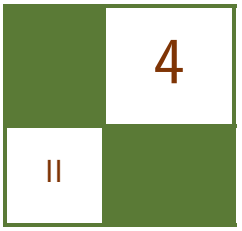

Practical Morphological Antialiasing

Jorge Jimenez, Belen Masia, Jose I. Echevarria, Fernando Navarro, and Diego Gutierrez

The use of antialiasing techniques is crucial when producing high quality graphics. Up to now, *multisampling antialiasing* (MSAA) has remained the most advanced solution, offering superior results in real time. However, there are important drawbacks to the use of MSAA in certain scenarios. First, the increase in processing time it consumes is not negligible at all. Further, limitations of MSAA include the impossibility, in a wide range of platforms, of activating multisampling when using *multiple render targets* (MRT), on which fundamental techniques such as deferred shading [Shishkovtsov 05, Koonce 07] rely. Even on platforms where MRT and MSAA can be simultaneously activated (i.e., DirectX 10), implementation of MSAA is neither trivial nor cost free [Thibieroz 09]. Additionally, MSAA poses a problem for the current generation of consoles. In the case of the Xbox 360, memory constraints force the use of CPU-based tiling techniques in case high-resolution frame buffers need to be used in conjunction with MSAA; whereas on the PS3 multisampling is usually not even applied. Another drawback of MSAA is its inability to smooth nongeometric edges, such as those resulting from the use of alpha testing, frequent when rendering vegetation. As a result, when using MSAA, vegetation can be antialiased only if alpha to coverage is used. Finally, multisampling requires extra memory, which is always a valuable resource, especially on consoles.

In response to the limitations described above, a series of techniques have implemented antialiasing solutions in shader units, the vast majority of them being based on edge detection and blurring. In *S.T.A.L.K.E.R.* [Shishkovtsov 05], edge detection is performed by calculating differences in the eight-neighborhood depth values and the four-neighborhood normal angles; then, edges are blurred using a cross-shaped sampling pattern. A similar, improved scheme is used in *Tabula Rasa* [Koonce 07], where edge detection uses threshold values that are resolution

independent, and the full eight-neighborhood of the pixel is considered for differences in the normal angles. In *Crysis* [Sousa 07], edges are detected by using depth values, and rotated triangle samples are used to perform texture lookups using bilinear filtering. These solutions alleviate the aliasing problem but do not mitigate it completely. Finally, in *Killzone 2*, samples are rendered into a double horizontal resolution G-buffer. Then, in the lighting pass, two samples of the G-buffer are queried for each pixel of the final buffer. The resulting samples are then averaged and stored in the final buffer. However, this necessitates executing the lighting shader twice per final pixel.

In this article we present an alternative technique that avoids most of the problems described above. The quality of our results lies between $4x$ and $8x$ MSAA at a fraction of the time and memory consumption. It is based on morphological antialiasing [Reshetov 09], which relies on detecting certain image patterns to reduce aliasing. However, the original implementation is designed to be run in a CPU and requires the use of list structures that are not GPU-amenable.

Since our goal is to achieve real-time practicality in games with current mainstream hardware, our algorithm implements aggressive optimizations that provide an optimal trade-off between quality and execution times. Reshetov searches for specific patterns (U-shaped, Z-shaped, and L-shaped patterns), which are then decomposed into simpler ones, an approach that would be impractical on a GPU. We realize that the pattern type, and thus the antialiasing to be performed, depends on only four values, which can be obtained for each edge pixel (*edgel*) with only two memory accesses. This way, the original algorithm is transformed so that it uses texture structures instead of lists (see Figure 4.1). Furthermore, this approach allows handling of all pattern types in a symmetric way, thus avoiding the need to decompose them into simpler ones. In addition, precomputation of

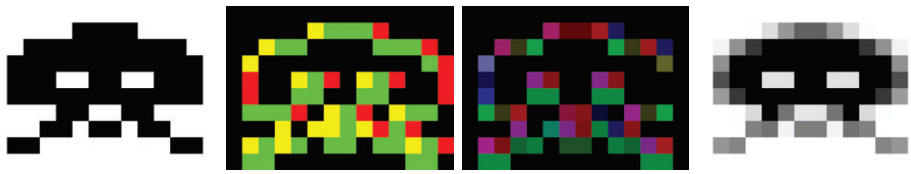

Figure 4.1. Starting from an aliased image (left), edges are detected and stored in the edges texture (center left). The color of each pixel depicts where edges are: green pixels have an edge at their top boundary, red pixels at their left boundary, and yellow pixels have edges at both boundaries. The edges texture is then used in conjunction with the precomputed area texture to produce the blending weights texture (center right) in the second pass. This texture stores the weights for the pixels at each side of an edgel in the RGBA channels. In the third pass, blending is performed to obtain the final antialiased image (right).

certain values into textures allows for an even faster implementation. Finally, in order to accelerate calculations, we make extensive use of hardware bilinear interpolation for smartly fetching multiple values in a single query and provide means of decoding the fetched values into the original unfiltered values. As a result, our algorithm can be efficiently executed by a GPU, has a moderate memory footprint, and can be integrated as part of the standard rendering pipeline of any game architecture.

