples. In our setup, this would be very inefficient since we are rendering a sparse set of pixels. Therefore, we employ analytical derivatives of the barycentric coordinates. These derivatives can simply replace the ordinary barycentric coordinates for attributes that are to be differentiated, rather than interpolated, before fragment shading. One issue that this fails to deal with is derivatives of indirect texture lookups. In this case, we resort to performing multiple lookups inside the fragment shader in order to perform the differentiation explicitly.

6 Implementation

This section describes our high performance CPU pipeline that implements all the pipeline stages described in Section 5. It is written specifically for the x86-64 architecture targeted at multicore CPUs supporting SSE 4.1 instructions at a minimum. Most of our implementation is independent of the architecture's SIMD width and

simply process more data in Struct of Arrays (SoA) fashion. This means that the same algorithm can use either 4-wide SSE or 8-wide AVX, as well as future SIMD widths, without much adjustment. The inputs to our pipeline, such as vertex buffers, contain regular Array of Structs (AoS) data. Therefore, we make frequent use of register transpose to convert loaded AoS data to SoA form.

Our pipeline makes use of a rasterizer based on two-dimensional edge equations from projected and grid-snapped vertices, with similarities to Larrabee rasterization [Abrash 2009]. While this makes it possible to perform rasterization using fixed point math, we still resort to AVX floating point computations for efficiency. This is fine as long as all floating point values are within ranges that guarantee an exact result. For example, a 32-bit IEEE 754 floating point value can represent a 24-bit fixed point value exactly (not counting the sign bit). In the future, we will investigate using AVX2 since it extends integer arithmetic to 256 bits.

In order to distribute work among multiple CPU cores, the pipeline splits the different stages of Section 5 into various *tasks* that are consumed by a pool of *worker threads*. The *main thread* can queue multiple independent tasks for processing. Each task has a number of *work items* that represent the basic unit of work distributed to the worker threads. The work items in each task have a global order; the first work item of the second task is numbered immediately after the last work item of the first task. This allows all worker threads to acquire their work in a lock-free fashion by using an atomic increment of a shared counter. Each task can spawn new tasks when finished and can thus implement dependency chains. The main thread has the ability to wait for all tasks to finish in order to synchronize different stages.

The vertex position shading stage is trivially implemented by processing multiple vertices simultaneously in SIMD fashion. Each task consists of a single draw call, and each work item consists of a subset of vertices to shade. Once a vertex shading task completes, it immediately queues a task for performing triangle setup for each triangle. Triangle setup tests for trivial rejection against the view frustum, performs clipping, snaps its vertices to a two-dimensional grid, determines its facing, computes a bounding box, and sets up edge equations. The facing information is used to speed up the silhouette detector, which follows in the dependency chain.

The silhouette detection task checks front facing triangles for silhouettes by testing the facing of adjacent triangles. The resulting silhouette edges are clipped against the view frustum and conservatively rasterized to the importance map. In order to avoid the synchronization necessary to set individual bits in the importance map, each entry is instead byte sized. Note that vertex shading, triangle setup, and silhouette detection are performed independently for all draw calls without any synchronization. Before starting the next stage, the main thread waits for all outstanding tasks to finish. Once all draw calls have been shaded, and all silhouettes have been rasterized to the importance map, the HIM is built. First, a task that builds tiled subsets of the HIM is executed in parallel. Then a single thread builds the highest levels of the tree over the generated tiles. The HIM is used when binning triangles in the next stage.

The sparse rasterization stage creates one task for each draw call, where a subset of a draw call’s triangles are processed in each work item. Each work item reads 4 or 8 front facing triangles at a time, depending on the SIMD width of the CPU, and stores them in vector registers in SoA form. Active triangles are indicated by a *lane mask*, which is updated as triangles get rejected. A bounding box is then computed around the active triangles, in order to allow for an immediate jump to the lowest possible level in the HIM. This is done efficiently as `int lvl=32-clz(max(bbw-1,bbh-1))`, where `bbw` and `bbh` are the bounding box width and height, respectively,

and `clz()` is an instruction that counts the number of leading zeroes. Note that level 0 is the highest resolution level in the HIM. All active triangles are then simultaneously traversed against the HIM, while maintaining active triangles in the lane mask. When the tile level is reached, z_{min}^{tri} and z_{max}^{tri} are computed and all triangles, which are currently active in the lane mask, are written to the tile’s triangle list, after testing for occlusion. In order to avoid synchronization, each thread has its own per-tile triangle list.

