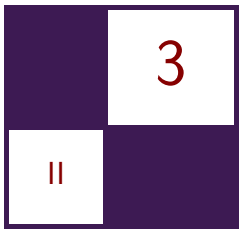

Hybrid Reconstruction Antialiasing

Michał Drobot

3.1 Introduction

In this article, we present the antialiasing (AA) solution used in the Xbox One and Playstation 4 versions of *Far Cry 4*, developed by Ubisoft Montreal: hybrid reconstruction antialiasing (HRAA). We present a novel framework that utilizes multiple approaches to mitigate aliasing issues with a tight performance budget in mind.

The Xbox One, Playstation 4, and most AMD graphics cards based on the GCN architecture share a similar subset of rasterizer and data interpolation features. We propose several new algorithms, or modern implementations of known ones, making use of the aforementioned hardware features. Each solution is tackling a single aliasing issue: efficient spatial super-sampling, high-quality edge antialiasing, and temporal stability. All are based around the principle of data reconstruction. We discuss each one separately, identifying potential problems, benefits, and performance considerations. Finally, we present a combined solution used in an actual production environment. The framework we demonstrate was fully integrated into the Dunia engine's deferred renderer. Our goal was to render a temporarily stable image, with quality surpassing 4× rotated-grid super-sampling, at a cost of 1 ms at a resolution of 1080p on the Xbox One and Playstation 4 (see Figure 3.1).

3.2 Overview

Antialiasing is a crucial element in high-quality rendering. We can divide most aliasing artifacts in rasterization-based rendering into two main categories: temporal and spatial. Temporal artifacts occur as flickering under motion when details fail to get properly rendered due to missing the rasterization grid on certain

Figure 3.1. The crops on the right show no AA (top), SMAA (middle), and the presented HRAA (bottom) results. Only HRAA is capable of reconstructing additional details while providing high-quality antialiasing.

frames. Spatial artifacts result from signal under-sampling when dealing with a single, static image. Details that we try to render are just too fine to be properly resolved at the desired resolution, which mostly manifests itself as jagged edges.

Both sources of aliasing are directly connected with errors of signal under-sampling and occur together. However, there are multiple approaches targeting different aliasing artifacts that vary in both performance and quality. We can divide these solutions into analytical, temporal, and super-sampling–based approaches.

In this article, we present a novel algorithm that builds upon all these approaches. By exploring the new hardware capabilities of modern GPUs (we will base our findings on AMD's GCN architecture), we optimize each approach and provide a robust framework that shares the benefits of each algorithm while minimizing their shortcomings.

3.3 Related Work

A typical antialiasing solution used in offline rendering is to super-sample (SSAA) the image; render at a higher resolution and then perform a resolve step, which is a down-sampling filter into the desired final resolution [Burley 07]. If enough samples are used, then this type of antialiasing tackles all the aliasing problems

mentioned earlier. Unfortunately, it requires effectively rendering the image multiple times and is of limited usefulness for real-time rendering.

An optimized version of super-sampling is provided by graphics hardware in the form of multi-sampled antialiasing (MSAA) [Kirkland et al. 99]. Instead of shading all pixels at higher resolution, only samples along triangle edges are rasterized multiple times (but only shaded once), which is followed by an optimized resolve. MSAA proves to be a valid solution to spatial aliasing issues, but is strictly limited to triangle edges. All samples need to be stored in an additional framebuffer until they are resolved, therefore making this method very expensive in terms of memory consumption. As a result, not many games use it as the main antialiasing solution on performance-limited platforms.

It is worth noting that the number of gradient steps on antialiased edges is strictly correlated to the number of samples that MSAA or SSAA uses (i.e., 4×MSAA can provide a maximum of five gradients depending on the sampling pattern and edge orientation).

On the previous generation of consoles (Xbox 360 and Playstation 3), we observed a rise in popularity of image-based, postprocessing, morphological antialiasing solutions such as FXAA [Lottes 09], MLAA [Reshetov 09], and SMAA [Jimenez et al. 11]. These algorithms provided excellent results, with perceptual quality comparable to extremely high levels of MSAA rendering at a fraction of the cost of MSAA. A typical morphological filter derives visually perceivable edges from the current image and performs edge re-vectorization. Unfortunately the result still relies only on the final rasterized image data, which can suffer from temporal and spatial aliasing. In practice, static images that are processed with these algorithms look much better than what the hardware-based MSAA can achieve. Unfortunately the quality degrades dramatically under motion, where spatial and temporal under-sampling result in "wobbly" edges and temporal flicker of high-contrast details.

It is clear that morphological methods alone will not achieve the high-quality spatio-temporal results of super-sampling. This sparked research in two different directions: analytical- and temporal-based antialiasing. Several researchers experimented with postprocessing methods, augmented by additional information, derived from actual triangle-edge equation. Probably the most well known is GBAA [Persson 11], which calculates per-pixel signed distance to the closest triangle edge. This information is stored in an additional buffer and is used later during a postprocessing pass to effectively rerasterize triangle edge-pixel intersections analytically. This method can provide a high level of quality and perfect temporal stability of triangle edges. Unfortunately, due to its use of a geometry shader pass to gather triangle information, it exhibits poor performance and thus never gained widespread adoption. It is also hindered by multiple other issues that we will discuss in-depth in Section 3.5.

Another approach gaining popularity is based on temporal algorithms that try to perform filtering using previously rendered frames utilizing image temporal

coherency [Nehab et al. 07]. This effectively allows multi-sampled algorithms to be amortized over time [Yang et al. 09]. Several titles use temporal resolves to augment SSAO [Bavoil and Andersson 12, Drobot 11] or to stabilize the final image in motion [Sousa 13]. Some engines experiment with temporal super-sampling [Malan 12]; however, due to a lack of robust sample rejection methods, those approaches are rather conservative, i.e., accumulating only two frames using a limited subset of visible pixels [Sousa 11].

Recently *Killzone: Shadow Fall* used a robust temporal up-sampling method, effectively rendering images with 2× super-sampling. It also used previously reconstructed frames to stabilize images in motion in order to avoid image flickering [Valient 14].

Several researchers have tried to combine the benefits of hardware-based MSAA, temporal sampling and morphological filtering into one combined solution. This resulted in 4×SMAA [Jimenez et al. 12], which combines the quality of SMAA edge gradients with the temporal stability of 2×MSAA and 2×temporal super-sampling. Unfortunately, not many console titles can afford this due to the use of expensive 2×MSAA.

One more research direction has been toward optimizing sampling patterns for multi-sampled approaches [Akenine-Möller 03]. Unfortunately, this approach didn't get much traction in the real-time rendering field due to a lack of hardware and software support for custom sampling patterns. Only a few predefined sampling patterns are supported in hardware-based MSAA modes.

Another hardware-based solution involves augmenting the standard MSAA pipeline with coverage samples that can be evaluated with minimal performance and memory overhead. This solution was, up to this point, a part of the fixed GPU pipeline in the form of EQAA [AMD 11] and CSAA [Young 06].

3.4 Hybrid Antialiasing Overview

Our antialiasing solution can be divided into several components. Each one can stand on its own and can be freely mixed with any other approach.

The aim of each component is to tackle a different source of aliasing, so each algorithm can be used to its best effect in limited use-case scenarios.

Our framework is built around the following components:

- temporally stable edge antialiasing,
- temporal super-sampling,
- temporal antialiasing.

3.5 Temporally Stable Edge Antialiasing

The aim of this component is to provide perceptually plausible gradients, for geometric edges, that remain stable under motion. We do not need to worry about pixel discontinuities that come from texture data or lighting, as that source of aliasing will be taken care of by a different framework component.

In Section 3.3, we briefly discussed potential algorithms that would suit our needs for high-quality edge rendering: morphological and analytical. However, only the latter can provide temporally stable antialiasing. Unfortunately, all purely analytical methods exhibit problems, including performance issues.

We would like to propose a new implementation based on AMD's GCN architecture that makes analytical edge antialiasing virtually free. In Section 3.5.1, we propose several extensions as well as real production issues connected with the method itself. Section 3.5.2 offers a brief introduction to EQAA's inner workings. It also introduces a new algorithm—coverage reconstruction antialiasing—that uses coverage samples from hardware-based EQAA to analytically estimate the edge orientation as well as triangle spatial coverage, building upon previous analytical-only algorithms.

3.5.1 Analytical Edge Antialiasing (AEAA)

The original GBAA algorithm relies on a geometry shader to pass down geometry information to the pixel shader. Interpolators are used to store the distance to the edge in the major direction. Then, the pixel shader selects the closest signed distance to the currently rasterized pixel and outputs it into an additional offscreen buffer. Distance data needs to contain the major axis and the actual signed distance value in range $[-1, 1]$, where 0 is considered to be at the rasterized pixel center. Later, a fullscreen postprocessing pass searches for each pixel's immediate neighbor's closest edges. After an edge crossing the pixel is found, we use its orientation and distance to blend the two nearest pixels accordingly (see Figure 3.2).

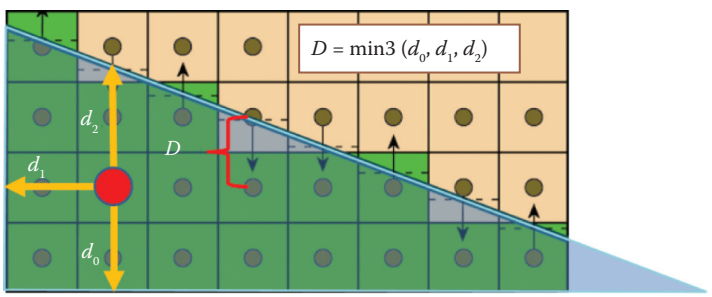

Figure 3.2. In analytical distance-to-edge techniques, every triangle writes out the distance to the closest edge used to antialias pixels in a postprocessing pass.

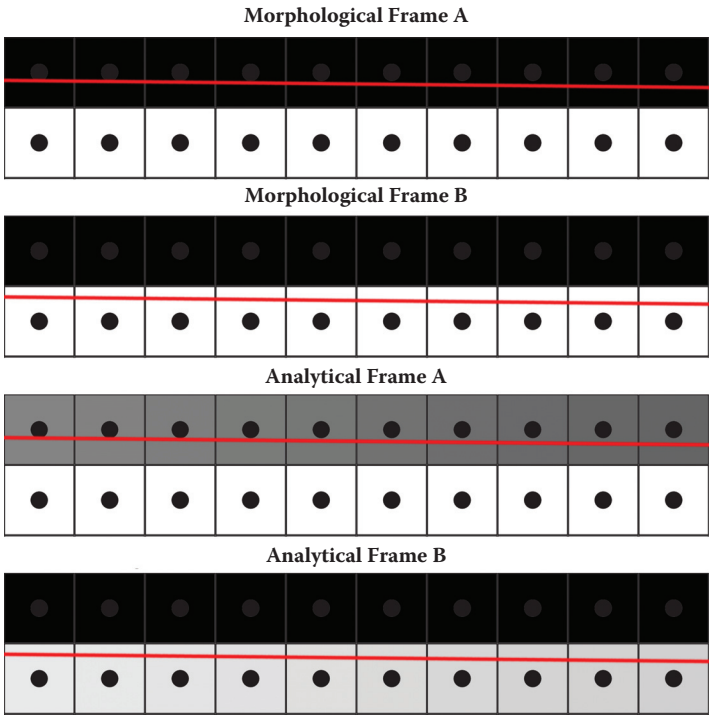

Figure 3.3. Antialiased edge changes in motion when using analytical data. Note that every morphological solution will fail as no gradient change will be detected due to the same results of rasterization. This gets more problematic with shorter feature search distance.

