JORGE JIMENEZ, JOSE I. ECHEVARRIA, BELEN MASIA, FERNANDO NAVARRO, NATALYA TATARCHUK, DIEGO GUTIERREZ

DESTROY ALL JAGGIES

MLAA ON THE GPU — THE OPEN WAY

You've got your mind-blowing brand-new graphics engine featuring state-of-the-art lighting and shaders ... But looking closer, there are jaggies everywhere! You obviously need some kind of anti-aliasing in order to improve the final image quality, but which solution? Your typical choice would be to use multisampling anti-aliasing (MSAA). But you may already know (or are probably going to face) some of its limitations, such as big performance penalties for high qualities (4x and above), the hardware limitations of current consoles, and the troubles of combining it with multiple render targets and deferred shading. Then there is the inclusion of alpha to cover for proper anti-aliasing of transparencies, artifacts when resolving HDR framebuffers, plus depth-related artifacts when rendering objects after MSAA resolves, and the biggest limitation of all, additional memory cost for MSAA buffers. So, what can you do?

For the past few years, the solution has been to move anti-aliasing techniques to the shader units and apply them as any other postprocess in screen space. This has resulted in a plethora of custom solutions featured in games like S.T.A.L.K.E.R, TABULA RASA, CRYSIS, and BRUTAL LEGEND (just to name a few). It turns out that all of them share one core idea: edge detection and smoothing. Among these solutions, directionally localized anti-aliasing (DLAA—featured in STAR WARS THE FORCE UNLEASHED 2) shines with its smart use of blurs. Edge detection and smoothing is also one of the underlying ideas of morphological anti-aliasing (MLAA), a technique that was originally published by Intel, and has been getting some buzz on the internet during the past few months. As opposed to those custom solutions, with MLAA this smoothing is done adaptively by taking into account the coverage areas coming from a perceptual re-vectorization of the image. Compared with DLAA, MLAA is more accurate in terms of resemblance to the ground truth MSAA, while DLAA results, although good-looking, often present some blurriness.

MLAA detects edges based on color information. These are then classified according to a series of patterns. The final anti-aliasing is performed by blending each pixel with its neighbors, depending on the previously detected patterns. Intel's CPU code was later followed by the first practical in-game implementation in Sony's GOD OF WAR III, with great success in image quality. This implementation, running on PlayStation 3 SPUs, has been featured recently in other first-party titles like LITTLE BIG PLANET 2 and KILLZONE 3. Concurrently, Kalloc Studios developed another SPU-based anti-aliasing technique for the PS3 version of SABOTEUR, which is similar in spirit to MLAA. Hybrid CPU-GPU implementations on the Xbox 360 have surfaced in Double Fine's COSTUME QUEST and STACKING. On the PC side, AMD introduced its own proprietary implementation at driver level with the launch of its 6000 series, making it a great choice for a bunch of games that were completely lacking AA. However, their results tend to be blurry, and its activation via driver panel means that even the GUI is processed (something not desirable). While each has its own strengths, they either come from proprietary technology with varying degrees of customization, or are tied to specific platforms, so you may want a more open and universal solution.

In this article, we will discuss how to easily implement MLAA exclusively on the GPU, making its integration in your graphics engine child's play. This integration boils down to a standard pixel shader post-process, allowing for total control of when exactly you want

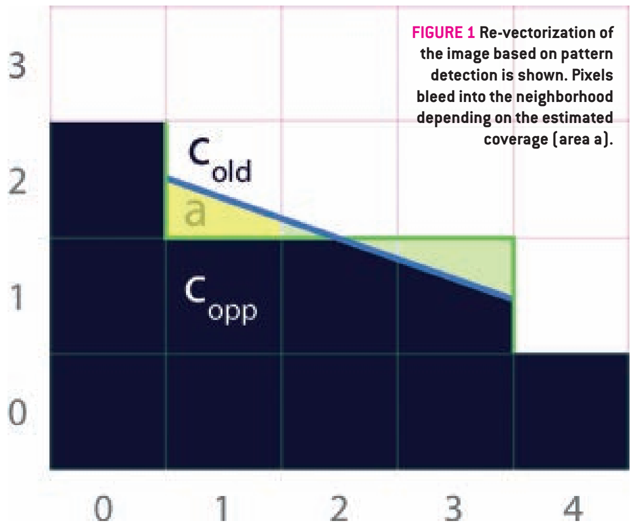


FIGURE 1 Re-vectorization of the image based on pattern detection is shown. Pixels bleed into the neighborhood depending on the estimated coverage (area a).

it to kick in, boosting flexibility and final image quality. This lightweight implementation takes just 0.44 ms, on average, on a mainstream NVIDIA 9800 GTX+ for a resolution of 720p. On the Xbox 360, for this same resolution, it takes 2.6 ms (which can be further reduced depending on the edge detection method used and specific engines). You can run it on any DX9-class GPU (and above), including the Xbox 360. Also for comparison, our implementation runs around 9x faster than AMD's MLAA on a Radeon 6870.

Are you ready to join us in the battle against the jaggies? If so, let's begin with the basics to set the stage for the real action!

FIRST THINGS FIRST: MLAA ABC

/// The main goal of anti-aliasing as a post-process is to work at 1x resolution throughout the graphics pipeline, optimizing memory consumption and sparing processing power. The color buffer you are going to apply AA over is typically going to be 1x. This means that you will not have subpixel information at hand. In that case, how do you know which pixels you have to use for the final blending? The answer is in the make-up of our own eyes, which assumes that on the edges, things behind one object will have the same color as their background neighborhood. Therefore, by properly mixing pixels on the edges, you can obtain a nice perceptual anti-aliasing.