Once all triangles have been binned to tiles, the sparse anti-aliasing stage creates a single task with work items corresponding to a tile each. Each work item starts by reserving and clearing samples for all important pixels within the tile. Then, the draw call triangles overlapping the tile are read in submission order, by pick sorting from the per-thread triangle lists created in the previous stage. Once a whole triangle sequence have been read, and possibly reordered, the triangles are sample tested. First, the edge equations for a SIMD width of triangles are read. Then, a lookup table, containing the result of evaluating the edge equations at the local sample positions, is set up in order to accelerate the sample tests. For each important pixel, all loaded triangles are conservatively tested against the pixel extents in parallel. Then, each triangle, possibly overlapping the pixel, is sample tested one at a time. Inside- and depth testing is performed at the sample locations using the previously computed lookup table, offset by the edge equations evaluated at the pixel. This is done in parallel over samples. Shading is computed exactly as described in Section 5.3. Note that since the CPU cores work in parallel with the graphics processor, our AA algorithm does not increase the frame latency, which is a highly desired feature.

Our code base is rather extensive, and in order for others to be able to reproduce the results and make use of our algorithm, the source code of our CPU rasterizer is released under the MIT license.

7 Results

For all our results, we have used an Intel Ivy Bridge Core i7 (3770K) at 3.5 GHz with four cores (eight threads) and with an integrated graphics processor (HD Graphics 4000) containing 16 EUs (execution units) for unified shading. The EUs are capable of a theoretical 166.4/294.4 GFLOPS without/with Turbo, while the CPU cores have a theoretical peak of 224/249.6 GFLOPS. The thermal design power (TDP) for this chip is 77 W, including both the CPU and the GPU. Our machine uses Windows 7 as operating system, and we have implemented our algorithm using DirectX 11 with the *DirectResourceAccess*¹ extension for shared memory from Intel.

Four different scenes have been used for our experiments. These are *Chess*, *Sponza*, *Buddha*, and *Hairball*. Chess and Hairball both have Phong shading and shadow mapping with one light source. Sponza also has Phong shading and one shadow map, but also texturing and alpha-textured transparent flowers. Silhouette detection was disabled for the flowers, because their textures are transparent towards the edges. This is optional, however. The Buddha scene has four Phong shaded light sources, each with an individual shadow map. Percentage-closer filtering (PCF), for shadow mapping, was implemented in the shader by interpolating four nearest neighbor samples.² We compare our asynchronous adaptive anti-aliasing algorithm, which is denoted A4 and is described in Sections 3-6, against both the integrated GPU and against WARP [Glaister 2008], which is an optimized software rasterizer used as a fallback in Windows 7. For the GPU and WARP, we use MSAA with 4 and 8 spp, where 8 spp is the highest setting for these two renderers. For A4,

¹Also called *InstantAccess*.

²Hardware PCF could not be combined with R32 float render targets, which we use for shadow maps on our current platform.



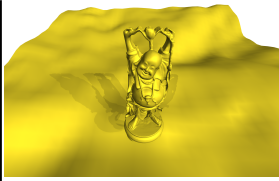
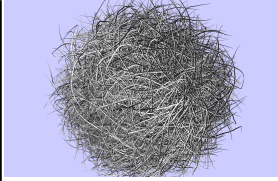
Scene	Chess			Sponza			Buddha			Hairball		
												