Such methods provide temporally stable edge antialiasing, as the blend factor relies on continuous triangle information rather than discrete rasterization results (see Figure 3.3).

Gradient length is limited only by storage. In practice, it is enough to store additional data in 8 bits: 1 bit for the major axis and 7 bits for signed distance, providing 64 effective gradient steps.

This algorithm also deals efficiently with alpha-tested silhouettes, if a meaningful distance to an edge can be estimated. This proves to be relatively easy with nonbinary alpha channels. Alpha test derivatives can be used to estimate the distance to a cutout edge. A better solution would be to use signed distance fields for alpha testing and directly output the real distance to the edge.

Both methods are fast and easy to implement in practice. It is worth noting that the final distance to the edge should be the minimum of the geometric distance to the triangle edge and the edge derived from the alpha channel.

We implemented GBAA and optimized it to take advantage of the benefits of the new hardware features found in modern GPUs. AMD's GCN architecture allows pixel shaders to sample vertex data from the triangle descriptors used to rasterize the current pixel. This means that we no longer need the expensive geometry-shader stage to access vertex data in order to calculate the distance to the edge as shown by [Drobot 14].

The snippet in Listing 3.1 shows the improved GBAA algorithm with offsets directly evaluated in the pixel shader. The final postprocess resolve step remains unchanged from the original algorithm.

```
// Calculate closest axis distance between point X
// and line AB. Check against known distance and direction
float ComputeAxisClosestDist(float2 inX,
float2 inA,
float2 inB,
inout uint     ioMajorDir,
inout float    ioAxisDist)
{
float2 AB         = normalize(inB - inA);
float2 normalAB = float2(-AB.y, AB.x);
float dist        = dot(inA, normalAB) - dot(inX, normalAB);
bool majorDir     = (abs(normalAB.x) > abs(normalAB.y));
float axisDist    = dist * rcp(majorDir? normalAB.x : normalAB.y)↩
    ;

if(axisDist < ioAxisDist) ioAxisDist = axisDist;
if(axisDist < ioAxisDist) ioMajorDir = majorDir;
}

void GetGeometricDistance(float2 inScreenCoord,
out float oDistance,
out bool oMajorDir)
   {
   // GetParameterX are HW implementation dependant
float2 sc = GetParameterInterpolated( inScreenCoord );
   float2 sc0 = GetParameterP0( inScreenCoord )
   float2 sc1 = GetParameterP1( inScreenCoord );
   float2 sc2 = GetParameterP2( inScreenCoord );
oDistance = FLT_MAX;

ComputeAxisClosestDist(sc, sc0, sc1, oMajorDir, oDistance);
ComputeAxisClosestDist(sc, sc1, sc2, oMajorDir, oDistance);
ComputeAxisClosestDist(sc, sc2, sc0, oMajorDir, oDistance);
}

// inAlpha is result of AlphaTest,
// i.e., Alpha - AlphaRef
// We assume alpha is a distance field
void GetSignedDistanceFromAlpha(float inAlpha,
out float oDistance,
out bool oGradientDir)
{
   // Find alpha test gradient
float xGradient = ddx_fine(inAlpha);
float yGradient = ddy_fine(inAlpha);
oGradientDir = abs(xGradient) > abs(yGradient);
// Compute signed distance to where alpha reaches zero
oDistance = -inAlpha * rcp(oGradientDir ? xGradient : yGradient);
```

```
    }

  void GetAnalyticalDistanceToEdge(float inAlpha,
  float2 inScreenCoord,
  out float oDistance,
  out bool oMajorDir )
  {
  bool alphaMajorAxis; float alphaDistance;
  GetSignedDistanceFromAlpha(inAlpha,
  alphaDistance,
  alphaMajorAxis)
  GetGeometricDistance(inScreenCoord,
  oDistance,
  oMajorDir);
  if(alphaDistance < oDistance) oDistance = alphaDistance;
  if(alphaDistance < oDistance) alphaMajorAxis = alphaMajorAxis;
 }
```

Listing 3.1. Optimized GBAA distance to edge shader. This uses direct access to vertex data from within the pixel shader.

In terms of quality, the analytical methods beat any morphological approach. Unfortunately, this method proves to be very problematic in many real-world scenarios. Malan developed a very similar antialiasing solution and researched further into the practical issues [Malan 10].

The main problem stems from subpixel triangles, which are unavoidable in a real game production environment. If an actual silhouette edge is composed of multiple small or thin triangles, then only one of them will get rasterized per pixel. Therefore, its distance to the edge might not be the actual distance to the silhouette that we want to antialias. In this case, the resulting artifact will show up as several improperly smoothed pixels on an otherwise antialiased edge, which tends to be very visually distracting (see Figure 3.4 and Figure 3.5).

Malan proposed several ways of dealing with this problem [Malan 10]. However, none of these solutions are very practical if not introduced at the very beginning of the project, due to complex mesh processing and manual tweaking.

Another issue comes again from the actual data source. Hints for antialiasing come from a single triangle, therefore it is impossible to correctly detect and process intersections between triangles. Many assets in a real production scenario have intersecting triangles (i.e., a statue put into the ground will have side triangles intersecting with the terrain mesh). GPU rasterization solves intersections by depth testing before and after rendering a triangle's pixels. Therefore, there is no analytical information about the edge created due to intersection. In effect, the distance to the closest edge does not represent the distance to the intersection edge, which results in a lack of antialiasing.

3.5.2 Coverage Reconstruction Antialiasing (CRAA)

In order to improve upon the techniques and results shared in Section 3.5.1, we would like to find a way to determine more information about a triangle's actual

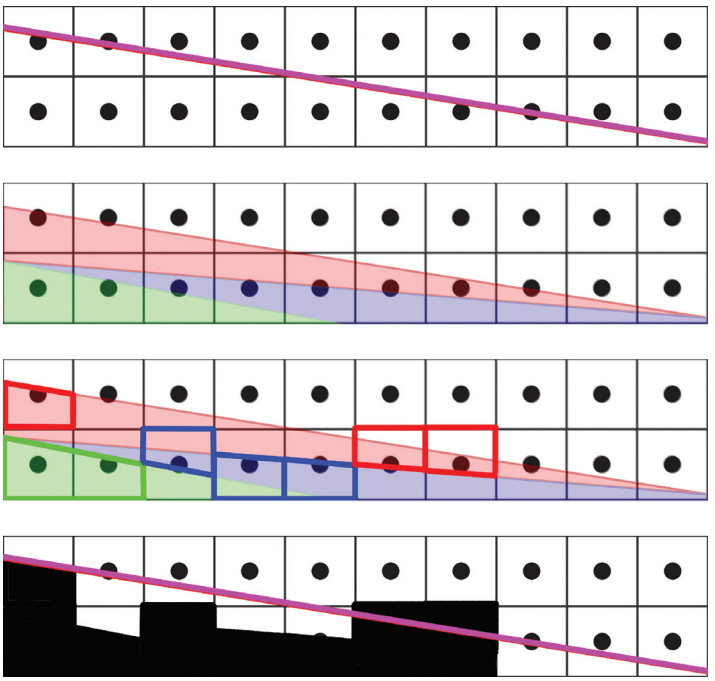

Figure 3.4. False distance to a silhouette edge due to subpixel triangles. Taking a single triangle into account would result in rerasterization of a false edge (blue) instead of the real silhouette edge (red).

Figure 3.5. Top to bottom: a visualization of analytical distance to edge, rasterized edge, analytically antialiased edge, an edge using 5-bit gradients, and an edge showing minor artifacts when multiple triangles intersect one pixel.

intersections and edges within a pixel. With this information, we could partially address most of the aforementioned issues. Fortunately, EQAA provides exactly the information we are interested in by using AMD's hardware EQAA.

EQAA overview. The enhanced quality antialiasing (EQAA) framework augments the standard MSAA color/depth resolve with coverage samples. The rasterizer, while processing triangles, can do cheap analytical sample coverage tests within a triangle. The results of such tests are saved into a compressed buffer called a *fragment mask* (FMask). The FMask acts as an indirection table that associates sample locations with color fragments that were rasterized and stored in the fragment buffer (as in normal MSAA). The number of samples can be higher than the number of stored fragments. In order to accomidate this, a sample in FMask may be marked as "unknown" if it is associated with a fragment that cannot be stored in the fragment buffer (see Figure 3.6).

An important aspect of coverage rendering is correct depth testing. Normally, incoming coverage samples need to be tested against depth fragments stored in the MSAA depth buffer. In order to get correct coverage information, normal MSAA would require a depth buffer that stores of the same number of depth fragments as the number of coverage samples (we can get away with storing fewer color fragments because FMask allows us to associate a single fragment with multiple samples). Fortunately, one feature of AMD's GCN architecture is an ability to work with a compressed depth buffer, which is stored as a set of plane equations. When this mode is enabled, EQAA uses these plane equations to do correct depth testing by analytically deriving depth values for all coverage samples. This means that it is possible to get correct coverage information, even if depth and color information is evaluated and stored as a single fragment (thus MSAA is effectively turned off for all render targets).

Another important requirement for correctly resolving coverage is to sort triangles from front to back. Otherwise, due to the rules of rasterization, it is possible for a triangle to partially overlap a pixel and not get rasterized. If that happens, then that pixel's coverage value might not get updated. Therefore, it is essential to sort objects from front to back (which most rendering engines do already). Fragment sorting is mostly taken care of by GCN hardware. Unfortunately, it is still possible to get incorrect results due to subpixel triangles that won't get rasterized and therefore can't correctly update pixel coverage information.

The memory footprint of FMask is directly proportional to number of unique fragments and samples used. For every sample, we need enough bits to index any of the fragment stored for that pixel and also an additional flag for an unknown value. For the sake of this article, we will focus on use cases with one fragment and eight samples (1F8S)—which require 1 bit per sample, thus 8 bits per pixel total (see Figure 3.7). Such a setup proved to be optimal with regard to EQAA performance as well as the FMask's memory footprint.