As mentioned before, MLAA searches for specific patterns among the detected edges in order to perform the final blending. This process can be viewed as a re-vectorization of the original 3D scene. Take a look at Figure 1, which is a close-up of a dark triangle (already rasterized) over a white background.

Green lines mark the discontinuities that create the edges, which in turn define the pattern type. The basic pattern types are U-shaped, Z-shaped, and L-shaped, but there are many possible variations given that each pattern can be found flipped and/or rotated. The blue line is the estimated re-vectorization for that pattern. So, if this blue line represents a perfectly anti-aliased edge, we'd just have to bleed the black pixels into the areas of the upper part of the pattern, and the white pixels into the areas of the lower part. Thus, we can calculate the green and yellow areas and use them as weights for the final blending, like so:

$$c_{new} = (1 - a) \cdot c_{old} + a \cdot c_{opp}$$

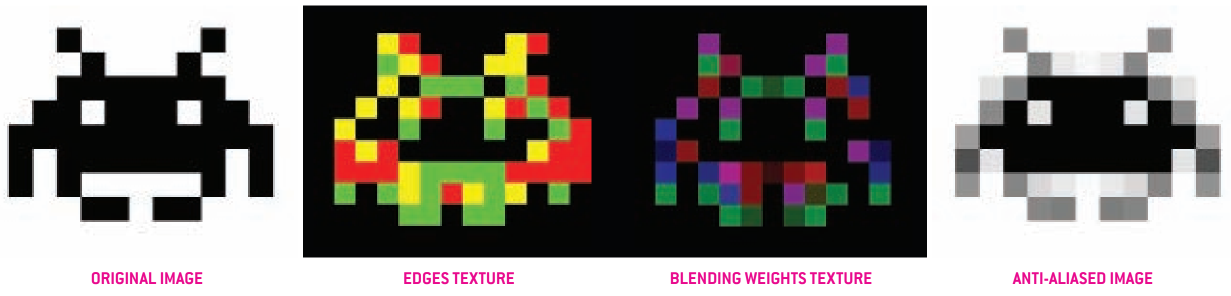
Here, c_{old} is the original color of the pixel, c_{opp} is the color of the pixel on the other side of the edge, and c_{new} is the new color of the pixel resulting from the bleeding of c_{opp} into c_{old} amount a (the area shown in yellow). The value of a approximates the coverage of the re-vectorized polygon, and is a function of both the pattern type of the edge and the position of the edgel (edge pixel) within the edge. The pattern type is defined by the crossing edges of the current edge, that is to say edges which are perpendicular to the current one, and thus define its ends (represented by the vertical green lines in Figure 1). In order to save processing time, this area a can be pre-computed and stored as a two-channel texture, as we will explain later on.

FROM LISTS TO TEXTURES

/// Intel's CPU-based implementation searches for specific patterns (U-shaped, Z-shaped, and L-shaped) that are then decomposed into simpler ones, an approach which would be impractical on current-generation GPU architectures , as this would involve complex dynamic branches. We have observed that the pattern type, and thus the anti-aliasing to be performed, only depends on four values (the four possible crossing edges), which can be obtained for each edgel with only two memory accesses. This way, Intel's algorithm is transformed in such a way that it uses texture structures instead of lists. Furthermore, our approach allows handling of all pattern types in a symmetric way, thus avoiding the need to decompose them into simpler ones and the use of a complex, branchy pattern-matching algorithm at run-time.

Our algorithm consists of three passes, the complete pipeline of which is shown in Figure 2. Starting from an original source image (with aliasing present—in Figure 2 left), we perform edge detection in the first pass. This yields a texture containing edgels (Figure 2, center-left). In the second pass, we process each edgel in the edges texture, generated in the previous pass, obtaining the corresponding blending weights of each pixel adjacent to the edgel being smoothed. To do this, we first calculate the distances from each edgel to the end of the lines to which it may belong (of which there are two: horizontal and vertical). Then, the crossing edges are fetched and used, together with the distances, to query the precomputed area texture (we can think of that as a look-up table), which returns the corresponding blending weights (Figure 2, center-right). The third and final pass involves blending each pixel with its 4-neighborhood using the blending weights from the previous pass to obtain the final anti-aliased image (Figure 2, right).

FIGURE 2 COMPLETE MLAA PIPELINE



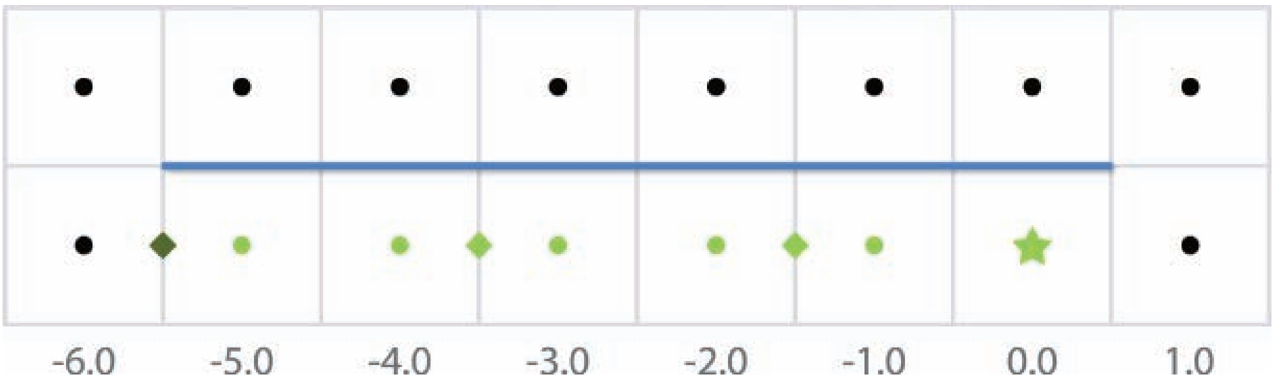


FIGURE 3 An example of a bilinear filter trick for distances search.

