CHAPTER 22

Improving Temporal Antialiasing with Adaptive Ray Tracing

Adam Marrs, Josef Spjut, Holger Gruen, Rahul Sathe, and Morgan McGuire
NVIDIA

ABSTRACT

In this chapter, we discuss a pragmatic approach to real-time supersampling that extends commonly used temporal antialiasing techniques with adaptive ray tracing. The algorithm conforms to the constraints of a commercial game engine, removes blurring and ghosting artifacts associated with standard temporal antialiasing, and achieves quality approaching 16× supersampling of geometry, shading, and materials within the 16 ms frame budget required of most games.

22.1 INTRODUCTION

Aliasing of primary visible surfaces is one of the most fundamental and challenging limitations of computer graphics. Almost all rendering methods sample surfaces at points within pixels and thus produce errors when the points sampled are not representative of the pixel as a whole, that is, when primary surfaces are undersampled. This is true regardless of whether the points are tested by casting a ray or by the amortized ray casts of rasterization, and regardless of what shading algorithm is employed. Even "point-based" renderers [15] actually ray trace or splat points to the screen via rasterization. Analytic renderers such as perfect beam tracing in space and time could avoid the ray (under)sampling problem, but despite some analytic solutions for limited cases [1], point samples from ray or raster intersections remain the only fully developed approach for efficient rendering of complex geometry, materials, and shading.

Aliasing due to undersampling manifests as jagged edges, spatial noise, and flickering (temporal noise). Attempts to conceal these errors by wider and more sophisticated reconstruction filters in space (e.g., morphological antialiasing (MLAA) [22], fast approximate antialiasing (FXAA) [17]) and time (e.g., subpixel morphological antialiasing (SMAA) [12], temporal antialiasing (TAA) [13, 27]) convert those artifacts into blurring (in space) or ghosting (blurring in time). Under a *fixed* sample count per pixel across an image, the only true solution to aliasing is to increase the sample density and band-limit the signal being

E. Haines, T. Akenine-Möller (eds.), *Ray Tracing Gems*, https://doi.org/10.1007/978-1-4842-4427-2_22

sampled. Increasing density helps but does not solve the problem at rates affordable for real time: supersampling antialiasing (SSAA) incurs a cost linearly proportional to the number of samples while only increasing quality with the square root; multisampling (MSAA)—including coverage sampling (CSAA), surface based (SBAA) [24], and subpixel reconstruction (SRAA) [4]—samples geometry, materials, and shading at varying rates to heuristically reduce the cost but also lowers quality; and aggregation (decoupled coverage (DCAA) [25], aggregate G-buffer (AGAA) [7]) reduces cost even more aggressively but still limits quality at practical rates. For band-limiting the scene, material prefiltering by mipmapping and its variants [19], level of detail for geometry, and shader level of detail reduce the undersampling artifacts but introduce other nontrivial problems such as overblurring or popping (temporal and spatial discontinuities) while complicating rendering systems and failing to completely address the problem.

The standard in real-time rendering is to employ many of these strategies simultaneously, with a focus on leveraging temporal antialiasing. Despite succeeding in many cases, these game-specific solutions require significant engineering complexity and careful hand-tuning of scenes by artists [20, 21]. Since all these solutions depend on a fixed sampling count per pixel, an adversary can always place material, geometric, or shading features between samples to create unbounded error. More recently, Holländer et al. [10] aggressively identified pixels in need of antialiasing from coarse shading and high-resolution geometry passes and achieved nearly identical results to SSAA. Unfortunately, this rasterization-based approach requires processing all geometry at high resolutions even if only a few pixels are identified for antialiasing. Despite cutting the number of shading samples in half, the reduction in frame time is limited to 10%. Thus, we consider the aliasing challenge open for real-time rendering.

In this chapter, we describe a new pragmatic algorithm, *Adaptive Temporal Antialiasing* (ATAA), that attacks the aliasing problem by extending temporal antialiasing of rasterized images with adaptive ray traced supersampling. Offline ray tracing renderers have long employed highly adaptive sample counts to solve aliasing (e.g., Whitted's original paper [26]), but until now hybrid ray and raster algorithms [2] have been impractical for real-time rendering due to the duplication of data structures between ray and raster APIs and architectures. The recent introduction of the DirectX Raytracing API (DXR) and the NVIDIA RTX platform enable full interoperability between data structures and shaders for both types of rendering on the GPU across the full game engine. Crucially, RTX substantially improves ray tracing performance by delivering hardware acceleration of the bounding volume hierarchy (BVH) traversal and triangle intersection tasks on the NVIDIA Turing GPU architecture. Thus, we build on the common idea of

adaptive sampling by showing how to *efficiently* combine state-of-the-art temporal antialiasing solutions with a hybrid rendering approach unlocked by the recent evolution in the GPU ray tracing ecosystem. Shown in Figure 22-1, our method conforms to the constraints of a commercial game engine, eliminates the blurring and ghosting artifacts associated with standard temporal antialiasing, and achieves image quality approaching 16× supersampling of geometry, shading, and materials within a 16 ms frame budget on modern graphics hardware. We provide details from our hands-on experience integrating ATAA into a prototype version Unreal Engine 4 (UE4) extended with DirectX Raytracing support, tuning the adaptive distribution of ray traced samples, experimenting with ray workload compaction optimizations, and understanding ray tracing performance on NVIDIA Turing GPUs.