CRAA setup. Our goal is to use part of the EQAA pipeline to acquire coverage information at a high resolution (8 samples per pixel) without paying the computational and memory overhead of full MSAA rendering. We would like to use

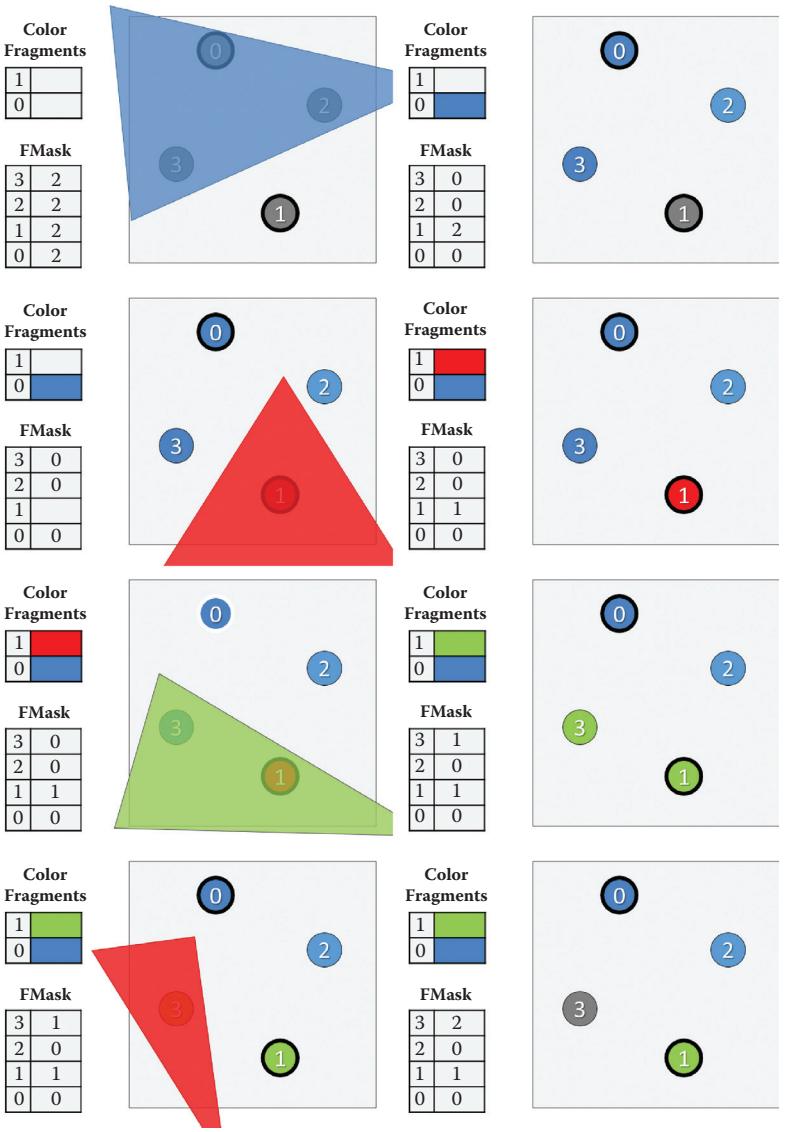

Figure 3.6. The steps here illustrate an updated FMask as new triangles are rasterized. Important note: in the last step, the red triangle does not need to evict Sample 3 if it would fail a Z-test against the sample. (This, however, depends on the particular hardware setup and is beyond the scope of this article.)

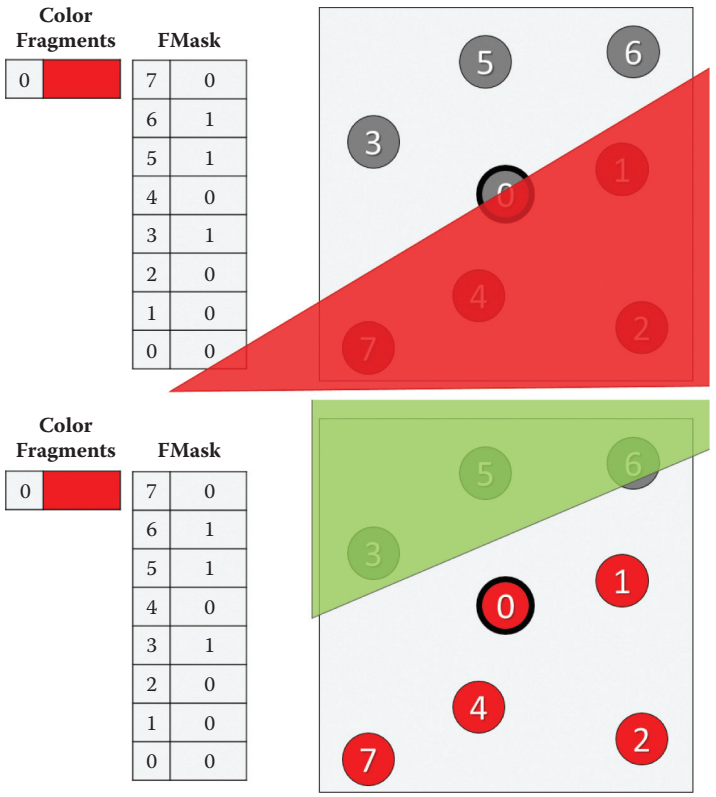

Figure 3.7. Simple rasterization case and corresponding FMask.

information recovered from the coverage data to derive blending hints in a similar fashion to AEAA.

In our simplified case of 1F8S we know that FMask will be an 8-bit value, where the nth bit being set to 0 represents the nth sample being associated with the rasterized fragment (therefore it belongs to the current pixel's triangle and would pass depth testing), while 1 informs us that this sample is unknown—i.e., it was occluded by another triangle.

We can think of FMask as a subset of points that share the same color. If we were to rasterize the current pixel with this newly acquired information, we would need to blend the current pixel's fragment weighted by the number of its known coverage samples, with the other fragment represented by "unknown" coverage samples. Without adding any additional rendering costs, we could infer the unknown color fragments from neighboring pixels. We assume that the depth buffer is working in a compressed mode and that EQAA is using analytical depth testing, thus providing perfectly accurate coverage information.

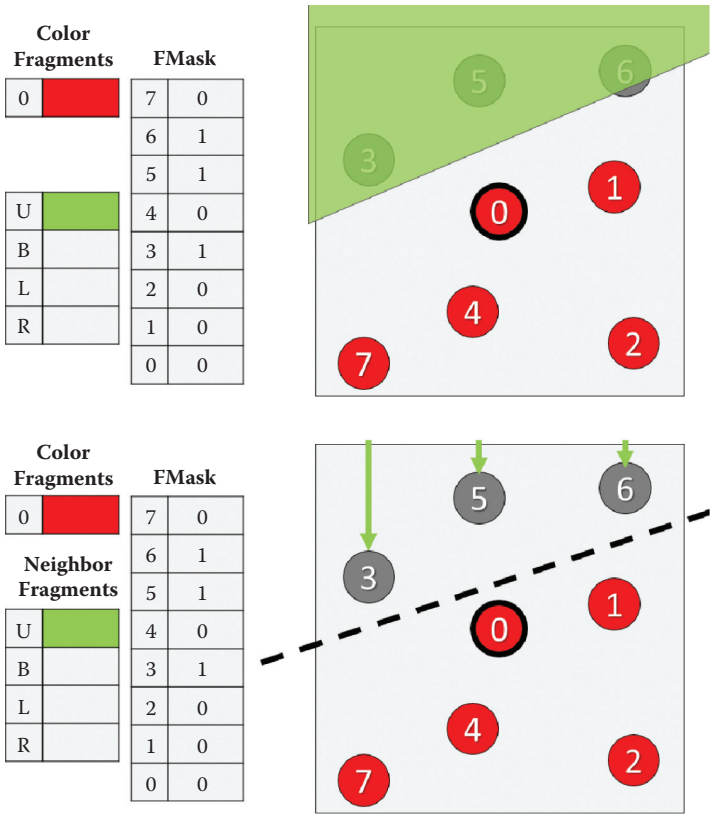

Figure 3.8. Here we illustrate the process of finding an edge that divides a set of samples into "known" and "unknown" samples. Later, this half plane is used to find an appropriate neighboring pixel for deriving the unknown color value.

Single edge scenario. We can apply the same strategy behind AEAA to a simple case in which only a single edge has crossed the pixel. In this case, the pixel's FMask provides a clear division of coverage samples: those that passed will be on one side of the edge, while failed samples will be on the other side. Using a simple line-fitting algorithm, we can find an edge that splits our set of samples into two subsets—passed and failed. This edge approximates the real geometric edge of the triangle that crossed the pixel. In the same spirit of the GBAA algorithm, we find the major axis of the edge as well as its distance from the pixel's center. Then we just need to blend the nearest neighboring pixel color with the current fragment using the edge distance as a weight. Thus, this technique infers the unknown samples from the pixel closest to the derived half plane (see Figure 3.8).

```
float4 CRAA( Texture2DMS<float4> inColor,
             Texture2D<uint2> inFMask,
             uint2 inTexcord )
{
  // Read FMask / HW dependant
  uint iFMask = inFMask.Load( uint3( viTexcoord, 0 ) );
  uint unknownCov = 0;
  float2 hP = 0.0;

  // Average all directions to unknown samples
  // to approximate edge halfplane
  for( uint iSample = 0; iSample < NUM_SAMPLES; ++iSample )
    if( getFMaskValueForSample(iFMask, iSample) == UNKNOWN_CODE )
    {
       hP += TexColorMS.GetSamplePosition(iSample);
       unknownCoverage++;
    }

  // Find fragment offset to pixel on the other side of edge
  int2 fOff = int2( 1, 0);
  if(abs(hP.x) > abs(hP.y) && hP.x <= 0.0) fOff = int2(-1, 0);
  if(abs(hP.x) <= abs(hP.y) && hP.x > 0.0) fOff = int2( 0, 1);
  if(abs(hP.x) <= abs(hP.y) && hP.x <= 0.0) fOff = int2( 0,-1);

  // Blend in inferred sample
  float knownCov = NUM_SAMPLES - unknownCoverage;
  float4 color = inColor.Load( viTexcoord, 0 ) * knownCov;
  color += inColor.Load( viTexcoord + fOff, 0 ) * unknownCov;
  return color /= NUM_SAMPLES;
}
```

Listing 3.2. A simple shader for finding the half plane that approximates the orientation of the "unknown" subset of samples. The half plane is then used to find the closest pixel on the other side of the edge in order to infer the unknown sample's color.

The resolve we have described is akin to GBAA with a limited number of gradient steps (the number of steps is equal to the number of samples used by EQAA). An important thing to note is that our resolve does not need to know anything about the neighboring geometric data (all the information that is needed for reconstruction is contained within the pixel). This is an important difference, because we can reconstruct an edge that was created as the result of rasterizating multiple overlapping triangles; GBAA can only recreate the edge of a single triangle.

Our resolve can be efficiently implemented at runtime by approximating the half plane (see Listing 3.2) while still providing quality comparable to MSAA with the same sampling ratio (see Figure 3.9).

Complex scenario. Following what we learned about resolving simple edges using FMask, we would now like to apply similar ideas to resolving more complex situations in which multiple edges cross a given pixel. In order to achieve this, we would like to be able to group together "failed" samples from different triangles

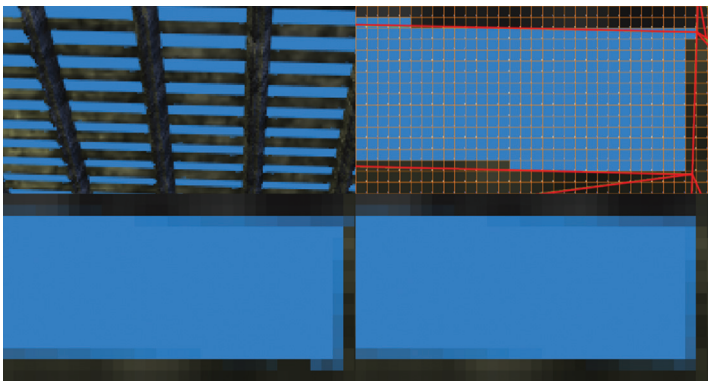

Figure 3.9. A challenging rendering scenario for antialiasing (top left). Rasterization grid and edge layout (top right). Simple 8×CRAA resulting in edge antialiasing comparable to 8×MSAA apart from pixels that are intersected by multiple triangles (bottom left). The results of 8×MSAA (bottom right).

into multiple disconnected sets. For every disconnected set, we find edges (up to two edges in our implementation) that split it off from other sets. Then we use the acquired edges to find major directions that should be used for subset blending. For every subset of unknown fragments, we blend in a color from the neighboring fragment associated with that subset and weighted by the subset's area coverage within the pixel. Finally, we sum all the color values for each subset and blend this with the current fragment's known color weighted by the percentage of passing coverage samples. This way, we can partially reconstruct the subpixel data using the current pixel's surrounding neighborhood (see Figure 3.10).