You may be thinking, why not just merge the last two passes into one? The reason is simple: Doing them separately allows you to spare calculations, taking advantage of the fact that two adjacent pixels share the same edgel. Put another way, you can think of the second pass as performing the pattern detection and the subsequent area calculation on a per-edgel basis, instead of on a per-pixel basis (which would be the naïve approach). This way, in the third pass, the two adjacent pixels (which share an edgel) will fetch the same information.

Since only a few pixels will need to be smoothed, we can add the stencil buffer into our recipe to apply the second and third passes only on the pixels which contain an edge, considerably reducing processing time. In the first pass, we store "1" in the stencil buffer for all pixels that contain an edge. Then, in the following passes, we just process pixels that have a stencil value of "1." As the destination buffer will not likely contain the color image, we probably won't want to just copy the processed edges into it in the last pass, as that would give us a black image with only anti-aliased edges on it. Thus, a full copy must be performed before executing this last pass. This is still faster than just disabling the stencil, as a hardware copy is relatively fast and only a few pixels are usually processed by our technique.

Now it's time to dive into the meaty nuts and bolts of our MLAA implementation!

STEP 1 LOOKING FOR EDGES

/// Edge detection is a critical step for the quality of the final image. Each undetected edge will remain aliased in the final image, so we need to detect as many edges as possible. Robustness in this step is also desirable, given that good edge detection enhances temporal stability. However, this is not as easy as it may sound. Optimally, we just want to detect edges that are visible to the human eye (no need to spare time anti-aliasing edges which won't be seen, right?). And not only that, we need clean edges as well, in order to detect their patterns properly.

In this article, we focus on a color-based edge detection, which is the most straightforward option. Depth, normals, or object IDs could also be used, since they are better estimators for geometrical edges; however, they are sometimes tricky and require extra information in the form of maps. Working with color also provides seamless handling of shading aliasing, which may improve quality in some scenarios.

First, we calculate luma values following the ITU-R Recommendation BT. 709:

$$Y' = 0.2126 \times R' + 0.7152 \times G' + 0.0722 \times B'$$

Note that R', G', B' are gamma-corrected values; this is crucial to performing accurate edge detection, so pay attention to your `SRGBTexture` flags and `DXGI_FORMAT_R8G8B8A8_UNORM_SRGB` texture flags, in DirectX 9 and 10 respectively.

Armed with this info, we calculate and threshold the luma differences between the current pixel and its top and left neighbors in order to obtain a binary value indicating whether there is a sharp edge in between. The result of this pass is stored as a two-channel edges texture (as we saw in Figure 2, center-left). The color of each pixel codes the location of the edges: green pixels have an edge at the top, red pixels have them at their left, and yellow pixels have edges at both boundaries. We actually oversimplified a little bit here: Since we need to create the stencil mask in this first pass and because it needs to tag every pixel that has an edge (in either the top, right, bottom, or left boundaries), we actually calculate and threshold the differences between the pixel and the full 4-neighborhood. Listing 1 shows the shader code of this luma-based edge detection.

In low-end cards, these dot products can introduce an important

LISTING 1 COLOR-BASED EDGE-DETECTION SHADER

```
float4 ColorEdgeDetectionPS(float4 position : SV_POSITION,
                                        float2 texcoord : TEXCOORD0) :
SV_TARGET {
float3 weights = float3(0.2126,0.7152, 0.0722);

/**
   * Luma calculation requires gamma-corrected colors, and thus
´colorTex´ should
   * be a non-sRGB texture.
   */
   float L = dot(colorTex.SampleLevel(PointSampler, texcoord,
0).rgb, weights);
   float Lleft = dot(colorTex.SampleLevel(PointSampler, texcoord,
0, -int2(1, 0)).rgb, weights);
   float Ltop  = dot(colorTex.SampleLevel(PointSampler, texcoord,
0, -int2(0, 1)).rgb, weights);
   float Lright = dot(colorTex.SampleLevel(PointSampler, texcoord,
0, int2(1, 0)).rgb, weights);
   float Lbottom  = dot(colorTex.SampleLevel(PointSampler,
texcoord, 0, int2(0, 1)).rgb, weights);

   float4 delta = abs(L.xxxx - float4(Lleft, Ltop, Lright,
Lbottom));
  float4 edges = step(threshold.xxxx, delta);

   if (dot(edges, 1.0) == 0.0)
    discard;

   return edges;
}
```