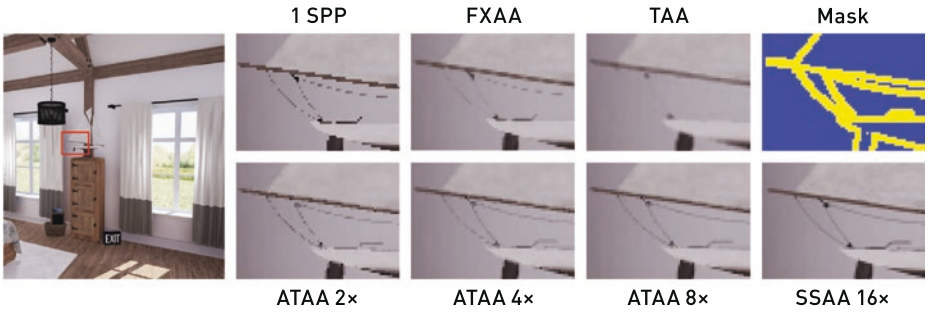

Figure 22-1. *The Modern House scene in Unreal Engine 4 with deferred shading, ray traced shadows, our Adaptive Temporal Antialiasing technique, and a moving camera all rendered in 9.8 ms on an NVIDIA GeForce 2080 Ti. The zoomed-in inlays compare boat rope details rendered with one-sample-per-pixel (1 SPP) rasterization, FXAA, UE4's stock TAA, a visualization of our segmentation mask, ATAA 2×, 4×, and 8×, and an SSAA 16× reference.*

22.2 PREVIOUS TEMPORAL ANTIALIASING

Temporal antialiasing [13, 27] is fast and quite good in the cases that it can handle, which is why it is the de facto standard for games today. TAA applies a subpixel shift to the image plane at each frame and accumulates an exponentially weighted moving average over previous frames, each of which was rendered with only one sample per pixel. On static scenes, TAA approaches the quality of full-screen supersampling. For dynamic scenes, TAA reprojects samples from the accumulated history buffer by offsetting texture fetches along per-pixel motion vectors generated by the rasterizer.

TAA fails in several cases. When new screen areas are disoccluded (revealed) by object motion, they are not represented in the history buffer or are misrepresented by the motion vectors. Camera rotation and backward translation also create thick disocclusions at the edges of the screen. Subpixel

features, such as wires and fine material details, can slip between consecutive offset raster samples and thus can be unrepresented by motion vectors in the next frame. Transparent surfaces create pixels at which the motion vectors from opaque objects do not match the total movement of represented objects. Finally, shadows and reflections do not move in the direction of the motion vectors of the surfaces that are shaded by them.

When TAA fails, it either produces ghosting (blurring due to integrating incorrect values) or reveals the original aliasing as jaggies, flicker, and noise. Standard TAA attempts to detect these cases by comparing the history sample to the local neighborhood of the corresponding pixel in the new frame. When they appear too different, TAA employs a variety of heuristics to clip, clamp, or interpolate in color space. As summarized by Salvi [23], the best practices for these heuristics change frequently, and no general-purpose solution has previously been found.

22.3 A NEW ALGORITHM

We designed Adaptive Temporal Antialiasing to be compatible with conventional game engines and to harness the strengths of TAA while addressing its failures unequivocally and simply. The core idea is to run the base case of TAA on most pixels and then, rather than attempting to combat its failures with heuristics, output a conservative *segmentation mask* identifying where TAA fails and why. We then replace the complex heuristics of TAA at failure pixels with robust alternatives, such as sparse ray tracing, that adapt to the image content. Figure 22-2 shows our algorithm in the context of the Unreal Engine 4 rendering pipeline. In the diagram, rectangular icons represent visualizations of data (buffers) and rounded rectangles represent operations (shader passes). Not all intermediate buffers are shown. For example, where the previous frame's output feeds back as input to TAA, we do not show the associated ping-pong buffers. The new sparse ray tracing step executes in DXR Ray Generation shaders, accepts the new Segmentation buffer, and outputs a new *Sparse Color* buffer that is composited with the dense color output from TAA before tone mapping and other screen-space post-processing.

Since the base case of TAA is acceptable for most screen pixels, the cost of ray tracing is highly amortized and requires a ray budget far less than one sample per pixel. For example, we can adaptively employ 8× ray traced supersampling for 6% of the total image resolution at a cost of fewer than 0.5 rays per pixel. Image quality is then comparable to at least 8× supersampling everywhere; were it not, the boundaries between segmented regions would flicker in the final result due to the different algorithms being employed.

22.3.1 SEGMENTATION STRATEGY

The key to *efficiently* implementing any form of adaptive sampling is to first identify the areas of an image that will benefit most from improved sampling (i.e., detect undersampling) and to then perform additional sampling only in those regions. In ATAA, we guide the adaptivity of ray traced supersampling by computing a screen-space segmentation mask that detects undersampling and TAA failures. The buffer labeled "Segmentation" in Figure 22-2 is a visualization of our segmentation mask generated for the Modern House scene. Figure 22-3 presents a larger, annotated version of this mask. Our mask visualizations map the antialiasing strategy to pixel colors, where red pixels use FXAA, blue pixels use TAA, and yellow pixels use ray traced supersampling. Achieving the ideal segmentation of arbitrary images for ray traced supersampling, while also balancing performance and image quality, is a challenging problem. The budget of rays available for antialiasing may vary based on scene content, viewpoint, field of view, per-pixel lighting and visual effects, GPU hardware, and the target frame rate. As a result, we don't advocate a single "one size fits all" segmentation strategy, but instead we categorize and discuss several options so that the optimal combination of criteria can be implemented in a variety of scenarios.