# triangles	64k			262k			1.1M			2.85M		
resolution	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
% imp. pixels	1.6%	1.0%	0.7%	4.4%	2.7%	1.9%	1.5%	0.8%	0.5%	31%	21%	15%
WARP 4 spp	25 ms	55 ms	107 ms	62 ms	114 ms	208 ms	154 ms	225 ms	349 ms	319 ms	418 ms	530 ms
WARP 8 spp	37 ms	90 ms	169 ms	81 ms	163 ms	310 ms	188 ms	277 ms	433 ms	399 ms	564 ms	761 ms
A4 16spp100%	26 ms	56 ms	101 ms	50 ms	104 ms	186 ms	96 ms	156 ms	247 ms	126 ms	182 ms	254 ms
A4 8 spp	5 ms	9 ms	15 ms	15 ms	22 ms	31 ms	34 ms	40 ms	52 ms	110 ms	149 ms	193 ms
A4 16 spp	5 ms	9 ms	15 ms	16 ms	23 ms	34 ms	37 ms	40 ms	52 ms	119 ms	160 ms	207 ms
GPU 1 spp	5 ms	7 ms	12 ms	8 ms	12 ms	18 ms	25 ms	33 ms	44 ms	40 ms	52 ms	66 ms
GPU 4 spp	7 ms	13 ms	20 ms	14 ms	23 ms	37 ms	30 ms	41 ms	55 ms	64 ms	111 ms	169 ms
GPU 8 spp	10 ms	20 ms	35 ms	20 ms	37 ms	59 ms	34 ms	48 ms	63 ms	112 ms	200 ms	300 ms

Table 1: Statistics for rendering of our four test scenes, where $R_1 = 1280 \times 800$, $R_2 = 2048 \times 1280$, and $R_3 = 2880 \times 1800$ are the resolutions that we have used. The third row shows the percentage of important pixels after culling in the rendered images, and below that, results are shown in time (milliseconds) per frame. Note that asynchronous adaptive anti-aliasing (A4) method uses 8 and 16 samples per pixel (spp), while both WARP and the GPU only use 4 and 8 spp MSAA. In addition, the row denoted A4 16 spp 100% also uses our algorithm with the difference that **every** pixel is rendered using the CPU pipeline.

results are shown for both 8 and 16 spp. We have used sampling patterns which are digital nets for high quality. For example, for 16 spp, we use a (0, 4, 2)-net in base 2. The results have been generated for 1280×800 , 2048×1280 , and 2880×1800 resolutions, where the latter is the native resolution of a Macbook Pro 15” (2012). We use WARP as a reference for software rendering performance. In practice, A4 has the advantage of letting the GPU render all shadow maps. In order to make the comparison more informative, we do not count the time WARP needs to render the shadow maps. Note that this gives WARP an advantage over A4 in that it does not need to wait for the GPU to finish shadow map rendering.

Our main results are shown in Table 1. For A4, we have used a tile size of 8×8 , since that size gave best results for all scenes and resolutions. One row in the table says “A4 16 spp 100%”, which means that we deliberately set all pixels to be important in our algorithm. As a consequence, all pixels are rendered using the CPU pipeline for that setting. It is interesting to see that with 16 spp, A4 100% is about on par with WARP 4 spp for Chess, but for the rest of the scenes, A4 100% is significantly faster, despite the fact that WARP does no shadow map rendering and A4 has overhead for silhouette detection, HIM generation & construction, and is optimized for sparse rendering & shading. In addition, A4 100% has $4\times$ as many spp. For the hairball, A4 100% is more than twice as fast. Note also that our adaptive A4 algorithm with 16 spp is between $3.9\text{--}7.1\times$ faster than WARP at 4 spp, for the first three scenes, and for the hairball, this is reduced to $2.6\text{--}2.7\times$ faster. Hence, our method generates higher quality images and is faster than WARP.

The other interesting results are just below the A4 100% row, where the numbers for our A4 algorithm are shown together with MSAA numbers for the GPU. Note, for example, that A4 with 8 spp always is as fast or faster than the GPU with 8 spp, for all resolutions and all scenes, and the gap is bigger, the higher resolution. In fact, for the highest resolution, A4 is $1.2\text{--}2.3\times$ faster than the GPU. Comparing A4 with 16 spp vs. GPU with 8 spp, it can be seen that A4 is faster or about the same. Also, our algorithm scales better than the other algorithms with increasing resolution. This is expected since the percentage of the pixels with silhouette edges decreases with higher resolution. We also note that A4 with 8 spp is exactly as fast or just a little bit faster than A4 with 16 spp, which is a consequence of that we have focused on optimizing the 16 spp variant more.