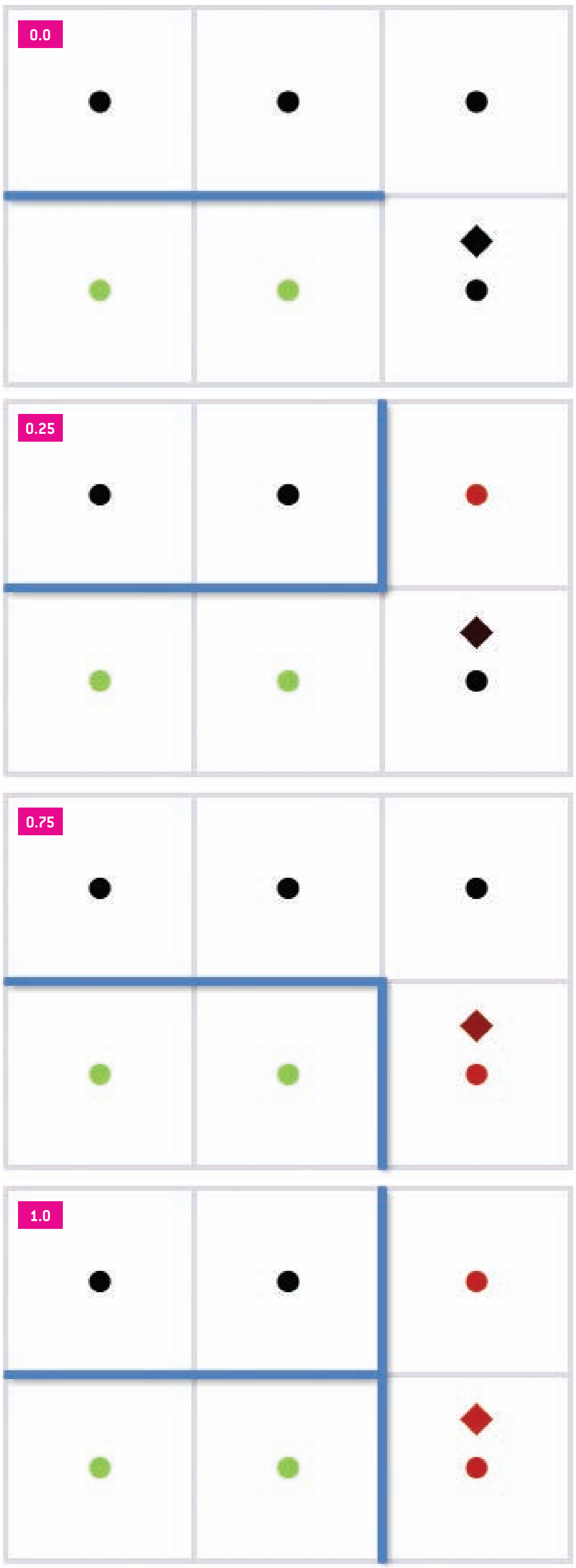


FIGURE 4 **Crossing edges and their corresponding value using bilinear filter fetch.**

overhead (given the fact that this pixel shader is executed for all the pixels in the framebuffer); thus, it can be wise to calculate lumas in the main render pass, and store them in the alpha channel of the color render target. If a z pre-pass is performed by the engine, this reduces the five dot products required per pixel to only one. Furthermore, in a DirectX 9 implementation, since offsets cannot be directly specified in texture functions, further cycles can be saved by offloading their calculations to the vertex shader.

STEP 2 OBTAINING BLENDING WEIGHTS

/// We're now getting to the most complex step of our crusade against the jaggies, so please keep Figure 1 in mind for what comes next: blending weights computation. To do this, we need to obtain the distances to the ends of the line segment that each edgel belongs to, using the edges textures from the previous pass. Once we know these distances, we will use them to fetch the crossing edges at both ends of the line. These crossing edges indicate the type of pattern we are dealing with. Both the distances to the ends of the line segment and the type of pattern are used to access the pre-calculated area texture used for blending in the final pass.

As you may have already noticed, two adjacent pixels share the same boundary. This allows sharing of calculations between adjacent pixels—you can perform an area calculation on a per-edgel basis. However, even though two adjacent pixels share the same calculation, the resulting a value is different for each of them: only one has a blending weight *a*, whereas for the opposite one *a* equals zero (pixels (1,2) and (1,1) in Figure 1, respectively). The one exception to this is the case in which the pixel lies at the middle of a line of odd length (as in pixel (2,1) in Figure 1); in this case, both the actual pixel and its opposite have a non-zero value for *a*. As a consequence, the output of this pass is a texture that, for each pixel, stores the areas at each side of its corresponding edgels (by the areas at each side, we mean those of the actual pixel and its opposite). This yields two values for north edgels and two values for west edgels in the final blending weights texture, perfectly fitting in the allocated RGBA storage. These weights will be used in the third pass to perform the final blending. Listing 2 shows the source code of this pass, while Figure 2, center-right, shows the resulting blending weights texture.

SEARCHING FOR DISTANCES

/// The search for the distances to the ends of the line is done using an iterative algorithm. In each iteration, it checks whether the end of the line has been reached. To accelerate this search, we leverage the fact that the information stored in the edges texture is binary (as it simply encodes whether an edgel exists), and query at positions between pixels using bilinear filtering for fetching two pixels at a time, thus advancing two pixels per iteration. The result of the query can be:

A. 0.0, WHICH MEANS THAT NEITHER PIXEL CONTAINS AN EDGEL,
B. 1.0, WHICH IMPLIES AN EDGEL EXISTS IN BOTH PIXELS, OR
C. 0.5, WHICH IS RETURNED WHEN JUST ONE OF THE TWO PIXELS CONTAINS AN EDGEL.

Stop the search if the returned value is lower than one (in practice, we use 0.9 due to bilinear filtering precision issues). By using a simple approach like this, we are introducing two sources of inaccuracy. First, we do not stop the search when encountering an edgel perpendicular to the line we are following, but when the line comes to an end instead. Second, when the returned value is 0.5, we cannot distinguish which of the two pixels contains an edgel. While these inaccuracies may introduce errors in some cases, we found them not be noticeable in practice. Moreover, the speed-up resulting from jumping two pixels per iteration is considerable.