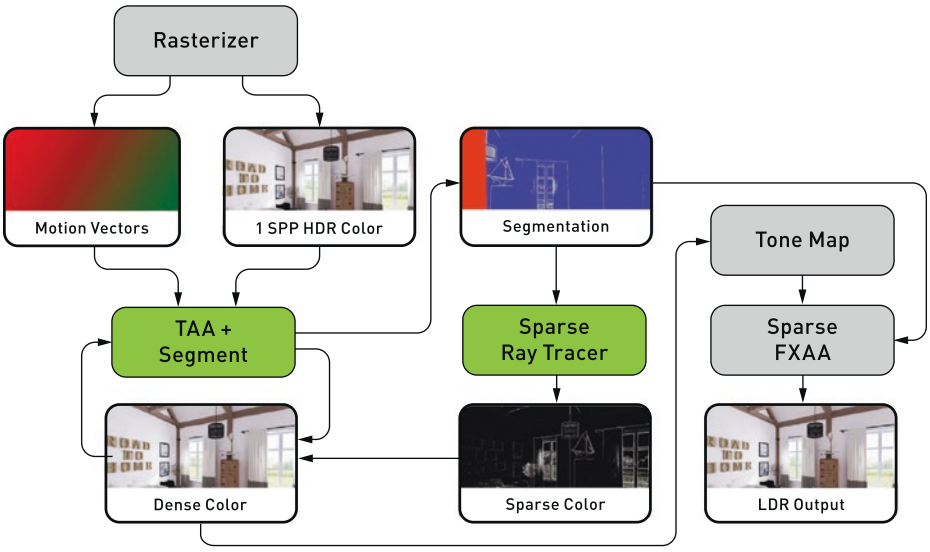

Figure 22-2. *The data flow of ATAA integrated into the UE4 rendering pipeline. Gray boxes represent operations that are either unchanged or slightly modified. Green boxes represent operations that are modified or new. The Segmentation and Sparse Color buffers are new.*

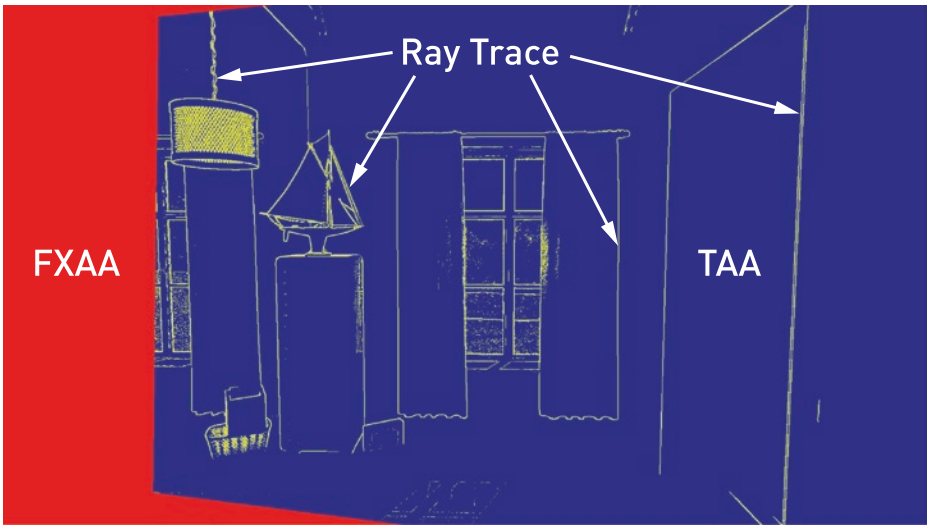

Figure 22-3. *An annotated visualization of an ATAA segmentation mask. Blue pixels use standard TAA, red pixels use FXAA, and yellow pixels use ray traced supersampling.*

22.3.1.1 AUTOMATIC SEGMENTATION

Images can be effectively and efficiently segmented by inspecting the scene data available in screen space after rasterization. Since segmentation is generated algorithmically, without manual intervention from artists or developers, we refer to this process as *automatic segmentation*.

Modern rendering engines maintain per-pixel motion vectors, which we use during segmentation to determine if the current pixel was previously outside of the view (i.e., offscreen) or occluded by another surface. In the case of offscreen disocclusion, temporal raster data does not exist for use in antialiasing. Shown in Figure 22-3, we process these areas with FXAA (red), since it has a low cost, requires no historical data, and runs on the low dynamic range output, i.e., after tone mapping, to conserve memory bandwidth. By running FXAA only at offscreen disocclusion pixels, we further reduce its cost compared to full-screen applications, typically to less than 15% even for rapid camera movement. In the case of disocclusions from animated objects and skinned characters, temporal raster data exists but the shaded color is not representative of the currently visible surface. We eliminate common TAA ghosting artifacts and avoid aliasing caused by TAA clamping, as shown in Figure 22-4, by ignoring the temporal raster data and marking these pixels for ray traced supersampling (yellow). The result of inspecting motion vectors overrides all other criteria and may trigger an early exit in the segmentation process if either type of disocclusion is present. Now that TAA failures from disocclusions are handled, the segmentation process can turn to identifying areas of undersampling.

Figure 22-4. *A skinned character in the middle of a run animation (left). Motion vectors are used to determine disocclusions that cause TAA to fail. TAA ghosting artifacts are eliminated and disocclusions are antialiased by marking these areas for ray traced supersampling (right).*

Undersampling artifacts occur primarily at geometric edges and within high-frequency materials. Similar to common edge detection algorithms, we perform a set of 3 × 3 pixel convolutions to determine the screen-space derivatives of surface normals, depth, mesh identifiers, and luminance. Figure 22-5 visualizes segmentation results for each of these data types.

Not shown in Figure 22-5, we also compare the luminance of the current pixel with that of the reprojected pixel location in the TAA history buffer to determine the luminance change in time as well as space. Since our antialiasing method produces new samples accurately by ray tracing, no error is introduced by the reprojection or potential disocclusions.