Using precomputed LUT. Clearly, our algorithm could be rewritten to provide a set of weights for blending colors from the surrounding 3×3 pixel neighborhood. The blend weights rely only on the data present in FMask, and thus our blend weights can be precomputed and stored in a look up table (LUT), which is indexed directly by a pixel's FMask bit pattern.

In our 1F8S case, the LUT would only need 256 entries. Our implementation uses only top, bottom, left, and right neighboring pixels and uses only 4-bit gradients or blend weights. Therefore, the whole LUT requires $256 \times 4 \times 4 = 512$-byte array, which easily fits entirely within the cache.

We also experimented with more complex LUT creation logic. FMask can be evaluated in a morphological fashion to distinguish shapes, thus calculating more accurate and visually plausible gradients. Unfortunately, due to our project's time constraints, we did not have time to properly pursue this direction of research. We believe that a significant quality improvement can be gained from smarter

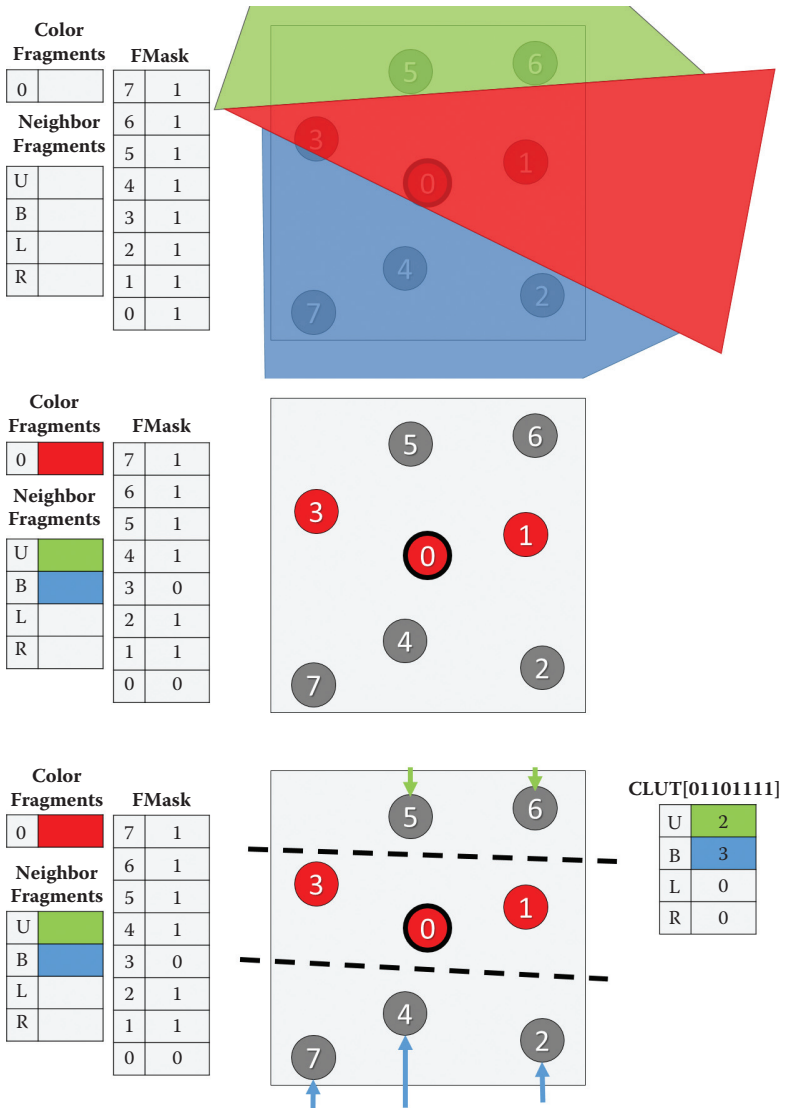

Figure 3.10. One of the possible methods for finding blend weights for sample subsets. The bottom image illustrates a blend weight resolve using a lookup table.

FMask analysis. Using 1F16S would also provide significantly better precision and subpixel handling.

It is worth noting that even the simple logic presented in this section allows for significant aliasing artifact reduction on thin triangles. Figure 3.11 illustrates a

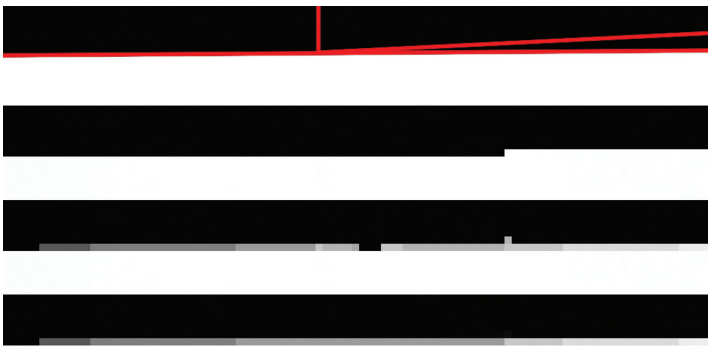

Figure 3.11. Top to bottom: edge layout, rasterized edge, simple 8×CRAA resolve, and 8×CRAA LUT correctly resolving subpixel artifacts.

problematic case for AEAA, where our simple CRAA resolve correctly antialiased the edge. Unfortunately, when there are too many subpixel triangles that don't pass rasterization, CRAA may also fail due to incorrect coverage information. In practice, this heavily depends on the exact rendering situation, and still CRAA has much more relaxed restrictions than AEAA.

The code snippet in Listing 3.3 illustrates the CRAA LUT resolve properly resolving minor subpixel details (see Figure 3.10).

```
float4 CRAA_LUT( Texture2DMS<float4> inColor,
                 Texture2D<uint2> inFMask,
                 Texture1D<uint> inCRAALUT,
                 uint2 inTexcord )
{
  // Read FMask / HW dependant
  uint iFMask      = inFMask.Load( uint3( viTexcoord, 0 ) );
  uint LUTREsult = inCRAALUT[iFMask];
  float wC, wN, wE, wS, wW;

  // LUT is packed as 8bit integer weights
  // North 8b | West 8b | South 8b | East 8b
  // Can also pack whole neighborhood weights in 4 bits
  ExctractLUTWeights(LUTResult, wC, wN, wE, wS, wW);

  float4 color = inColor.Load( viTexcoord + int2( 0, 0) * wC;
         color += inColor.Load( viTexcoord + int2( 0,-1) * wN;
         color += inColor.Load( viTexcoord + int2( 0, 1) * wE;
         color += inColor.Load( viTexcoord + int2( 0, 1) * wS;
         color += inColor.Load( viTexcoord + int2(-1, 0) * wW;
         return color;
}
```

Listing 3.3. CRAA LUT implementation.

3.6 Temporal Super-Sampling

3.6.1 Overview

Temporal super-sampling can be succinctly described by the following simple algorithm:

1. Every frame, offset the projection matrix by a subpixel offset.
2. Use the current frame's N motion vectors to reproject data from frame $N - k$ to time step N.
3. Test if the reprojected data is valid:
   - No occlusion occurred.
   - The data comes from the same source (i.e., object or triangle).
   - The data is not stale (i.e., due to lighting changes).
4. Accumulate data from frame N with data reprojected from frame $N - k$.
5. Repeat steps 2–4 for k frames back in time.

The number k dictates the number of unique subpixel offsets used for jitter in order to get spatially unbiased results after converging by accumulation of all the k samples. However, it is easy to see that by increasing the number k of frames of history, the algorithm has a much higher complexity and therefore a much higher chance of failure.

The most proper (and expensive) approach would be to hold the last k frames in memory along with their motion vectors and additional data required to verify sample validity. Then we would need to evaluate a series of dependent reads and checks to verify if reprojected pixels were valid back in both the spatial and temporal domains. For a given pixel, we could only accumulate as many samples as managed to pass these checks until we encountered the first failure. This approach would guarantee a very high-quality result [Yang et al. 09], however the cost when $k > 1$ is prohibitively expensive for real-time, performance-oriented scenarios, such as games.

Other solutions rely on a so-called *history buffer* that holds all the accumulated samples, thus simplifying the previously described method to the $k = 1$ case. Unfortunately, this solution can't guarantee convergence as it is impossible to remove stale samples from the history buffer without discarding it totally. Also, the validation functions need to be much more conservative in order to prevent incoherent samples from entering the history buffer. This approach, when used for super-sampling, results in a somewhat unstable image with possible "fizzing" high-contrast pixels since they can't converge [Malan 12]. Very similar approaches can be used to stabilize the image over time (instead of super-sampling), as shown in [Sousa 13] and [Valient 14]. We will be discussing this more in Section 3.7.

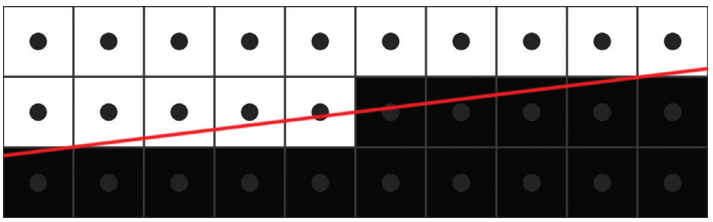

Figure 3.12. One sample centroid rasterization pattern.

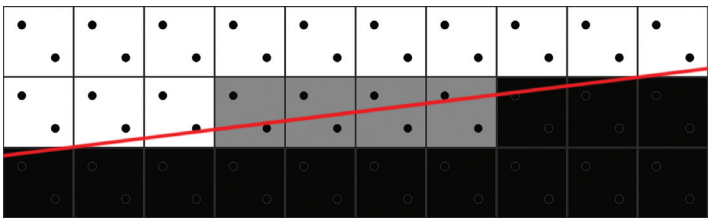

Figure 3.13. Two rotated samples from a standard 2×MSAA pattern.

Taking all pros and cons into account, we decided to pursue the highest possible quality with $k = 1$. This means that we are only dealing with one frame of history, and for every single frame we have two unique samples at our disposal (assuming that our history sample was accepted as valid); we would like to get as much data from them as possible.

3.6.2 Sampling Patterns

A single sample resolve can't provide any gradients (see Figure 3.12 for a baseline reference).

As discussed in Section 3.6.1, we want to maximize the resolve quality while using only two unique samples. One possible way to achieve this is through a more complex resolve and sampling pattern. Currently, common implementations of 2× super-sampling use the 2×MSAA sampling pattern (see Figure 3.13).