Figure 3 shows an example where the color of the dot at the center of each pixel represents its value in the edges texture. The distance search for the left end of the line is performed for the pixel marked with a star. Positions where the edges texture is accessed, fetching pairs of pixels, are

LISTING 2 BLENDING WEIGHTS CALCULATION SHADER

```
float4 BlendingWeightCalculationPS(float4 position : SV_POSITION,
                                   float2 texcoord : TEXCOORD0) :
SV_TARGET {
  float4 weights = 0.0;

  float2 e = edgesTex.SampleLevel(PointSampler, texcoord, 0).rg;

  [branch]
  if (e.g) { // Edge at north

  // Search distances to the left and to the right:
  float2 d = float2(SearchXLeft(texcoord), SearchXRight(texcoord));

  // Now fetch the crossing edges. Instead of sampling between
edgels, we
  // sample at -0.25, to be able to discern what value each edgel
has:
  float4 coords = mad(float4(d.x, -0.25, d.y + 1.0, -0.25),
                       PIXEL_SIZE.xyxy, texcoord.xyxy);
  float e1 = edgesTex.SampleLevel(LinearSampler, coords.xy, 0).r;
   float e2 = edgesTex.SampleLevel(LinearSampler, coords.zw, 0).r;

  // Ok, we know how this pattern looks; now it is time for getting
  // the actual area:
   weights.rg = Area(abs(d), e1, e2);
 }

  [branch]
  if (e.r) { // Edge at west

  // Search distances to the top and to the bottom:
  float2 d = float2(SearchYUp(texcoord), SearchYDown(texcoord));

  // Now fetch the crossing edges (yet again):
  float4 coords = mad(float4(-0.25, d.x, -0.25, d.y + 1.0),
                       PIXEL_SIZE.xyxy, texcoord.xyxy);
  float e1 = edgesTex.SampleLevel(LinearSampler, coords.xy, 0).g;
  float e2 = edgesTex.SampleLevel(LinearSampler, coords.zw, 0).g;

  // Get the area for this direction:
  weights.ba = Area(abs(d), e1, e2);
  }

  return weights;
}
```

marked with rhombuses (their color represents the fetched value).

In order to keep execution times practical, we limit the search to a certain distance. As expected, the greater the maximum distance, the better the quality of the anti-aliasing. However, we have found that for the majority of cases, distance values between 8 and 16 pixels offer a good trade-off between quality and performance. Listing 3 shows one of the distance search functions.

In the particular case of the Xbox 360 implementation, we make use of the `tfetch2D` assembler instruction, which allows us to specify an offset in pixel units with respect to the original texture coordinates of the query. This instruction is limited to offsets of –8 and 7.5, which constrains the maximum distance that can be searched. When searching for distances greater than eight pixels we cannot use the hardware as efficiently, and performance takes a hit.

LISTING 3 DISTANCE SEARCH FUNCTION (LEFT DIRECTION CASE)

```
float SearchXLeft(float2 texcoord) {
  texcoord -= float2(1.5, 0.0) * PIXEL_SIZE;
  float e = 0.0;
  // We offset by 0.5 to sample between edgels, thus fetching two
in a row
  for (int i = 0; i < maxSearchSteps; i++) {
    e = edgesTex.SampleLevel(LinearSampler, texcoord, 0).g;
  // We compare with 0.9 to prevent bilinear access precision
problems
  [flatten] if (e < 0.9) break;
  texcoord -= float2(2.0, 0.0) * PIXEL_SIZE;
  }
  // When we exit the loop without finding the end, we want to
return
  // -2 * maxSearchSteps
  return max(-2.0 * i - 2.0 * e, -2.0 * maxSearchSteps);
}
```

LISTING 4 PRECOMPUTED AREA TEXTURE ACCESS FUNCTION

```
#define MAX_DISTANCE 32

float2 Area(float2 distance, float e1, float e2) {
  // * By dividing by areaSize - 1.0 below we are implicitly
offsetting to
  //   always fall inside a pixel
  // * Rounding prevents bilinear access precision problems
  float areaSize = MAX_DISTANCE * 5.0;
  float2 pixcoord = MAX_DISTANCE * round(4.0 * float2(e1, e2)) +
distance;
  float2 texcoord = pixcoord / (areaSize - 1.0);
  return areaTex.SampleLevel(PointSampler, texcoord, 0).rg;
}
```

FETCHING CROSSING EDGES

/// Once we have the distances to the ends of the line, we use them to obtain the crossing edges. A naïve approach for fetching the crossing edge of an end of a line would imply querying two edgels. A more efficient approach is to use bilinear filtering for fetching both edgels at a time, similar to the distance search. However, in this case, we must be able to distinguish the actual value of each edgel, so we query with an offset of 0.25, allowing us to distinguish which edgel is equal to 1.0 when only one of the edgels is present. Figure 4 shows the crossing edge corresponding to each of the different values returned by the bilinear query. The color of the dot at the center of each pixel represents the value of that pixel in the edges texture. The rhombuses indicate the sampling position, while their color represents the value returned by the bilinear access.

THE PRECOMPUTED AREA TEXTURE

/// With distances and crossing edges at hand, we now have all the ingredients for calculating the area which corresponds to the current pixel. Since this is an expensive operation, we pre-compute it in a 4D table, stored as a conventional 2D texture (see Figure 5, right). This texture is divided into subtextures of size 9x9, each of them corresponding to a pattern type coded by the fetched crossing edges e1 and e2 at each end of the line. Figure 5 (left) shows you the 16 different patterns we handle (each one with direct correspondence to a subtexture); the orange lines indicate the perceptual re-vectorization to be performed in each case.