As you may have noticed, each of the screen-space data types alone does not provide the complete segmentation we desire. Surface normal derivatives identify interior and exterior object edges effectively, but miss layered objects with similar normals and undersampled materials. Depth derivatives detect layered objects and depth discontinuities well, but create large areas of false positives where sharp changes in depth are common (e.g., planes that are near edge-on to the view, such as walls). Mesh identifier derivatives are excellent at detecting exterior object edges, but miss undersampled edges and materials on the interior of objects. Finally, luminance derivatives detect undersampled materials (in space and time), but miss edges where luminance values are similar. As a result, a combination of these derivatives must be used to arrive at an acceptable segmentation result.

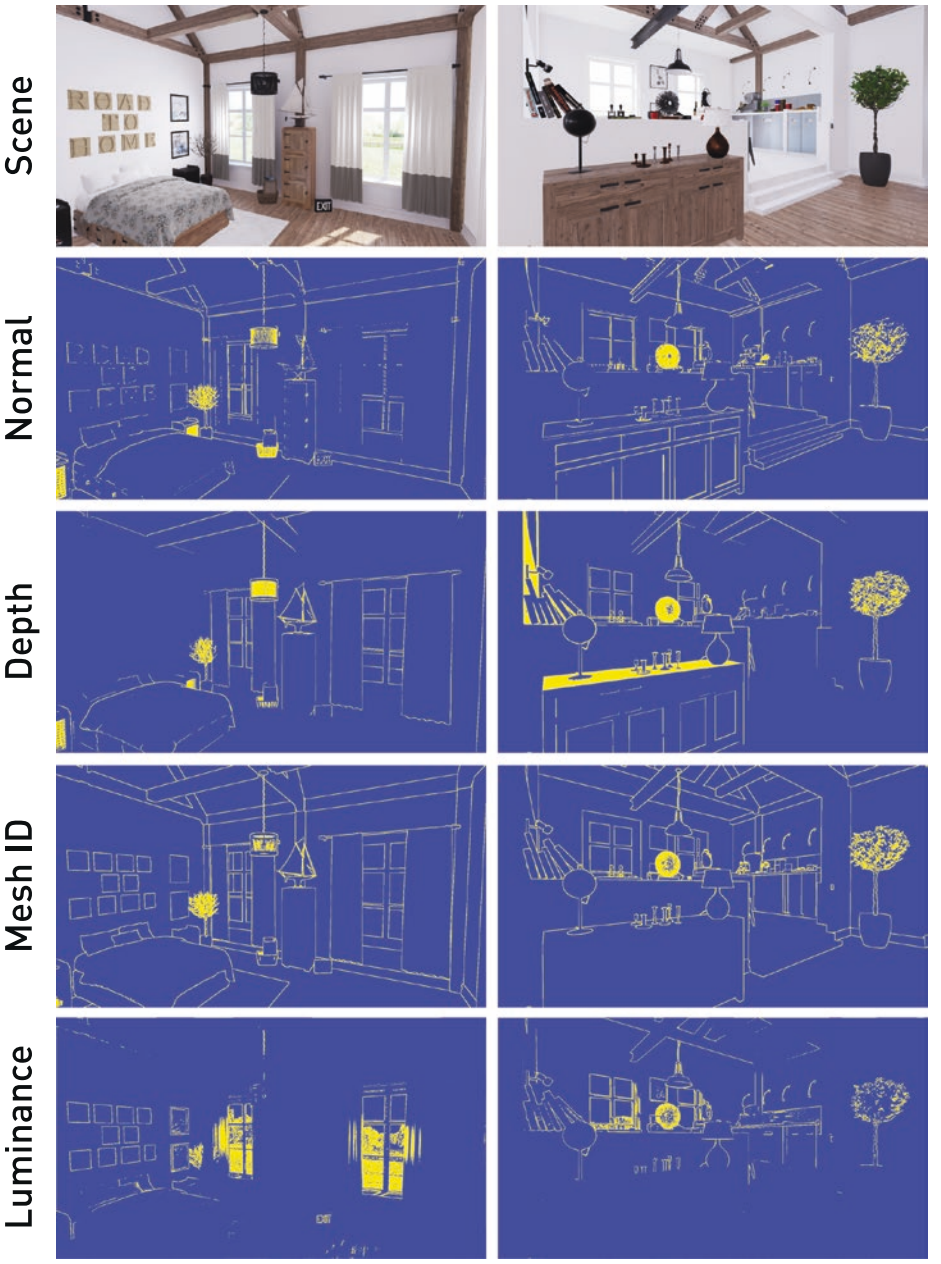

Figure 22-5. *Segmentation results of 3 × 3 pixel convolutions for various types of screen-space data from two viewpoints in the Modern House scene: from top to bottom, final shaded scene, surface normals, depth, mesh identifiers, and luminance.*

22.3.1.2 UE4 AUTOMATIC SEGMENTATION IMPLEMENTATION

In our UE4 implementation, the segmentation mask is generated by extending the existing full-screen TAA post-process pass. After inspecting motion vectors for TAA failures, we use a weighted combination of mesh identifiers, depth, and temporal luminance to arrive at the final segmentation result. The mask is stored as two half-precision unsigned integer values packed into a single 32-bit memory resource. The first integer identifies a pixel's antialiasing method (0 = FXAA, 1 = TAA, 2 = ray tracing), and the second integer serves as a segmentation history that stores whether a pixel received ray traced supersampling in previous frames. The segmentation classification history is important to temporally stabilize the segmentation mask results, as a subpixel jitter is applied to the view every frame for TAA. If a pixel is marked for ray traced supersampling, it will continue to be classified for ray tracing over the next few frames until significant changes in the pixel's motion vectors reset the segmentation history. An alternative to storing segmentation history is to filter the segmentation mask before ray traced supersampling.