One possible improvement upon this is to use a quincunx sampling pattern [NVIDIA 01]. This pattern relies on sharing a noncentral sample with adjacent pixels; thus, the resolve kernel needs to sample corner data from neighboring pixels (see Figure 3.14).

In practice, quincunx sample positions are not very beneficial. Having sampling points on regular pixel rows and columns minimizes the number of potential edges that can be caught. In general, a good pattern should be optimized for maximum pixel row and column coverage. The 4× rotated-grid pattern is a good example (see Figure 3.15).

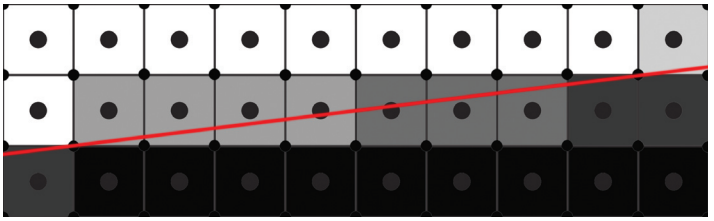

Figure 3.14. Quincunx sampling and resolve pattern guarantees higher-quality results than 2×MSAA while still keeping the sample count at 2.

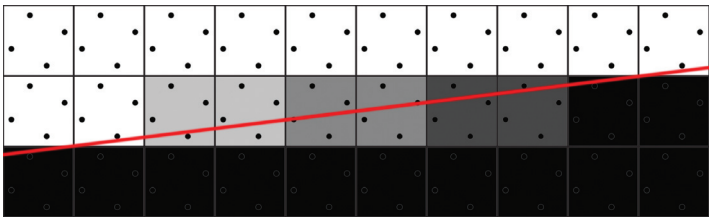

Figure 3.15. The 4× rotated-grid super-sampling pattern maximizes row and column coverage.

3.6.3 FLIPQUAD

[Akenine-Möller 03] proposed several other low-sample-cost patterns such as FLIPTRI and FLIPQUAD. We will focus on FLIPQUAD as it perfectly matches our goal of using just two unique samples. This sampling pattern is similar to quincunx in its reuse of samples between pixels. However, a massive quality improvement comes from putting sampling points on pixel edges in a fashion similar to the rotated-grid sampling patterns. This provides unique rows and columns for each sample, therefore guaranteeing the maximum possible quality.

The FLIPQUAD pattern requires a custom per-pixel resolve kernel as well as custom per-pixel sampling positions (see Figure 3.16). An important observation is that the pattern is mirrored, therefore every single pixel quad is actually the same.

The article [Laine and Aila 06] introduced a unified metric for sampling pattern evaluation and proved FLIPQUAD to be superior to quincunx, even surpassing the 4× rotated-grid pattern when dealing with geometric edges (see Figure 3.17 and Table 3.1).

We can clearly see that the resolve kernel is possible in a typical pixel shader. However, the per-pixel sampling offsets within a quad were not supported in hardware until modern AMD graphic cards exposed the EQAA rasterization pipeline extensions. This feature is exposed on Xbox One and Playstation 4, as well as through an OpenGL extension on PC [Alnasser 11].

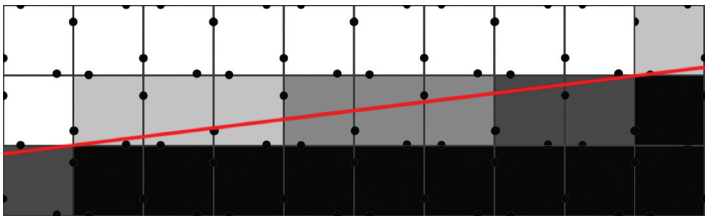

Figure 3.16. FLIPQUAD provides optimal usage of two samples matching quality of 4× rotated-grid resolve.

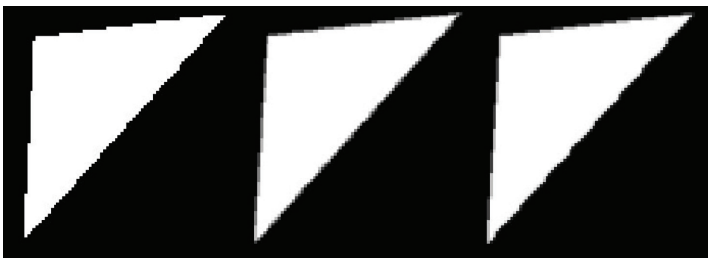

Figure 3.17. Left to right: single sample, FLIPQUAD, and quincunx. [Image courtesy [Akenine 03].]

The implementation of the FLIPQUAD pattern is fairly straightforward using 2×MSAA. The snippets in Listings 3.4 and 3.5 give sampling positions for pixels within a quad and the reconstruction kernel.

3.6.4 Temporal FLIPQUAD

In Section 3.6.3 we discussed implementing FLIPQUAD sampling and reconstruction on modern hardware. However, in the context of temporal super-sampling, we need to adapt our pattern and resolve kernel. We decided to split the pattern into two subsets—one that will be used to render even frames (blue) and one used on odd frames (red) (see Figure 3.18).

Pattern	E
1× Centroid	> 1.0
2×2 Uniform Grid	0.698
2×2 Rotated Grid	0.439
Quincunx	0.518
FLIPQUAD	0.364

Table 3.1. Error metric (E) comparison against a 1024-sample reference image as reported by [Laine and Aila 06] (lower is better).

```
// Indexed [SAMPLES LOCATIONS] in n/16 pixel offsets
int2 gFQ_Q00[2] = { int2 (-8,-2), int2 ( 2,-8) };
int2 gFQ_Q10[2] = { int2 (-8, 2), int2 (-2,-8) };
int2 gFQ_Q01[2] = { int2 (-8, 2), int2 (-2,-8) };
int2 gFQ_Q11[2] = { int2 (-8,-2), int2 ( 2,-8) };
```

Listing 3.4. FLIPQUAD sample array.

```
s0 = CurrentFrameMS.Sample(PointSampler, UV, 0);
s1 = CurrentFrameMS.Sample(PointSampler, UV, 1);
s2 = CurrentFrameMS.Sample(PointSampler, int2( 1, 0), 0);
s3 = CurrentFrameMS.Sample(PointSampler, int2( 0, 1), 1);

return 0.25 * (s0 + s1 + s2 + s3);}
```

Listing 3.5. FLIPQUAD reconstruction kernel.

To resolve this sampling scheme properly, we need two resolve kernels—one for even and one for odd frames (see Listings 3.6 and 3.7). Due to the alternating patterns, in each frame the kernel will be guaranteed to properly resolve horizontal or vertical edges. If data from the previous frame is accepted, a full pattern will be properly reconstructed.

It is worth noting that an incorrect (or missing) FLIPQUAD reconstruction pass will result in jigsaw edges, which are a direct result of a nonuniform sampling grid (see Figure 3.19).

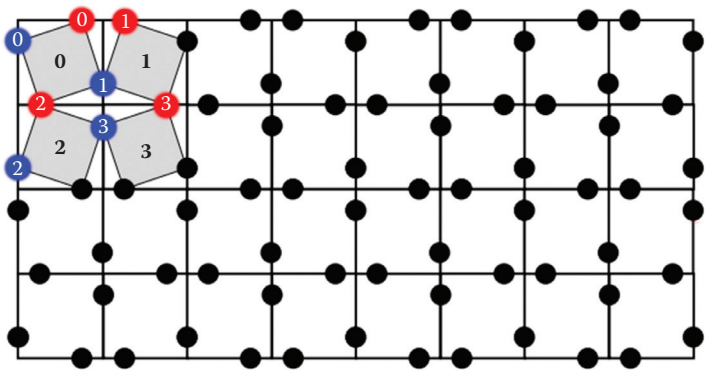

Figure 3.18. Temporal FLIPQUAD pattern. Red samples are rendered on even frames. Blue samples are rendered on odd frames.

```
// Indexed[FRAME][SAMPLES LOCATIONS] in n/16 pixel offsets
// BLUE RED
int2 gTempFQ_Q0[2][1]   = { int2 (-8,-2), int2 ( 2,-8) };
int2 gTempFQ_Q1[2][1]   = { int2 (-8, 2), int2 (-2,-8) };
int2 gTempFQ_Q2[2][1]   = { int2 (-8, 2), int2 (-2,-8) };
int2 gTempFQ_Q3[2][1]   = { int2 (-8,-2), int2 ( 2,-8) };
```

Listing 3.6. Temporal FLIPQUAD sample array.

```
#if defined(ODD_FRAME) // RED
// Horizontal pattern for frame [1]
int2 offset0 = int2(0, 1);
int2 offset1 = int2(1, 0);
#else if defined(EVEN_FRAME) // BLUE
int2 offset0 = int2(1, 0);
int2 offset1 = int2(0, 1);
#endif

s0 = CurrentFrame.Sample(PointSampler, UV);
s1 = CurrentFrame.Sample(PointSampler, UV, offset0);
s2 = PreviousFrame.Sample(LinearSampler, previousUV);
s3 = PreviousFrame.Sample(LinearSampler, previousUV, offset1);

return 0.25 * (s0 + s1 + s2 + s3);}
```

Listing 3.7. Temporal FLIPQUAD reconstruction kernel.

Rasterization should happen at sample locations in order to take full advantage of FLIPQUAD. This can be easily achieved by using the `sample` prefix in HLSL's interpolator definition. This way, texture data will be offset properly, resulting in a correctly super-sampled image.

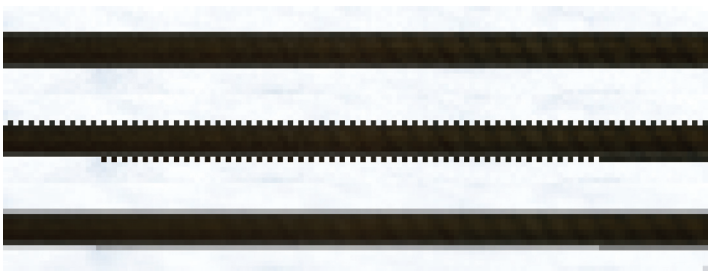

Figure 3.19. Top to bottom: edge rasterized on an even frame and then an odd frame and the final edge after temporal FLIPQUAD reconstruction kernel.

```
// Quad defined as (sample positions within quad)
// s00 s10
// s01 s11
DDX[f(s00)] = [f(s00) -- f(s10)]/dx, dx = |s00 -- s10|
DDY[f(s00)] = [f(s00) -- f(s01)]/dy, dy = |s00 -- s01|

// Hardware assumes dx = dy = 1
// In case of sampling pattern from Listing 6. dx != dy
// Footprint-based sampling picks base mip level
// Based on max(ddx, ddy)
// Frame A max(ddx, ddy) != Frame B max(ddx, ddy)
// Implies non temporarily coherent mip selection

// Calculated in 1/16th of pixel
// Frame A (BLUE)
dx = |-8 -- (16 + (-8))| = 16
dy = |-2 -- (16 + ( 2))| = 20
baseMip ~ max(dx, dy) = 20

// Frame B (RED)
dx = | 2 -- (16 + (-2))| = 12
dy = |-8 -- (16 + (-8))| = 16
baseMip ~ max(dx, dy) = 16}
```

Listing 3.8. Default method for derivative calculation.