LISTING 5 4-NEIGHBORHOOD BLENDING SHADER

```
float4 NeighborhoodBlendingPS(float4 position : SV_POSITION,
                              float2 texcoord : TEXCOORD0) :
SV_TARGET {
  // Fetch the blending weights for current pixel:
  float4 topLeft = blendTex.SampleLevel(PointSampler, texcoord,
0);
  float bottom = blendTex.SampleLevel(PointSampler, texcoord, 0,
int2(0, 1)).g;
  float right = blendTex.SampleLevel(PointSampler, texcoord, 0,
int2(1, 0)).a;
  float4 a = float4(topLeft.r, bottom, topLeft.b, right);

  // Up to 4 lines can be crossing a pixel (one in each edge).
Thus, we perform
  // a weighted average, where the weight of each line is ´a´
cubed, which
  // favors blending and works well in practice.
  float4 w = a * a * a;

  // Is there any blending weight with a value greater than 0.0?
  float sum = dot(w, 1.0);
  [branch]
  if (sum > 0.0) {
    float4 o = a * PIXEL_SIZE.yyxx;
    float4 color = 0.0;

  // Add the contributions of the 4 possible lines that can cross
this
  // pixel:
  color = mad(colorTex.SampleLevel(LinearSampler, texcoord +
float2( 0.0, -o.r), 0), w.r, color);
  color = mad(colorTex.SampleLevel(LinearSampler, texcoord +
float2( 0.0,  o.g), 0), w.g, color);
  color = mad(colorTex.SampleLevel(LinearSampler, texcoord +
float2(-o.b,  0.0), 0), w.b, color);
  color = mad(colorTex.SampleLevel(LinearSampler, texcoord +
float2( o.a,  0.0), 0), w.a, color);

   // Normalize the resulting color and we are finished!
   return color / sum;
 } else {
   return colorTex.SampleLevel(LinearSampler, texcoord, 0);
 }
}
```

Inside each of these subtextures, the (u, v) coordinates correspond to distances to the ends of the line, eight being the maximum. Resolution can be increased if a higher maximum distance is required.

Maybe at this point you are wondering why, according to Figure 5, patterns 5, 10, and 15 do not perform any anti-aliasing. That's because we found objects like rails (patterns 5 and 10) and contiguous quads (pattern 15) to be better preserved this way. Compound patterns like 7, 11, 13, and 14 are also special cases. In these cases, in order to obtain the best results, you must choose a main pattern in order to reduce artifacts. We chose Z-shapes, but U-shapes could also be a valid choice.

Following the same reasoning, in which we store area values for two adjacent pixels in the same pixel of the final blending weights texture, the precomputed area texture needs to be built on a per-edgel basis. Thus, each pixel of the texture stores two *a* values, both for a pixel and its opposite (again, *a* will be zero for one of them in all cases except those

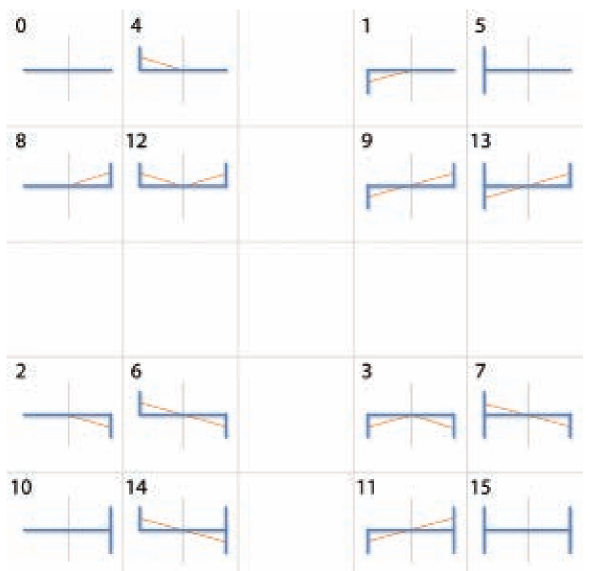


FIGURE 5 Processed patterns (left) and their corresponding precalculated weights depending on their size (right).

of pixels at the center of lines of odd length).

Listing 4 gives details on how the precomputed area texture is accessed. To query the texture, we first convert the bilinear filtered values e1 and e2 to an integer value in the range [0..4] Value 2 (which would correspond to value 0.5 for e1 or e2) cannot occur in practice, which is why the corresponding row and column in the texture are empty. Maintaining those empty spaces in the texture allows for a simpler and faster indexing. The round instruction is used to avoid possible precision problems caused by the bilinear filtering.

STEP 3 BLENDING WITH THE NEIGHBORHOOD

/// We already have the edges where anti-aliasing needs to be performed, plus the blending weights. In this last pass, we will obtain the final color of each pixel by blending the current color with its four neighbors according to the area values stored in the weights texture. To do this, we have to access three positions in the blending weights texture:

A. THE CURRENT PIXEL, WHICH GIVES US THE NORTH AND WEST BLENDING WEIGHTS;
B. THE PIXEL AT THE SOUTH; AND
C. THE PIXEL AT THE EAST.

This yields the blending weights with the complete 4-neighborhood. Once more, to exploit hardware capabilities, we use four bilinear filtered accesses to blend the current pixel with each of its four neighbors. Finally, given that one pixel can belong to four different lines, we perform a weighted average between the contributing lines. The cubed blending weight (*a* cubed) of each possible line is used as the weight of this average, which favors blending and works well in practice. Listing 5 shows the source code of this pass, while Figure 2, right, shows the resulting anti-aliased image.

It's important to note that if you want all this blending calculated properly, you have to ensure you are working in linear space. Using bilinear filtering and DXGI_FORMAT_R8G8B8A8_UNORM_SRGB textures for calculating this step in DirectX 10 enforces linear blending. In DirectX 10 hardware running DirectX 9 code, this bilinear filtering blending will be performed, again, in linear space. However, DirectX 9 hardware running in DirectX 9 will perform the blending in gamma space. Thus, in this case, manual blending using lerps is advised.