22.3.1.3 MANUAL SEGMENTATION

Rendering images in real time presents unique challenges due to the large variation in art, content, and performance goals across projects. Consequently, an automatic segmentation approach may not always produce results that fit within the performance budget of every project. Artists and game developers know their content and constraints best; therefore, a manual approach to segmentation may also be useful. For example, artists and developers may tag specific types of meshes, objects, or materials to write to the segmentation mask during rasterization. Practical examples include hair, telephone wires, ropes, fences, high-frequency materials, and consistently distant geometry. Similar to adaptive tessellation strategies, manual segmentation could also use geometry metadata to guide the adaptivity of ray traced supersampling based on distance to viewpoint, material, or even the type of antialiasing desired (e.g., interior edge, exterior edge, or material).

22.3.2 SPARSE RAY TRACED SUPERSAMPLING

Once the segmentation mask is prepared, antialiasing is performed in a new sparse ray tracing pass implemented with DXR Ray Generation shaders dispatched at the resolution of the segmentation mask. Each ray generation thread reads a pixel of the mask, determines if the pixel is marked for ray tracing, and—if so—casts rays in either the 8×, 4×, or 2× MSAA *n*-rooks subpixel

sampling pattern. At ray hits, we execute the full UE4 node-based material graph and shading pipeline, using identical HLSL code to the raster pipeline. Since forward-difference derivatives are not available in DXR Ray Generation and Hit shaders, we treat them as infinite to force the highest resolution of textures. Thus, we rely on supersampling alone to address material aliasing, which is how most film renderers operate, for the highest quality. An alternative would be to use distance and orientation to analytically select a mipmap level or to employ ray differentials [6, 11]. Figure 22-6 shows a cross section of an image rendered using our method and displays the results of the sparse ray tracing step (top) and the final composited ATAA result (bottom).

Figure 22-6. *A cross section of an image rendered using our method, illustrating the result of the sparse ray tracing step (top) and the final composited ATAA result (bottom).*

22.3.2.1 SUBPIXEL SAMPLE DISTRIBUTION AND REUSE

Raster-based sampling, including that for antialiasing, is restricted to sample patterns available to graphics APIs and implemented efficiently in hardware. While it is possible to add fully programmable sample offset functionality to rasterizer pipelines, such functionality is not readily available today. In contrast, DXR and other ray tracing APIs enable rays to be cast with arbitrary origins and directions, allowing much more flexibility when sampling. If, for example, all useful samples existed on the right half of a pixel, it is possible to adjust the rays to densely sample the right half and leave the left half sparsely sampled (or not sampled at all!). Although completely arbitrary sample patterns are possible, and a variety of potential sample patterns may be worthwhile for particular uses, we suggest a more pragmatic approach.

To maintain parity of sample distribution with surrounding pixels in our hybrid algorithm, it is a natural choice to use the same jittered sample pattern that the rasterizer uses for TAA. With ATAA, we can produce samples from the set of sample positions *at each time step*, resulting in higher-quality new samples and reducing the reliance on reprojected history values. For instance, if TAA has an 8-frame jittered sampling pattern, and we are performing 8× adaptive ray traced supersampling, all eight of the jittered sample locations can be evaluated with rays at each frame. Ray traced supersampling then produces the same result to which TAA converges *prior* to incorporating texture filtering of the history values. Similarly, a 4× adaptive ray tracing sample pattern converges to the 8× TAA result in just two frames.

Even though matching sample patterns between ray tracing and rasterization appears to be the best approach at first, different sample patterns may enable adaptive ray tracing with 8× sampling to converge to 32× quality over just 4 frames. We look to production renderers [3, 5, 8, 9, 16] for inspiration in determining higher-count sample patterns. Correlated multi-jittered sampling [14] is commonly used today. While improved sample patterns should generate higher-quality results, when placed next to the TAA results in screen-space, discontinuities between the different sampling approaches may be noticeable and require further evaluation.

22.4 EARLY RESULTS

To demonstrate the utility of ATAA, we implemented the algorithm in a prototype branch of Unreal Engine 4 extended with DirectX Raytracing functionality. We gathered results using Windows 10 v1803 (RS4), Microsoft DXR, NVIDIA RTX, the NVIDIA 416.25 driver, and the GeForce RTX 2080 and 2070 GPUs.

22.4.1 IMAGE QUALITY

Figures 22-1 and 22-7 show output comparisons of the Modern House scene, shown in full in Figure 22-6, zoomed to challenging areas of the scene that feature thin rope geometry. In Figure 22-7, the "No AA" image demonstrates the baseline aliasing that is expected from a single raster sample per pixel. The FXAA and TAA images represent the standard implementations available in UE4. The SSAA 16× image results from 16× supersampling. We show the ATAA segmentation mask used and three variations of ATAA with 2, 4, and 8 rays per pixel. Since the drawbacks of standard TAA are difficult to capture in still images, and all TAA images come from a stable converged frame, typical TAA motion artifacts are not visible. Figure 22-8 shows output comparisons from the same scene, zoomed to a challenging area featuring a plant with complex branches. In both result

comparisons, notice how standard TAA misses or blurs out thin geometry that falls into the subpixel area between samples, while ATAA's segmentation step identifies much of the region surrounding these tough areas and avoids ghosting, blurring, and undersampling by using ray traced supersampling.