3.6.5 Temporal FLIPQUAD and Gradients

One side effect of using the temporal FLIPQUAD pattern is a nonuniform distance between samples within a quad. This causes problems for the gradient calculation and mipmap selection. Graphics cards rely on calculating per-pixel (or per-quad) derivatives using differentials within a quad. This process is fairly straightforward (see Listing 3.8).

As an optimization, spatial distances, used to normalize the differential, are assumed to be 1. However, if we look at our temporal FLIPQUAD pattern, we clearly see that distances between samples are different between the x- and y-axes, and we alternate from frame to frame (see Listing 3.8).

Nonuniform distances will result in a biased mipmap level-of-detail calculation, as $ddx(uv)$ or $ddy(uv)$ will be increasing faster than it should. In effect, the textures will appear sharper or blurrier than they should be. In the worst case, a single texture can select different mipmap levels, under the same viewing direction, when rendering even and odd frames. This would lead to temporal instability since bilinear filtering picks the mipmap based on $\max(ddx, ddy)$, which, in this case, would result in differences between frames (see Figure 3.20).

One way to solve this issue would be to switch all texture samples to a gradient-based texture read (i.e., `tex2Dgrad` in HLSL) and to calculate the gradients analytically taking sample distance into account. Unfortunately, this complicates all shaders and can have significant performance overhead.

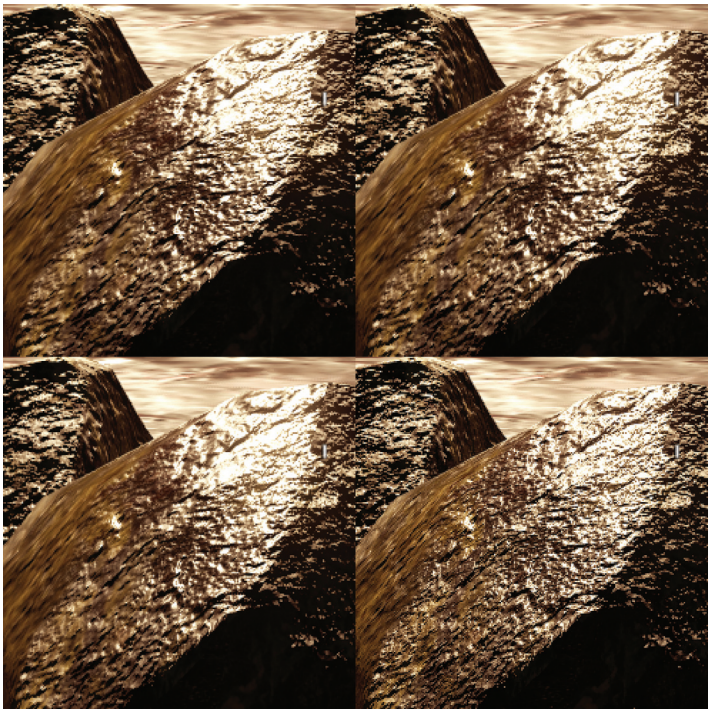

Figure 3.20. The top row shows even and odd frames of the reordered Temporal FLIPQUAD pattern. The bottom row shows the default temporal FLIPQUAD pattern clearly suffering from mipmap level mismatches. (The bottom right represents an oversharpened odd frame).

Another option is to change the pattern in order to minimize frame-to-frame sample distance variance. While this will not provide correct results, the error may not be noticeable in practice as long as it is temporarily stable (see Figure 3.20). Please note that this also requires different offsets (kernel and projection matrix offsets) to shift samples outside the pixel window (see Listings 3.9 and 3.10 for details).

```
// Indexed[FRAME][SAMPLES LOCATIONS] in n/16 pixel offsets
int2 gTempFQ_Q0[2][1] = { int2 ( 0,-2), int2 (-2, 0) };
int2 gTempFQ_Q1[2][1] = { int2 ( 0, 2), int2 ( 2, 0) };
int2 gTempFQ_Q2[2][1] = { int2 ( 0, 2), int2 ( 2, 0) };
int2 gTempFQ_Q3[2][1] = { int2 ( 0,-2), int2 (-2, 0) };
int2 gProjMatOff[2][1] = { int2 (-8, 0), int2 ( 0, 8) };}
```

Listing 3.9. Reordered temporal FLIPQUAD with additional projection matrix offsets.

```
#if defined(ODD_FRAME) // RED
// Horizontal pattern for frame [1]
int2 offset0 = int2(0,-1);
int2 offset1 = int2(1, 0);
#else if defined(EVEN_FRAME) // BLUE
int2 offset0 = int2(1, 0);
int2 offset1 = int2(0,-1);
#endif

s0 = CurrentFrame.Sample(PointSampler, UV);
s1 = CurrentFrame.Sample(PointSampler, UV, offset0);
s2 = PreviousFrame.Sample(LinearSampler, previousUV);
s3 = PreviousFrame.Sample(LinearSampler, previousUV, offset1);

return 0.25 * (s0 + s1 + s2 + s3);}
```

Listing 3.10. Reordered temporal FLIPQUAD reconstruction kernel.

3.6.6 History Sample Acceptance Method

Our acceptance method for history samples is based on the algorithm used in *Killzone: Shadow Fall* [Valient 14].

The history sample from frame $N-1$ is valid only if

- the motion flow between frame N and $N-1$ is coherent,
- the color flow between frames N and $N-2$ is coherent. (Note that $N-2$ and N have the same subpixel jitter.)

The first constraint guarantees that a sample was not occluded and was moving in a similar direction. The second constraint guarantees that there was no major change in lighting conditions between frames with the same subpixel jitter. Both tests use a 3×3 neighborhood using the sum of absolute differences to estimate the degree of similarity between frames. It is possible to achieve reasonable results using a smaller neighborhood, however, testing might need to be more conservative.

If any constraint fails, then we fall back to clamping history samples to the current frame color bounding box, as described in Section 3.7. This guarantees no ghosting and enhanced temporal stability. It is worth noting that the color flow constraint is very important in a real production environment. It enables the unrestricted usage of animated textures and particle effects as well as lighting changes. Another important benefit is that it grants convergence in the event that the viewport becomes fixed.

3.6.7 Resampling

Any reprojection method is prone to numerical diffusion errors. When a frame is reprojected using motion vectors and newly acquired sampling coordinates do

not land exactly on a pixel, a resampling scheme must be used. Typically, most methods resort to simple bilinear sampling. However, bilinear sampling will result in over-smoothing. If we would like to use a history buffer in order to accumulate multiple samples, we will also accumulate resampling errors, which can lead to serious image quality degradation (see Figure 3.22). Fortunately, this problem is very similar to well-researched fluid simulation advection optimization problems.

In fluid simulation, the advection step is very similar to our problem of image reprojection. A data field of certain quantities (i.e., pressure and temperature) has to be advected forward in time by a motion field. In practice, both fields are stored in discretized forms; thus, the advection step needs to use resampling. Assuming that the operation is a linear transform, this situation is equal to the problem of reprojection.

Under these circumstances, a typical semi-Lagrangian advection step would be equal to reprojection using bilinear resampling. A well-known method to prevent over-smoothing is to use second order methods for advection. There are several known methods to optimize this process, assuming that the advection operator is reversible. One of them is the MacCormack scheme and its derivation: back and forth error compensation and correction (BFECC). This method enables one to closely approximate the second order accuracy using only two semi-Lagrangian steps [Dupont and Liu 03].

BFECC is very intuitive. In short, we advect the solution forward and backward in time using advection operator A and its reverse, A^R. Operator error is estimated by comparing the original value against the newly acquired one. The original value is corrected by error ($\frac{\varphi^n - \widehat{\varphi}^n}{2}$) and finally advected forward into the next step of the solution (see Algorithm 3.1 and Figure 3.21 for an illustration).

In the context of reprojection, our advection operator is simply a bilinear sample using a motion vector offset. It is worth noting that the function described by the motion vector texture is not reversible (i.e., multiple pixels might move to same discretized position).

A correct way to acquire a reversible motion vector offset would be through a depth-buffer–based reprojection using an inverse camera matrix. Unfortunately, this would limit the operator to pixels subject to camera motion only. Also, the operator would be invalid on pixels that were occluded during the previous time step.

1: $\widehat{\varphi}^{n+1} = A(\varphi^n)$.
2: $\widehat{\varphi}^n = A^R(\widehat{\varphi}^{n+1})$.
3: $\bar{\varphi} = \varphi^n + \dfrac{\varphi^n - \widehat{\varphi}^n}{2}$.
4: $\varphi^{n+1} = A(\bar{\varphi})$.

Algorithm 3.1. Original BFECC method.

1: $\widehat{\varphi}^{n+1} = A(\varphi^n)$.
2: $\widehat{\varphi}^{n} = A^R(.\widehat{\varphi}^{n+1})$.
3: $\varphi^{n+1} = \widehat{\varphi}^{n+1} + \frac{\varphi^n - \widehat{\varphi}^n}{2}$.

Algorithm 3.2. Simplified BFECC method.

Another option is to assume a high-coherency motion vector field (texture) and just use a negative motion vector for the reverse operator. However, this approach would break under perspective correction (i.e., high slopes) as well as with complex motion.

In practice, we used a mix of both approaches. The reverse operator is acquired through depth-based reprojection for static pixels and reversing motion vectors for pixels from dynamic objects. For us, this proved to be both efficient and visually plausible (even if not always mathematically correct).

Another important optimization we used was inspired by the simplified BFECC method by [Selle et al. 08]. In this approach, it is proven that the error is not time dependent; therefore the results from frame $n + 1$ can be directly compensated by an error estimate. This simplifies the original BFECC by one full semi-Lagrangian (see Algorithm 3.2).

Unfortunately, the proposed method requires reading $\widehat{\varphi}^n$, φ^n, and $\widehat{\varphi}^{n+1}$ in order to evaluate φ^{n+1}. However, as we already assumed that the error estimate is time invariant, we can as use the values from step $n + 1$ to estimate the error. Therefore, we can calculate φ^{n+1} using only $\widehat{\varphi}^{n+1}$ and $\widehat{\widehat{\varphi}}^{n+1}$, where $\widehat{\widehat{\varphi}}^{n+1}$ is easy to acquire in a shader (see Algorithm 3.3, Listing 3.11, and Figure 3.21 for details).

One last thing worth mentioning is that BFECC, by default, is not unconditionally stable. There are multiple ways of dealing with this problem, but we found bounding by local minima and maxima to be the most practical [Dupont and Liu 03, Selle et al. 08]. Listing 3.11 presents the simplest implementation of our optimized BFECC, and Figure 3.22 demonstrates the results.

1: $\widehat{\varphi}^{n+1} = A(\varphi^n)$.
2: $\widehat{\varphi}^{n} = A^R(\widehat{\varphi}^{n+1})$.
3: $\widehat{\widehat{\varphi}}^{n+1} = A(\widehat{\varphi}^n)$.
4: $\varphi^{n+1} = \widehat{\varphi}^{n+1} + \frac{\widehat{\varphi}^{n+1} - \widehat{\widehat{\varphi}}^{n+1}}{2}$.