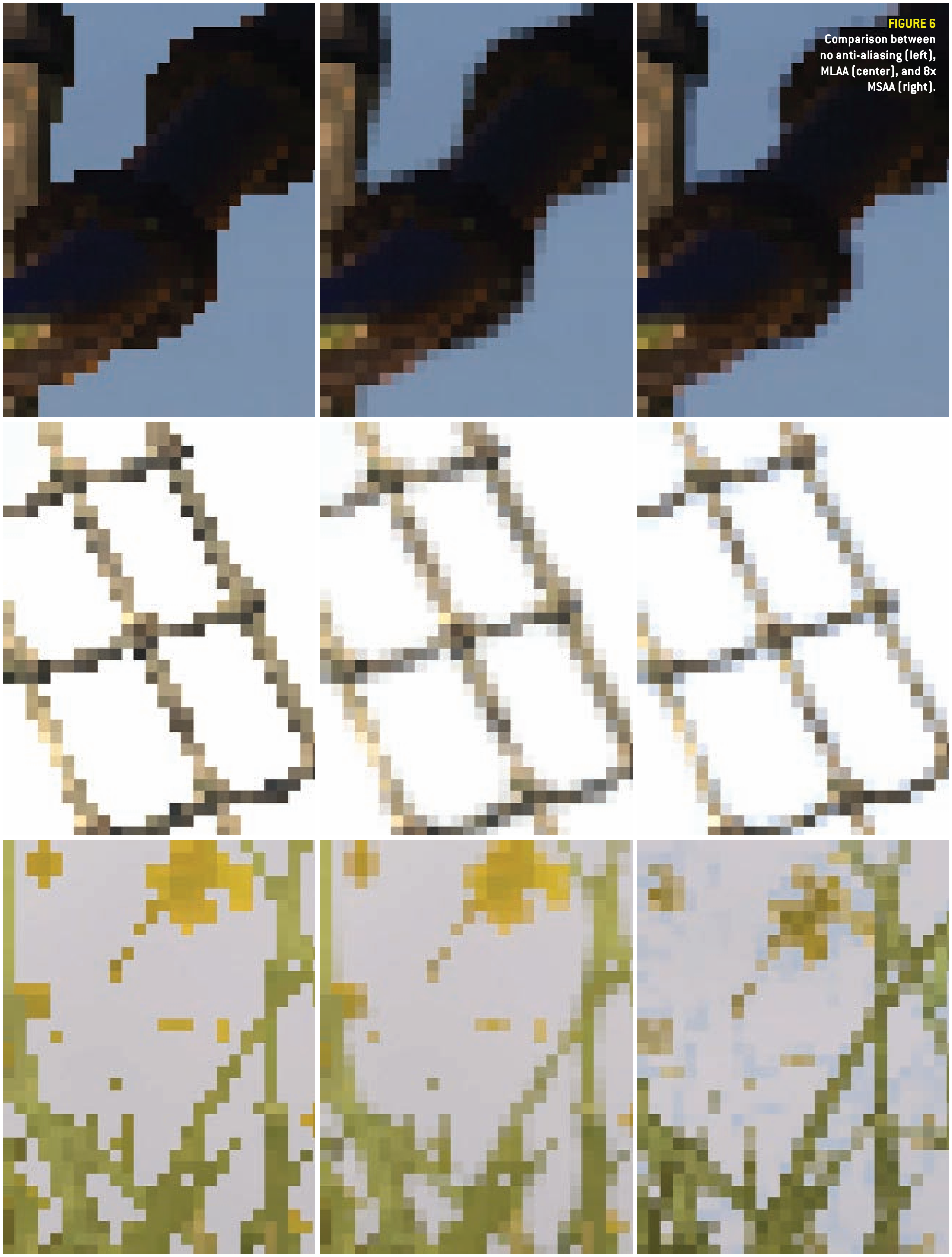


FIGURE 6 Comparison between no anti-aliasing (left), MLAA (center), and 8x MSAA (right).

Figure 8. More examples of our technique applied over images from Unigine's Heaven demo. Insets show no anti-aliasing (left) and MLAA (right).