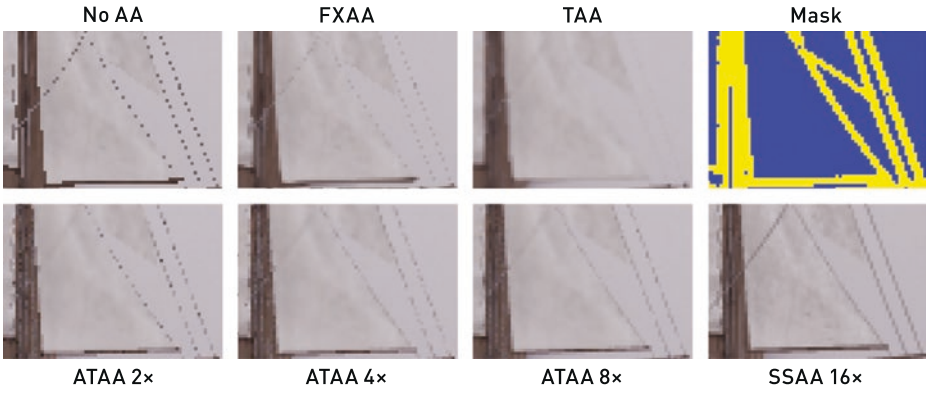

Figure 22-7. *Zoomed images from the Modern House scene highlighting a challenging area featuring thin rope geometry, comparing boat rope details rendered with 1 SPP rasterization, FXAA, UE4's stock TAA, a visualization of our segmentation mask, ATAA 2×, 4×, and 8×, and an SSAA 16× reference.*

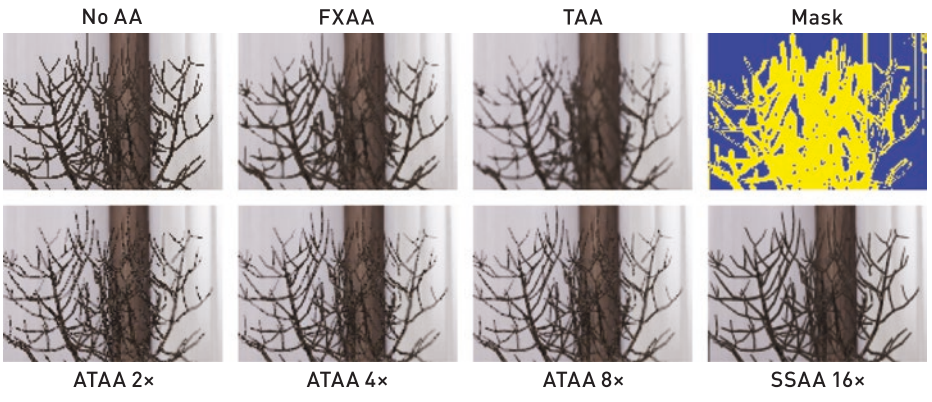

Figure 22-8. *Zoomed images from the Modern House scene highlighting a challenging area featuring complex detail from a plant, comparing plant details rendered with 1 SPP rasterization, FXAA, UE4's stock TAA, a visualization of our segmentation mask, ATAA 2×, 4×, and 8×, and an SSAA 16× reference.*

22.4.2 PERFORMANCE

Table 22-1 shows GPU times, reported in milliseconds (ms), of ATAA compared to equivalent configurations of SSAA. ATAA renders images at 1080p resolution, and the number of rays cast for antialiasing varies per frame according to the segmentation mask. The Modern House view shown in Figure 22-6 is used for performance testing, and the segmentation mask identifies 103,838 pixels for ray

traced supersampling. These pixels represent just 5% of the total image resolution, but combined with non-failure TAA results (blue pixels in the segmentation mask), ATAA adaptively produces results similar to SSAA at a much lower cost. Primary rays cast by ATAA also shoot a shadow ray to the scene's directional light source (the sun) to determine occlusion. In addition, the FXAA pass adds as much as 0.75 ms when the whole frame is new, but in practice scales linearly down to 0 ms as fewer pixels are identified for FXAA in the mask. Under typical camera motion, fewer than 5% of pixels are chosen for FXAA.

Table 22-1. *A comparison of GPU times, in milliseconds, for several SSAA and ATAA configurations on GeForce RTX 2080 and 2070 GPUs. ATAA runs at 1080p resolution and selects 103,838 pixels for ray traced supersampling. ATAA produces similar results compared to SSAA for challenging areas in need of antialiasing while running approximately 2× to 4× faster.*

	GeForce RTX 2080				GeForce RTX 2070		
	SSAA	ATAA	Speedup		SSAA	ATAA	Speedup
2×	6.30	1.81	3.48×	2×	7.00	3.02	2.32×
4×	12.60	3.48	3.62×	4×	14.00	5.94	2.36×
8×	25.20	6.70	3.76×	8×	28.00	11.64	2.41×

In the Modern House scene, the adaptive nature of ATAA produces a significant 2× to 4× speedup compared to SSAA, even with our relatively unoptimized implementation. These early results are captured on new hardware, new drivers, and the experimental DXR API in a prototype branch of UE4 that was not natively designed for ray tracing. Consequently, significant opportunities still remain to optimize the performance of both the ATAA algorithm implementation and the game engine's implementation of DXR ray tracing functionality.

One such algorithmic optimization of ATAA is to create a compact one-dimensional buffer that contains the location of pixels identified for ray traced supersampling, instead of a screen-space buffer that aligns with the segmentation mask, and only dispatch DXR Ray Generation shader threads for elements of the compacted buffer. We refer to this process as *ray workload compaction*. Table 22-2 compares the GPU times of ATAA with and without the compaction optimization. Compaction yields a 13% to 29% performance improvement over the original ATAA implementation and performs approximately 2.5× to 5× faster than the equivalent SSAA configuration. This is an exciting finding, but keep in mind that the segmentation mask (and resulting ray workload) changes dynamically every frame; therefore, compaction may not always be beneficial. Experimentation with various rendering workloads across a project are key to discovering which optimization approaches will achieve the best possible performance.