Algorithm 3.3. Shader optimized simplified BFECC method.

```
// Pass outputs phiHatN1Texture
// A() operator uses motion vector texture
void GetPhiHatN1(float2 inUV, int2 inVPOS)
{
  float2 motionVector = MotionVectorsT.Load(int3(inVPOS, 0)).xy;
  float2 forwardProj = inUV + motionVector;
  // Perform advection by operator A()

  return PreviousFrameT.SampleLevel(Linear, forwardProj, 0).rgb;
}

// Pass outputs phiHatTexture
// AR() operator uses negative value from motion vector texture
// phiHatN1 texture is generated by previous pass GetPhiHatN1()
void GetPhiHatN(float2 inUV, int2 inVPOS)
{
  float2 motionVector = MotionVectorsT.Load(int3(inVPOS, 0)).xy;
  float2 backwardProj = inUV - motionVector;

  // Perform reverse advection by operator AR()
  return phiHatN1T.SampleLevel(Linear, backwardProj, 0).rgb;
}

// Final operation to get correctly resampled phiN1
// A() operator uses motion vector texture
// phiHatN1 and phiHatN textures are generated by previous passes
void GetResampledValueBFECC(float2 inUV, int2 inVPOS)
{
  float3 phiHatN1 = phiHatN1T.Load(int3(inVPOS, 0)).rgb;

  // Find local minima and maxima
  float3 minima, maxima;
  GetLimitsRGB(phiHatN1Texture, inUV, minima, maxima);

  float2 motionVector = MotionVectors.Load(int3(inVPOS, 0)).xy;
  float2 A = inUV + motionVector;

  // Perform advection by operator A()
  float3 phiHatHatN1 = phiHatT.SampleLevel(Linear, A, 0).rgb;

  // Perform BFECC
  float3 phiN1         = 1.5 * phiHatN1 - 0.5 * phiHatHatN1;

  // Limit the result to minima and maxima
  phiN1                = clamp(phiN1, minima, maxima);
  return phiN1;
}
```

Listing 3.11. Reordered temporal FLIPQUAD reconstruction kernel.

3.7 Temporal Antialiasing (TAA)

3.7.1 Overview

In Sections 3.5 and 3.6, we presented methods for improving spatial and temporal stability and resolution of antialiased images. However, even when using 4× rotated-grid super-sampling and dedicated edge antialiasing, disturbing temporal

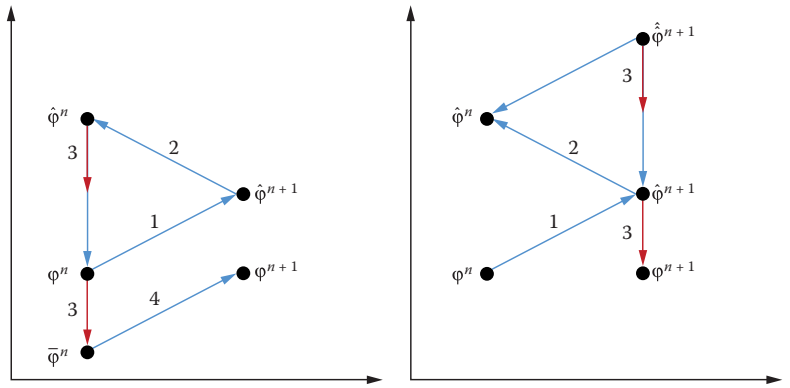

Figure 3.21. Conceptual scheme of the original BFCE method (left) and of the shader optimized BFCE used for texture resampling (right).

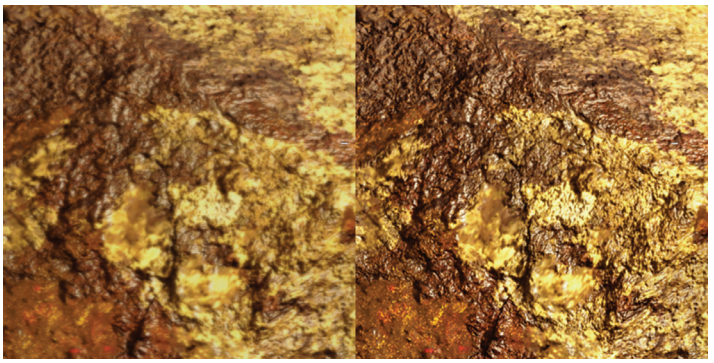

Figure 3.22. Continuous resampling of 30 frames using a history buffer. The camera is in motion, panning from left to right. Using bilinear sampling shows numerical diffusion errors resulting in a blurry image (left). Using optimized linear BFCE helps to minimizes blurring (right).

artifacts may occur. Ideally we would like to accumulate more frames over time to further improve image quality. Unfortunately, as described in Sections 3.6.1 and 3.6.6, it is very hard to provide a robust method that will work in real-world situations, while also using multiple history samples, without other artifacts. Therefore, several methods rely on super-sampling only in certain local contrast regions of an image [Malan 12, Sousa 13, Valient 14]. These approaches rely on visually plausible temporal stabilization (rather than super-sampling). We would like to build upon these approaches to further improve our results.

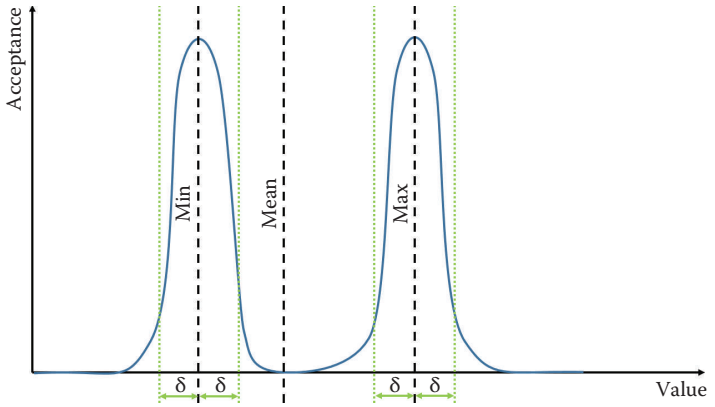

Figure 3.23. Frequency-based acceptance function plot.

3.7.2 Frequency-Based Acceptance Metric

A simple temporal antialiasing scheme can be described as a cumulative blend between the current frame and an accumulation buffer of previous frames (history data). However, finding the correct blend factor is difficult. We would like to stabilize the data such that we avoid fluctuations over time and enhance the image quality by accumulating new information. On the other hand, we need to detect when data becomes stale or invalid.

We build a color bounding box of fresh data by sampling the 3×3 spatial neighborhood from the current frame. From this data we evaluate the local minima, maxima, and mean value per RGB channel.

We follow a reasoning grounded in signal frequency analysis. If our history data is too similar to the local mean, it does not bring much new information to the current frame and might even diffuse the result if it contains accumulated errors (i.e., due to sampling errors). The more "different" the data is from mean, the more important it is. However, certain information might just be a fluctuation that skews the data. With all of this in mind, we can treat information that is in close proximity to fresh data's local minima and maxima as valid (to a degree). Therefore, we could plot our function of history data acceptance as two peaks centered at a local minima and maxima, with a slope curve steered by a user controlled δ value (see Figure 3.23).

To reduce incorrectly accepted samples even further, we utilize motion-based constraints as described in Section 3.6.6. This combination method minimizes the possible reprojection artifacts to a 3×3 pixel neighborhood (i.e., ghosting) while still guaranteeing a very high level of temporal stabilization. It's worth noting that this method can't provide convergence in the context of super-sampling (or jittered rendering), as sample acceptance relies on local data changes. See Listing 3.12 for details.

```
// curMin, curMax, curMean are estimated from 3x3 neighborhood
float3 getTAA(float2 inCurtMotionVec,
              float2 inPrevMotionVec,
              float3 inCurtMean,
              float3 inCurtMin,
              float3 inCurtMax,
              float3 inCurtValue,
              float3 inPrevValue)
{

  // Motion coherency weight
  float motionDelta = length(inCurMotionVec - inPrevMotionVec);
  float motionCoherence = saturate(c_motionSens * motionDelta));

  // Calculate color window range
  float3 range = inCutMin - inCurMax;

  // Offset the window bounds by delta percentage
  float3 extOffset = c_deltaColorWindowOffset * range;
  float3 extBoxMin = max(inCurMin - extOffset.rgb, 0.0);
  float3 extdBoxMax = inCurMax + extOffset;

  // Calculate deltas between previous and current color window
  float3 valDiff = saturate(extBoxMin - inPrevValue);
  valDiff += saturate(inPreviousValue - extBoxMax);
  float3 clampedPrevVal = clamp(inPrevValue, extBoxMin, extBoxMax);

  // Calculate deltas for current pixel against previous
  float3 meanWeight = abs(inCurValue - inPreValue);
  float loContrast = length(meanWeight)* c_loWeight;
  float hiContrast = length(valDiff) * c_hiWeight;

  // Calculate final weights
  float denom = max((loContrast - hiContrast), 0.0);
  float finalWeight = saturate(rcp(denom + epsilon));

  // Check blend weight against minimum bound
  // Prevents the algorithm from stalling
  // in a 'saddle' due to numerical imprecision
  // Regulates minimum blend of current data
  finalWeight= max(c_minLimiter, w);

  // Correct previous samples according to motion coherency weights
  finalWeight = saturate(finalWeight - motionCoherence);
  // Final value blend
  return lerp(inCurValue, clampedPrevVal, finalWeight);
}
```

Listing 3.12. Temporal antialiasing using our frequency-based acceptance metric.

3.8 Final Implementation

Our final framework's implementation can be split into two main stages:

- temporally stable edge antialiasing, which includes
  - SMAA,

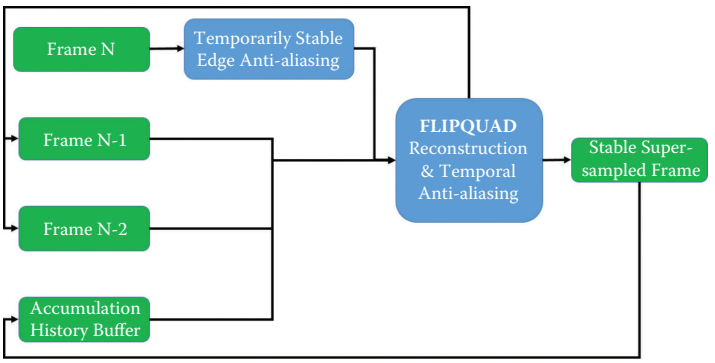

Figure 3.24. Data flow graph in our implementation of the HRAA pipeline.

  - CRAA,
  - AEAA (GBAA);
- Temporal FLIPQUAD reconstruction combined with temporal antialiasing (TAA) (see Listing 3.13).

Figure 3.24 illustrates the data flow inside the framework.

During production, we implemented and optimized all three approaches to temporarily stable edge antialiasing.

SMAA was implemented with geometric edge detection based on depth and normal buffers. Edges were refined by a predicated threshold based on the luminescence contrast. Our edge-detection algorithm choice was dictated by making the resolve as temporally stable as possible.

CRAA and AEAA used the implementations described in Sections 3.5.1 and 3.5.2. Our EQAA setup used a 1F8S configuration, while our AEAA offset buffer was compressed down to 5 bits (utilizing the last remaining space in our tightly packed G-buffer).