	Chess	Sponza	Buddha	Hairball
Vertex Position Shading	1.4%	1.7%	4.0%	2.1%
Triangle Setup	6.6%	5.9%	21.0%	13.5%
Silhouette Detection	3.1%	3.5%	10.0%	8.3%
HIM Build	3.1%	0.6%	0.7%	0.1%
Sparse Rasterization	14.4%	16.3%	19.2%	17.3%
Sparse AA - Visibility	40.9%	37.1%	28.4%	36.3%
Sparse AA - Shading	23.0%	30.8%	15.9%	21.3%
Copy to frame buffer	7.6%	4.1%	0.9%	1.1%

Table 2: The time spent in the different stages of our algorithm.

The time spent in the different stages of our pipeline is summarized in Table 2 for our test scenes at 2048×1280 . As can be seen, most of the time is spent in the later stages of the pipeline, much like a real GPU pipeline. The Buddha scene is a bit different and spends more time in the earlier stages. This is expected since Buddha has very few silhouettes compared to the number of triangles in the scene, which means that most work is culled before the later stages. It is also interesting to look at the CPU load using our algorithm, e.g., A4 with 16 spp for the scenes in Table 1. For Chess, it varies from 52% (highest resolution) to 72% (lowest resolution). For Sponza, the corresponding numbers are 84%–91%, and for Buddha: 72%–89%. The hairball is more extreme, due to the many silhouettes, and the CPU load is about 99% all the time. The CPU load is related to how much A4 needs to be idle and wait for the GPU to render shadow maps and the frame buffer at 1 spp. The fact that the CPU load generally decreases with higher resolution is evidence that the GPU becomes the bottleneck at higher resolution.

To determine how A4 scales with increasing shader complexity, a shader with varying number of diffuse point lights and no textures/shadow maps, was applied to all scenes. The results are shown in Figure 5, where it is clear that our algorithm just becomes a small offset to GPU 1 spp at some point. One interpretation for this is that more expensive shaders are beneficial to our algorithm. This and better resolution scaling are two really important features of A4. The results also indicate what factors contribute to increased rendering time when using MSAA, and hence how A4 can improve performance. The difference in time at zero point lights can be interpreted as the cost of additional visibility computations, while the slope of the curve indicates the amount of extra shading. The