Table 22-2. *A comparison of GPU times, in milliseconds, for ATAA and ATAA with ray workload compaction (ATAA-C) on GeForce RTX 2080 and 2070 (top). Compaction improves performance of the Modern House workload by 13% to 29% and performs approximately 2.5× to 5× faster than the equivalent SSAA configuration (bottom). Since performance varies with geometry and materials, compaction may not always improve performance.*

GeForce RTX 2080			
	ATAA	ATAA-C	Speedup
2×	1.81	1.47	1.23×
4×	3.48	2.70	1.29×
8×	6.70	5.32	1.26×

GeForce RTX 2070			
	ATAA	ATAA-C	Speedup
2×	3.02	2.67	1.13×
4×	5.95	5.13	1.16×
8×	11.64	9.95	1.17×

	SSAA	ATAA-C	Speedup
2×	6.30	1.47	4.29×
4×	12.60	2.70	4.67×
8×	25.20	5.32	4.74×

	SSAA	ATAA-C	Speedup
2×	7.00	2.67	2.63×
4×	14.00	5.13	2.73×
8×	28.00	9.95	2.81×

22.5 LIMITATIONS

ATAA as presented here does not comprehensively address every issue that you may encounter when implementing hybrid ray-raster antialiasing. For example, the segmentation mask is limited to discovering geometry with a single temporally jittered sample per pixel. As a result, subpixel geometry may be missed. This creates a spatial alternation between geometry appearing and not appearing in the segmentation mask, which therefore causes shifts between high-quality ray traced supersampling and entirely missed geometry. While rendering approaches to solve this problem almost exclusively include increasing the base sample rate, artists may be able to mitigate these issues by modifying geometry appropriately, or by producing alternate level of detail representations when the geometry is placed beyond a certain distance from the camera. Furthermore, filtering the segmentation mask prior to ray tracing may also increase temporal stability of the mask, although at the cost of tracing more rays.

There are minor differences in ATAA's antialiased result compared to SSAA, caused by the material evaluation in DXR not correctly computing and evaluating the texture mipmap level. Texture sampling is particularly challenging when shading ray traced samples in existing production game engines. While it is possible to compute ray differentials, the implementation of existing material models heavily depends on the forward-difference derivatives provided by the raster pipelines. As a result, a single set of ray differentials cannot be used to adjust texture

mipmap level when sampling, making the ray differential computation especially costly. In our implementation, all ray samples select the highest-frequency texture. This limitation results in texture aliasing in many cases, but at higher sample counts we are able to reconstruct the appropriate filtered result. Additionally, TAA history and new raster samples have filtered texture sampling, which can be blended with our ray traced samples to mitigate texture aliasing.

Another practical difficulty for ray traced antialiasing of primary visibility is supporting screen-space effects. Since rays are distributed sparsely across screen space, there is no guarantee the necessary data that post-process effects such as depth of field, motion blur, and lens flare use will exist in nearby pixels. A simple solution is to move the antialiasing step before these passes, at the cost of these effects not benefiting from additional antialiasing. In the long term, as the budget of ray samples increases, it may be sensible to move the raster-based screen-space effects to ray traced equivalents.

22.6 THE FUTURE OF REAL-TIME RAY TRACED ANTIALIASING

The recent arrival of graphics processors with dedicated acceleration for ray tracing creates an opportunity to reassess the state of the art, and in turn reinvent real-time antialiasing. This chapter presents implementation details beyond the initial publication of ATAA [18] and may serve as a foundation upon which production renderers of the future build. A primary remaining concern for production deployment is ensuring that the runtime of the sparse ray tracing pass fits within the available frame time budget. Once the pixels for ray traced supersampling are selected, we suggest pursuing additional heuristics to adjust performance, including naively dropping rays after the target number is hit, deprioritizing rays in screen-space regions where aliasing is less common or perceptually less important, and selecting ray counts based on a priority metric embedded in the segmentation mask. We expect adjusting ray counts per pixel will improve ray tracing performance in a given region of interest. Due to the SIMD architecture of current GPUs, these adjustments are optimally made on warp boundaries, thus pixels requiring similar sample counts may benefit from being placed together when spawning work, a task that can also be completed during a workload compaction pass.

22.7 CONCLUSION

Primary surface aliasing is a cornerstone problem in computer graphics. The best-known solution for offline rendering is adaptive supersampling. This was previously impractical for rasterization renderers in the context of complex materials and scenes because there was no way to efficiently rasterize sparse pixels. Even the most efficient GPU ray tracers required duplicated shaders and scene data. While DXR solves the technical challenge of combining rasterization and ray tracing, applying ray tracing to solve aliasing by supersampling is nontrivial: knowing *which* pixels to supersample when given only 1 SPP input and reducing the cost to something that scales are not solved by naively ray tracing.

We have demonstrated a practical solution to this problem—so practical that it runs within a commercial game engine, operates in real time even on first-generation real-time ray tracing commodity hardware and software, and connects to the full shader pipeline. Where film renderers choose pixels to adaptively supersample by first casting many rays per pixel, we instead amortize that cost over many frames by leveraging TAA's history buffer to detect aliasing. We further identify large, transient regions of aliasing due to disocclusions and employ post-process FXAA there rather than expending rays. This hybrid strategy leverages advantages of the most sophisticated real-time antialiasing strategies while eliminating many of their limitations. By feeding our supersampled results back into the TAA buffer, we also increase the probability that those pixels will not trigger supersampling on subsequent frames, further reducing cost.