The results of either edge antialiasing pass were used as N, $N-1$, and $N-2$ frame sources in the last pass. The history buffer used by TAA at frame N was the output buffer of TAA from frame $N-1$.

3.9 Results Discussion

Our packing/performance scheme resulted in fairly low-quality gradients coming out of edge antialiasing (only 3 bits for steps). However, FLIPQUAD reconstruction provided two to four times the edge gradients virtually for free. In practice, the whole system provided excellent results, matching a 4× rotated-grid super-sampled reference while providing much higher-quality edge antialiasing and temporal stability at minimal cost.

```
// Unoptimized pseudocode for final
// Temporal FLIPQUAD reconstruction & TAA
// Frames N & N-2 are assumed
// To have same jitter offsets
float3 getFLIPQUADTaa()
{
  float3 curMin, currMax, curMean;
  GetLimits(currentValueTexture, curMin, curMax, curMean);

  float3 prevVal = Resample(prevValueTexture);
  float3 prevPrevVal = Resample(prevPrevValueTexture);

  // Get sums of absolute difference
  float3 curSAD = GetSAD(curValueTexture);
  float3 prevPrevSAD = GetSAD(prevPrevValueTexture);

  // Motion coherency weight
  float moCoherence = GetMotionCoherency(curMotionTexture,
                                         prevMotionTexture);

  // Color coherency weight
  float colCoherence = GetColorCoherency(curSAD, prevPrevSAD);

  // FLIPQUAD parts
  float3 FQCurPart = GetCurFLIPQUAD(curValueTexture);
  float3 FQPrevPart = GetPrevFLIPQUAD(prevValueTexture);
  float FQCoherency = motionCoherence + colorCoherence;
  float3 clampFQPrev = clamp(FQPrevPart, curMin, curMax);

  // This lerp allows full convergance
  // If color flow (N-2 to N) is coherent
  FQPrevPart = lerp(FQPrevPart, clampFQPrev, colCoherence);

  // Final reconstruction blend
  float3 FLIPQUAD = lerp(FQCurPart, FQPrevPart,0.5*moCoherence);

  float3 historyVal = Resample(historyValueTexture);

  return getTAA(curMotionTexture, prevMotionTexture,
                curMin, curMax, curMean,
                FLIPQUAD, historyVal)
}
```

Listing 3.13. Pseudocode for the combined temporal FLIPQUAD reconstruction and temporal antialiasing.

While temporal FLIPQUAD and TAA remained stable and reliable components of the framework, the choice of the edge antialiasing solution proved to be problematic.

SMAA provided the most visually plausible results on static pixels under any circumstances. The gradients were always smooth and no edge was left without antialiasing. Unfortunately, it sometimes produced distracting gradient wobble while in motion. The wobble was partially mitigated by the FLIPQUAD and TAA resolves. Unfortunately, SMAA had the highest runtime cost out of the whole framework.

AEAA provided excellent stability and quality, even in close-ups where triangles are very large on screen. Unfortunately, objects with very high levels of tessellation resulted in very objectionable visual noise or even a total loss of antialiasing on some edges. Even though this was the fastest method for edge antialiasing, it proved too unreliable for our open world game. It is worth noting that our AEAA implementation required us to modify every single shader that writes out to the G-buffer. This might be prohibitively expensive in terms of developer maintainability and runtime performance.

CRAA mitigated most of the issues seen with AEAA and was also the easiest technique to implement. Unfortunately, on the current generation of hardware, there is a measurable cost for using even a simple EQAA setup and the cost scales with the number of rendered triangles and their shader complexity. However, in our scenario, it was still faster than SMAA alone. Even though we were able to solve multiple issues, we still found some finely tessellated content that was problematic with this technique and resulted in noisy artifacts on edges. These artifacts could be effectively filtered by temporal FLIPQUAD and TAA. Unfortunately the cost of outputting coverage data from pixel shaders was too high for our vegetation-heavy scenarios. We did not experiment with manual coverage output (i.e., not hardware based).

At the time of writing, we have decided to focus on two main approaches for our game: SMAA with AEAA used for alpha-tested geometry or CRAA with AEAA used for alpha-tested geometry. SMAA with AEAA is the most expensive and most reliable while also providing the lowest temporal stability. CRAA with AEAA provides excellent stability and performance with medium quality and medium reliability. The use of AEAA for alpha-tested objects seems to provide the highest quality, performance, and stability in both use cases; therefore, we integrated its resolve filter into the SMAA and CRAA resolves. See the performance and image quality comparisons of the full HRAA framework in Figure 3.25 and Table 3.2.

3.10 Conclusion

We provided a production proven hybrid reconstruction antialiasing framework along with several new algorithms, as well as modern implementations of well-known algorithms. We believe that the temporal FLIPQUAD super-sampling as well as temporal antialiasing will gain wider adoption due to their low cost, simplicity, and quality. Our improvements to distance-to-edge–based methods might prove useful for some projects. Meanwhile, CRAA is another addition to the temporally stable antialiasing toolbox. Considering its simplicity of implementation and its good performance, we believe that with additional research it might prove to be a viable, widely adopted edge antialiasing solution. We hope that the ideas presented here will inspire other researchers and developers and provide readers with valuable tools for achieving greater image quality in their projects.

Figure 3.25. Comparison of different HRAA setups showing different scenarios based on actual game content. From left to right: centroid sampling (no antialiasing), temporal FLIPQUAD (TFQ), AEAA + TFQ, CRAA + TFQ, and SMAA + TFQ.

Single Pass	Timing (ms)	G-Buffer Overhead (%)
BFECC single value	0.3	N/A
Temporal FLIPQUAD (TFQ)	0.2	N/A
AEAA	0.25	< 1% C
8×CRAA	0.25	< 8% HW/C
SMAA	0.9	N/A
TAA	0.6	N/A
TFQ + TAA	0.62	N/A
AEAA(alpha test) + 8×CRAA + TFQ + TAA	0.9	< 3% HW/C
SMAA + TFQ + TAA	1.4	N/A

Table 3.2. Different HRAA passes and timings measured on an AMD Radeon HD 7950 at 1080p resolution, operating on 32-bit image buffers. "C" means content dependent and "HW" means hardware type or setup dependent.

Bibliography

[Akenine-Möller 03] T. Akenine-Möller. "An Extremely Inexpensive Multisampling Scheme." Technical Report No. 03-14, Ericsson Mobile Platforms AB, 2003.

[AMD 11] AMD Developer Relations. "EQAA Modes for AMD 6900 Series Graphics Cards." http://developer.amd.com/wordpress/media/2012/10/EQAAModesforAMDHD6900SeriesCards.pdf, 2011.

[Alnasser 11] M. Alnasser, G. Sellers, and N. Haemel. "AMD Sample Positions." *OpenGL Extension Registry*, https://www.opengl.org/registry/specs/AMD/sample_positions.txt, 2011.

[Bavoil and Andersson 12] L. Bavoil and J. Andersson. "Stable SSAO in Battlefield 3 with Selective Temporal Filtering." Game Developer Conference Course, San Francisco, CA, March 5–9, 2012.

[Burley 07] B. Burley. "Filtering in PRMan." *Renderman Repository*, https://web.archive.org/web/20130915064937/http:/www.renderman.org/RMR/st/PRMan_Filtering/Filtering_In_PRMan.html, 2007. (Original URL no longer available.)

[Drobot 11] M. Drobot. "A Spatial and Temporal Coherence Framework for Real-Time Graphics." In *Game Engine Gems 2*, edited by Eric Lengyel, pp. 97–118. Boca Raton, FL: CRC Press, 2011.

[Drobot 14] M. Drobot. "Low Level Optimizations for AMD GCN Architecture." Presented at Digital Dragons Conference, Krakòw, Poland, May 8–9, 2014.

[Dupont and Liu 03] T. Dupont and Y. Liu. "Back and Forth Error Compensation and Correction Methods for Removing Errors Induced by Uneven Gradients of the Level Set Function." *J. Comput. Phys.* 190:1 (2003), 311–324.

[Jimenez et al. 11] J. Jimenez, B. Masia, J. Echevarria, F. Navarro, and D. Gutierrez. "Practical Morphological Antialiasing." In *GPU Pro 2: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 95–114. Natick, MA: A K Peters, 2011.

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

[Kirkland et al. 99] Dale Kirkland, Bill Armstrong, Michael Gold, Jon Leech, and Paula Womack. "ARB Multisample." *OpenGL Extension Registry*, https://www.opengl.org/registry/specs/ARB/multisample.txt, 1999.

[Laine and Aila 06] S. Laine and T. Aila. "A Weighted Error Metric and Optimization Method for Antialiasing Patterns." *Computer Graphics Forum* 25:1 (2006), 83–94.

[Lottes 09] T. Lottes. "FXAA." NVIDIA white paper, 2009.

[Malan 10] H. Malan. "Edge Anti-aliasing by Post-Processing." In *GPU Pro: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 265–290. Natick, MA: A K Peters, 2010.

[Malan 12] H. Malan. "Realtime global illumination and reflections in Dust 514." Advances in Real-Time Rendering in Games: Part 1, SIGGRAPH Course, Los Angeles, CA, August 5–9, 2012.

[NVIDIA 01] NVIDIA Corporation. "HRAA: High-Resolution Antialiasing Through Multisampling". Technical report, 2001.

[Nehab et al. 07] D. Nehab, P. V. Sander, J. Lawrence, N. Tararchuk, and J. R. Isidoro. "Accelerating Real-Time Shading with Reverse Reprojection Caching." In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, edited by Dieter Fellner and Stephen Spencer, pp. 25–35. Aire-la-Ville, Switzerland: Eurographics Association, 2007.

[Persson 11] E. Persson. "Geometric Buffer Antialiasing." Presented at SIGGRAPH, Vancouver, Canada, August 7–11, 2011.

[Reshetov 09] A. Reshetov. "Morphological Antialiasing." In *Proceedings of the Conference on High Performance Graphics*, edited by S. N. Spencer, David McAllister, Matt Pharr, and Ingo Wald, pp. 109–116. New York: ACM, 2009.

[Selle et al. 08] A. Selle, R. Fedkiw, B. Kim, Y. Liu, and J. Rossignac. "An Unconditionally Stable MacCormack Method." *J. Scientific Computing* 35:2–3 (2008), 350–371.

[Sousa 11] T. Sousa. "Anti-aliasing Methods in CryENGINE3." Presented at SIGGRAPH 2011, Vancouver, Canada, August 7–11, 2011.

[Sousa 13] T. Sousa T. "CryENGINE 3 Graphics Gems." Presented at SIGGRAPH 2013, Anaheim, CA, July 21–25, 2013.

[Valient 14] M. Valient. "Taking *Killzone Shadow Fall* Image Quality into the Next Generation." Presented at Game Developers Conference, San Francisco, CA, March 17–21, 2014.

[Yang et al. 09] L. Yang, D. Nehab, P. V. Sander, P. Sitthi-Amorn, J. Lawrence, and H. Hoppe. "Amortized Supersampling." Presented at SIGGRAPH Asia, Yokohama, Japan, December 17–19, 2009.

[Young 06] P. Young. "Coverage Sampled Antialiasing." Technical report, NVIDIA, 2006.