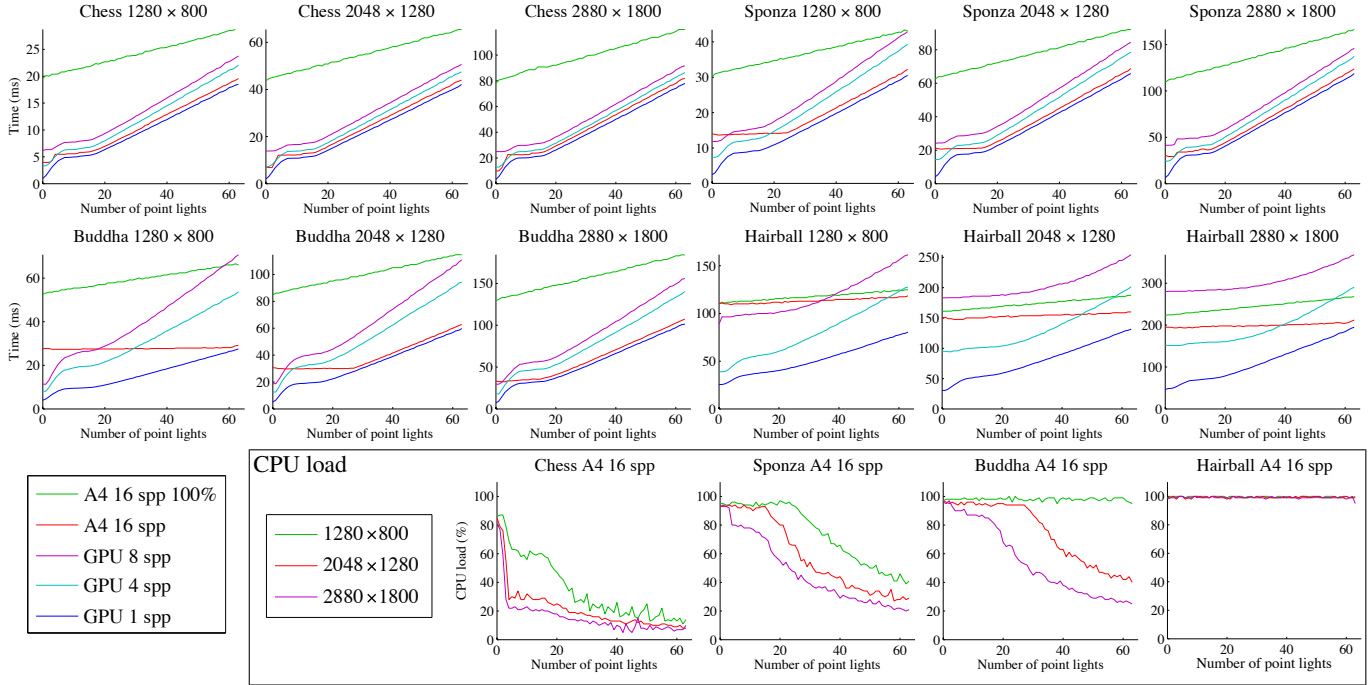


Figure 5: All scenes rendered at different resolutions using a variable number of diffuse point light sources. Each plot represents one scene at one resolution rendered using both A4 and the GPU at different settings. The number of point lights increase along the x-axis, and the execution time is indicated by the y-axis. In the framed region at the bottom, CPU load over the number of light sources is visualized for all configurations of A4 16 spp. CPU usage was measured by polling performance counters provided by the operating system.

amount of redundant shading on the GPU due to quad shading is hard to estimate, but we note that A4 100% scales best with increased shading complexity. This behavior can be attributed to, e.g., reduced quad shading overhead and better occlusion culling.

Next, we will discuss some disadvantages with our method. First, our current implementation does not handle tessellation. This would require our vertex position shading stage to also tessellate, compute the positions of the newly created triangles, and then detect silhouette edges after that. This would require accurate knowledge about how the hardware tessellator works, and therefore, it is left as future work. Second, for best results, surfaces should not intersect. This can, however, be avoided by the artists at the cost of time. Third, our algorithm is designed around the fact that many scenes have rather few important pixels, which need high-quality AA. A difficult scene is therefore the hairball (right in Table 1), as seen in Figure 6. However, the results here were surprising. A4 (8/16 spp) rendered the scene as fast or faster than the GPU at 8 spp. In fact, for the two higher resolutions, A4 100% was also faster than the GPU at 8 spp.

Finally, we have also measured image quality, in terms of peak signal to noise ratio (PSNR), at 2048×1280 against a ground truth image rendered with 256 samples per pixel with MSAA. The results are shown below in dB (decibels), i.e., higher numbers are better.

	Chess	Sponza	Buddha	Hairball
GPU 4 spp	48.9 dB	48.5 dB	50.1 dB	33.1 dB
GPU 8 spp	52.8 dB	51.9 dB	53.6 dB	37.1 dB
A4 16 spp	54.9 dB	47.1/52.0 dB	54.9 dB	40.1 dB

As expected, the PSNR is substantially higher for A4 using 16 spp compared to the GPU using 4 & 8 spp. The exception is Sponza, where the PSNR drops to 47.1 dB, which is mostly due to our current handling of alpha-textured geometry. If transparent geometry also generates silhouettes, PSNR increases to 52.0 dB, which is a bit

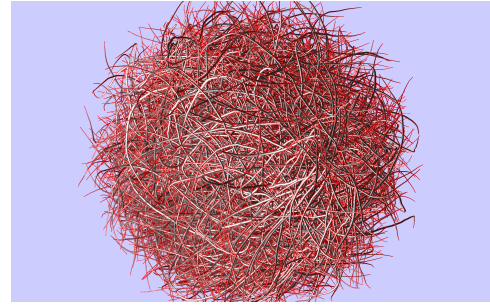


Figure 6: Hair ball rendered with A4 with 16 spp in 207 ms at 2880×1800 . This is a difficult case for our algorithm, since there are many silhouette edges. All important pixels (after all sorts of culling) are marked as red. The GPU used 300 ms with 8 spp MSAA.