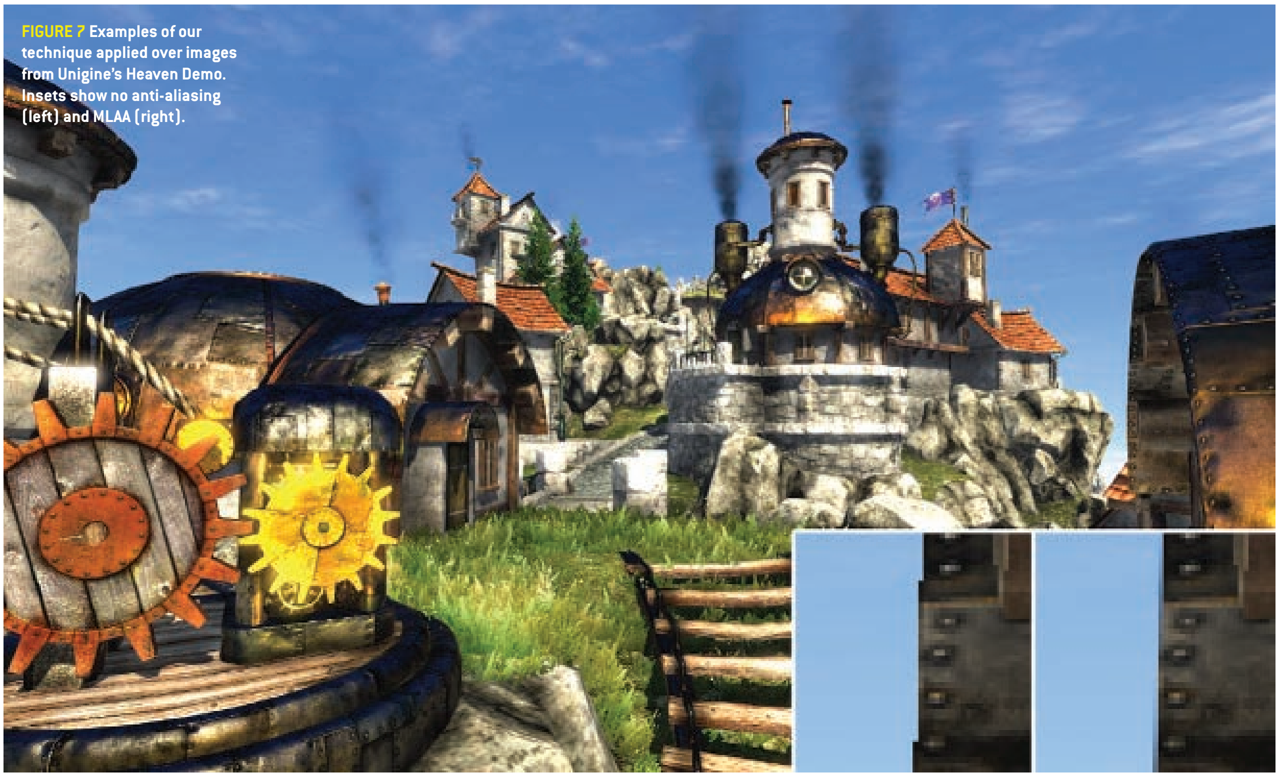


FIGURE 7 Examples of our technique applied over images from Unigine's Heaven Demo. Insets show no anti-aliasing (left) and MLAA (right).

WRAP-UP

/// So, what are the benefits of this MLAA implementation for your graphics engine? Well, you can run this technique as a regular pixel shader in DirectX 9 and above (including the Xbox 360). In comparison to proprietary techniques, it has the additional advantage that it can be triggered at any desired step along the pipeline (usually somewhere after tone mapping and before GUI rendering), providing you with great flexibility. Furthermore, now that you know what is under the hood, you can tweak it to fit your game like a glove.

When edge detection fails, our technique can be as bad as 1x (in fact, in these cases, it is 1x). In the presence of sub-pixel features MSAA can be superior (although proper care in art direction can solve some cases). However, our implementation is comparable, in general, to 16x MSAA, while only requiring a memory consumption of 1.5x the size of the backbuffer on DirectX 10, and of 2x on DirectX 9-based implementations. Figure 6 shows a comparison between the algorithm, 8x MSAA, and no anti-aliasing at all on images from the Unigine Heaven benchmark. More results of our technique are shown in Figures 7 and 8. Take a look at our project page for additional information about the technique, including an exhaustive performance analysis, an image gallery, a movie, and implementations for both DirectX 9 and 10: www.iryoku.com/mlaa.

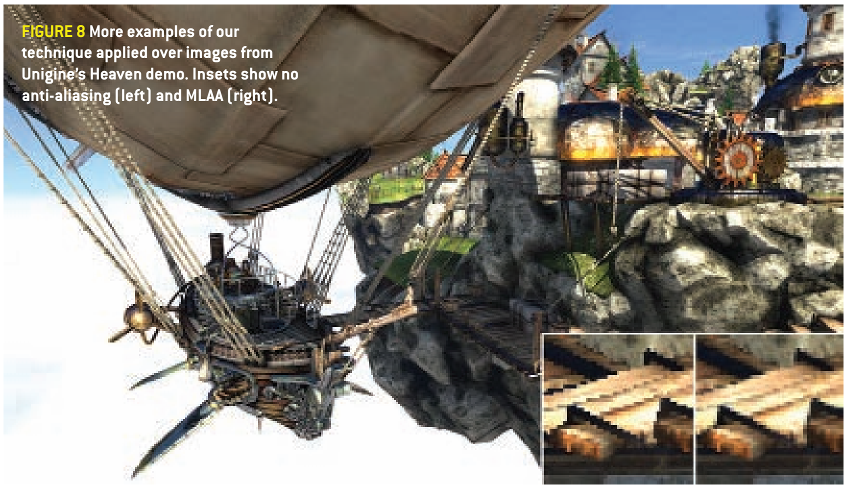


FIGURE 8 More examples of our technique applied over images from Unigine's Heaven demo. Insets show no anti-aliasing (left) and MLAA (right).

Typical execution times are 2.60ms on the Xbox 360 and 0.44 ms on an NVIDIA GeForce 9800 GTX+ for a resolution of 720p (tested in DirectX 10 and XNA, respectively). According to our measurements, 8x MSAA takes an average of 5 ms per image on the same GPU at the same resolution—that is, our algorithm is 11.80x faster.

The method presented has a minimal impact on existing rendering pipelines, and is entirely implemented as an image post-process. Resulting images can be on par with 16x MSAA in terms of quality, while requiring a fraction of their time and memory consumption. Furthermore, it can anti-alias transparent textures such as the ones used in alpha testing for rendering vegetation, whereas MSAA can only smooth vegetation when using alpha to coverage. We believe that the quality of the images produced by our algorithm, its speed, efficiency, and pluggability, make it an attractive choice for rendering high-quality images in today's game architectures, including platforms where benefiting from anti-aliasing together with outstanding techniques like deferred shading was previously difficult to achieve.

Some of the information presented in this article has been adapted by the original authors from the GPU Pro 2 chapter "Practical Morphological Anti-Aliasing."

AUTHORS: JORGE JIMENEZ *Universidad de Zaragoza*, JOSE I. ECHEVARRIA *Universidad de Zaragoza*, BELEN MASIA *Universidad de Zaragoza*, FERNANDO NAVARRO *Lionhead Studios*, NATALYA TATARCHUK *Bungie*, DIEGO GUTIERREZ *Universidad de Zaragoza*