REFERENCES

[1] Auzinger, T., Musialski, P., Preiner, R., and Wimmer, M. Non-Sampled Anti-Aliasing. In *Vision, Modeling and Visualization* (2013), pp. 169–176.

[2] Barringer, R., and Akenine-Möller, T. A4: Asynchronous Adaptive Anti-Aliasing Using Shared Memory. *ACM Transactions on Graphics 32*, 4 (July 2013), 100:1–100:10.

[3] Burley, B., Adler, D., Chiang, M. J.-Y., Driskill, H., Habel, R., Kelly, P., Kutz, P., Li, Y. K., and Teece, D. The Design and Evolution of Disneys Hyperion Renderer. *ACM Transactions on Graphics 37*, 3 (2018), 33:1–33:22.

[4] Chajdas, M. G., McGuire, M., and Luebke, D. Subpixel Reconstruction Antialiasing for Deferred Shading. In *Symposium on Interactive 3D Graphics and Games* (2011), pp. 15–22.

[5] Christensen, P., Fong, J., Shade, J., Wooten, W., Schubert, B., Kensler, A., Friedman, S., Kilpatrick, C., Ramshaw, C., Bannister, M., Rayner, B., Brouillat, J., and Liani, M. RenderMan: An Advanced Path-Tracing Architecture for Movie Rendering. *ACM Transactions on Graphics 37*, 3 (2018), 30:1–30:21.

[6] Christensen, P. H., Laur, D. M., Fong, J., Wooten, W. L., and Batali, D. Ray Differentials and Multiresolution Geometry Caching for Distribution Ray Tracing in Complex Scenes. *Computer Graphics Forum 22*, 3 (2003), 543–552.

[7] Crassin, C., McGuire, M., Fatahalian, K., and Lefohn, A. Aggregate G-Buffer Anti-Aliasing. *IEEE Transactions on Visualization and Computer Graphics 22*, 10 (2016), 2215–2228.

[8] Fascione, L., Hanika, J., Leone, M., Droske, M., Schwarzhaupt, J., Davidovič, T., Weidlich, A., and Meng, J. Manuka: A Batch-Shading Architecture for Spectral Path Tracing in Movie Production. *ACM Transactions on Graphics 37*, 3 (2018), 31:1–31:18.

[9] Georgiev, I., Ize, T., Farnsworth, M., Montoya-Vozmediano, R., King, A., Lommel, B. V., Jimenez, A., Anson, O., Ogaki, S., Johnston, E., Herubel, A., Russell, D., Servant, F., and Fajardo, M. Arnold: A Brute-Force Production Path Tracer. *ACM Transactions on Graphics 37*, 3 (2018), 32:1–32:12.

[10] Holländer, M., Boubekeur, T., and Eisemann, E. Adaptive Supersampling for Deferred Anti-Aliasing. *Journal of Computer Graphics Techniques 2*, 1 (March 2013), 1–14.

[11] Igehy, H. Tracing Ray Differentials. In *Proceedings of SIGGRAPH* (1999), pp. 179–186.

[12] Jimenez, J., Echevarria, J. I., Sousa, T., and Gutierrez, D. SMAA: Enhanced Morphological Antialiasing. *Computer Graphics Forum 31*, 2 (2012), 355–364.

[13] Karis, B. High Quality Temporal Anti-Aliasing. Advances in Real-Time Rendering for Games, SIGGRAPH Courses, 2014.

[14] Kensler, A. Correlated Multi-Jittered Sampling. Pixar Technical Memo 13-01, 2013.

[15] Kobbelt, L., and Botsch, M. A Survey of Point-Based Techniques in Computer Graphics. *Computers and Graphics 28*, 6 (Dec. 2004), 801–814.

[16] Kulla, C., Conty, A., Stein, C., and Gritz, L. Sony Pictures Imageworks Arnold. *ACM Transactions on Graphics 37*, 3 (2018), 29:1–29:18.

[17] Lottes, T. FXAA. NVIDIA White Paper, 2009.

[18] Marrs, A., Spjut, J., Gruen, H., Sathe, R., and McGuire, M. Adaptive Temporal Antialiasing. In *Proceedings of High-Performance Graphics* (2018), pp. 1:1–1:4.

[19] Olano, M., and Baker, D. LEAN Mapping. In *Symposium on Interactive 3D Graphics and Games* (2010), pp. 181–188.

[20] Pedersen, L. J. F. Temporal Reprojection Anti-Aliasing in INSIDE. Game Developers Conference, 2016.

[21] Pettineo, M. Rendering the Alternate History of The Order: 1886. Advances in Real-Time Rendering in Games, SIGGRAPH Courses, 2015.

[22] Reshetov, A. Morphological Antialiasing. In *Proceedings of High-Performance Graphics* (2009), pp. 109–116.

[23] Salvi, M. Anti-Aliasing: Are We There Yet? Open Problems in Real-Time Rendering, SIGGRAPH Courses, 2015.

[24] Salvi, M., and Vidimče, K. Surface Based Anti-Aliasing. In *Symposium on Interactive 3D Graphics and Games* (2012), pp. 159–164.

[25] Wang, Y., Wyman, C., He, Y., and Sen, P. Decoupled Coverage Anti-Aliasing. In *Proceedings of High-Performance Graphics* (2015), pp. 33–42.

[26] Whitted, T. An Improved Illumination Model for Shaded Display. *Communications of the ACM 23*, 6 (June 1980), 343–349.

[27] Yang, L., Nehab, D., Sander, P. V., Sitthi-amorn, P., Lawrence, J., and Hoppe, H. Amortized Supersampling. *ACM Transactions on Graphics 28*, 5 (Dec. 2009), 135:1–135:12.