better than the GPU at 8 spp. Even though the geometry gets transparent toward the edges, there are still some hard edges that could be resolved with more careful texture mapping. Note also that the rendering time increases from 23 to 34 ms when transparent geometry generates silhouettes, which is still faster than the GPU at 8 spp. This rather substantial increase comes from the fact that shader culling will be inefficient for all tiles with overlapping transparent geometry in our current implementation. There is no way to cull before the alpha geometry has been rendered, and at that point, the tile has already shaded everything for blend operations to work. We note that this could be resolved by having conservative z_{min} for all silhouettes within an important pixel, and perform shader culling if $z_{silhouette}^{min} > z_{pixel}^{max}$, but leave this to future work.

As mentioned earlier, it is possible to apply our algorithm to *only* as many important pixels as there is time before the target frame time has been reached. Recently, Guenter et al. [2012] used eye tracking to render in high resolution only in a small gaze region of the

viewer's eyes. With our algorithm, we could use a similar system, but instead increase the sampling rate only in the gaze regions, for faster rendering. This is left for future work, but it gives some confidence that our platform provides new and interesting alternatives. The energy efficiency of A4 is also left for future work, i.e., since our approach uses both the graphics processor and the CPU cores in the chip, it may be that more energy is used compared to using only $16\times$ MSAA on the graphics processor. However, since our method used less time to solve the problem, it is possible (but not certain) that our method also uses less energy.

8 Conclusions and Future Work

We have presented a real-time hybrid rasterization pipeline, which uses both the CPU and its integrated graphics processor, for adaptive edge anti-aliasing. The key hardware component that glues together the communication and sharing of resources between the GPU and the CPU is a shared memory architecture. Our results are very promising, and in particular, we believe that our results shows that a shared memory architecture can be of great benefit to researchers and developers. In addition, the resolutions of commodity devices have been increasing rapidly over the last few years, and our algorithm scales better with higher resolutions. This makes it an interesting alternative for integrated CPU and GPU architectures, now and for even higher display resolutions (e.g., 3k and 4k).

Most GPU research has focused on discrete graphics cards, but we believe that shared memory architectures open up for a wide range of novel rendering algorithms. Our system can be seen as a starting point for a more general platform for hybrid rendering, and in the future, we would like to investigate other uses, e.g., ray tracing. We also want to explore whether the sampling rate can be increased adaptively when a small triangle is missed inside a pixel. Furthermore, we also want to incorporate analytical [Catmull 1978; Auzinger et al. 2013] and semi-analytical methods [Barringer et al. 2012] into our renderer. It may, e.g., be possible to develop an analytical method that handles 2–4 triangles per pixel, and then revert to MSAA for more triangles. For future work, we will also attempt to augment Crow's method to motion blur and depth of field.

Acknowledgements Thanks to Aaron Lefohn, Charles Lingle, Axel Mamode, Petrik Clarberg, Richard Huddy, and Tom Piazza at Intel. The Buddha comes from the Stanford Computer Graphics Laboratory, Hairball is courtesy Samuli Laine, and Sponza is courtesy of Marko Dabrovic/Frank Meinel. Tomas is a Royal Swedish Academy of Sciences Research Fellow supported by a grant from the Knut and Alice Wallenberg Foundation.

References

- ABRASH, M. 2009. Rasterization on Larrabee. *Dr. Dobbs's Journal* (May). <http://www.drdobbs.com/parallel/rasterization-on-larrabee/217200602>.
- AILA, T., MIETTINEN, V., AND NORDLUND, P. 2003. Delay Streams for Graphics Hardware. *ACM Transactions on Graphics*, 22, 3, 792–800.
- AKENINE-MÖLLER, T., AND AILA, T. 2005. Conservative and Tiled Rasterization Using a Modified Triangle Set-Up. *Journal of Graphics Tools*, 10, 3, 1–8.
- AKENINE-MÖLLER, T., HAINES, E., AND HOFFMAN, N. 2008. *Real-Time Rendering*, 3rd ed. AK Peters Ltd.
- AMANATIDES, J., AND WOO, A. 1987. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Eurographics*, 3–10.
- ANDERSSON, J. 2012. Five Major Challenges in Real-Time Rendering. In *Beyond Programmable Shading course*, SIGGRAPH.
- AUZINGER, T., WIMMER, M., AND JESCHKE, S. 2013. Analytic Visibility on the GPU. *to appear in Computer Graphics Forum (Eurographics 2013)*.
- BARRINGER, R., GRIBEL, C. J., AND AKENINE-MÖLLER, T. 2012. High-Quality Curve Rendering using Line Sampled Visibility. *ACM Transactions on Graphics*, 31, 6, 162:1–162:10.
- BERGERON, P. 1986. A General Version of Crow's Shadow Volumes. *IEEE Computer Graphics and Applications*, 6, 9, 17–28.
- CARPENTER, L. 1984. The A-buffer, an Antialiased Hidden Surface Method. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, 103–108.
- CATMULL, E. 1978. A Hidden-Surface Algorithm with Anti-Aliasing. In *Computer Graphics (Proceedings of SIGGRAPH 78)*, 6–11.
- CROW, F. 1977. Shadow Algorithms for Computer Graphics. In *Computer Graphics (Proceedings of SIGGRAPH 77)*, 242–248.
- CROW, F. C. 1977. The Aliasing Problem in Computer-Generated Shaded Images. *Communications of the ACM*, 20, 11, 799–805.
- FATAHALIAN, K., BOULOS, S., HEGARTY, J., AKELEY, K., MARK, W. R., MORETON, H., AND HANRAHAN, P. 2010. Reducing Shading on GPUs using Quad-Fragment Merging. *ACM Transactions on Graphics*, 29, 67:1–67:8.
- GLAISTER, A., 2008. Windows Advanced Rasterization Platform (WARP) Guide. MSDN, November.
- GREENE, N., AND KASS, M. 1994. Error-Bounded Antialiased Rendering of Complex Environments. In *Proceedings of SIGGRAPH*, 59–66.
- GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical Z-Buffer Visibility. In *Proceedings of SIGGRAPH*, 231–238.
- GREENE, N. 1996. Hierarchical Polygon Tiling with Coverage Masks. In *Proceedings of SIGGRAPH*, 65–74.
- GUENTER, B., FINCH, M., AND SNYDER, J. 2012. Foveated 3D Graphics. *ACM Transactions on Graphics*, 31, 6, 164:1–164:10.
- JIMENEZ, J., GUTIERREZ, D., YANG, J., RESHETOV, A., DEMOREUILLE, P., BERGHOFF, T., PERTHUIS, C., YU, H., MCGUIRE, M., LOTTES, T., MALAN, H., PERSSON, E., ANDREEV, D., AND SOUSA, T. 2011. Filtering Approaches for Real-Time Anti-Aliasing. In *ACM SIGGRAPH 2011 Courses*.
- MOREIN, S. 2000. ATI Radeon HyperZ Technology. In *Workshop on Graphics Hardware, Hot3D Proceedings*.
- PIAZZA, T. 2012. Processor Graphics. In *High Performance Graphics – Hot3D Talks*.
- PINEDA, J. 1988. A Parallel Algorithm for Polygon Rasterization. In *Computer Graphics (Proceedings of ACM SIGGRAPH 88)*, 17–20.
- RESHETOV, A. 2009. Morphological Antialiasing. In *High Performance Graphics*, 109–116.
- SANDER, P. V., HOPPE, H., SNYDER, J., AND GORTLER, S. J. 2001. Discontinuity Edge Overdraw. In *Symposium on Interactive 3D Graphics*, 167–174.
- WARNOCK, J. W. 1969. A Hidden Surface Algorithm for Computer Generated Halftone Pictures. Tech. rep., Utah University.
- YOUNG, P. 2007. Coverage-Sampled Anti-Aliasing. Tech. rep., NVIDIA.