Some of the optimizations presented in this work may seem to add complexity at a conceptual level, but as our results show, their overall contribution makes them worth including. Our technique yields image quality between $4x$ and $8x$ MSAA, with a typical execution time of 3.79 ms on Xbox 360 and 0.44 ms on a NVIDIA GeForce 9800 GTX+, for a resolution of 720p. Memory footprint is $2x$ the size of the backbuffer on Xbox 360 and $1.5x$ on the 9800 GTX+. 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.

In order to show the versatility of our algorithm, we have implemented the shader both for Xbox 360 and PC, using DirectX 9 and 10 respectively. The code presented in this article is that of the DirectX 10 version.

4.1 Overview

The algorithm searches for patterns in edges which then allow us to reconstruct the antialiased lines. This can, in general terms, be seen as a revectorization of edges. In the following we give a brief overview of our algorithm.

First, edge detection is performed using depth values (alternatively, luminances can be used to detect edges; this will be further discussed in Section 4.2.1). We then compute, for each pixel belonging to an edge, the distances in pixels from it to both ends of the edge to which the edgel belongs. These distances define the position of the pixel with respect to the line. Depending on the location of the edgel within the line, it will or will not be affected by the antialiasing process. In those edges which have to be modified (those which contain yellow or green areas in Figure 4.2 (left)) a blending operation is performed according to Equation (4.1):

$$c_{\text{new}} = (1 - a) \cdot c_{\text{old}} + a \cdot c_{\text{opp}}, \tag{4.1}$$

where c_{old} is the original color of the pixel, c_{opp} is the color of the pixel on the other side of the line, c_{new} is the new color of the pixel, and a is the area shown in yellow in Figure 4.2 (left). The value of a is a function of both the pattern type of the line and the distances to both ends of the line. The pattern type is defined by the *crossing edges* of the line, i.e., edges which are perpendicular to the line and thus define the ends of it (vertical green lines in Figure 4.2). In order to save processing time, we precompute this area and store it as a two-channel texture that can be seen in Figure 4.2 (right) (see Section 4.3.3 for details).

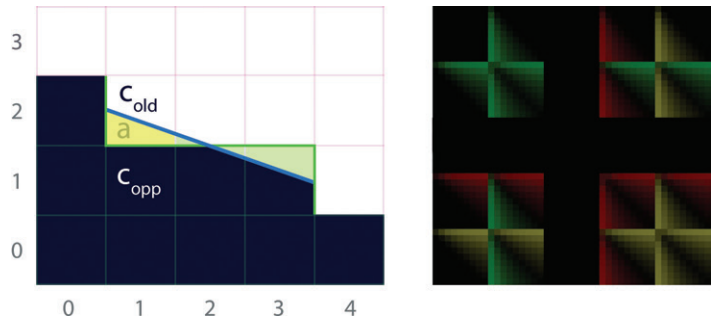

Figure 4.2. Antialiasing process (left). Color c_{opp} bleeds into c_{old} according to the area a below the blue line. Texture containing the precomputed areas (right). The texture uses two channels to store areas at each side of the edge, i.e., for a pixel and its opposite (pixels $(1, 1)$ and $(1, 2)$ on the left). Each 9×9 subtexture corresponds to a pattern type. Inside each of these subtextures, (u, v) coordinates encode distances to the left and to the right, respectively.

The algorithm is implemented in three passes, which are explained in detail in the following sections. In the first pass, edge detection is performed, yielding a texture containing edges (see Figure 4.1 (center left)). In the second pass the corresponding blending weight[1] (that is, value a) for each pixel adjacent to the edge being smoothed is obtained (see Figure 4.1 (center right)). To do this, we first detect the pattern types for each line passing through the north and west boundaries of the pixel and then calculate the distances of each pixel to the *crossing edges*; these are then used to query the precomputed area texture. The third and final pass involves blending each pixel with its four-neighborhood using the blending weights texture obtained in the previous pass.

The last two passes are performed separately to spare calculations, taking advantage of the fact that two adjacent pixels share the same edgel. To do this, in the second pass, pattern detection and the subsequent area calculation are performed on a per-edgel basis. Finally, in the third pass, the two adjacent pixels will fetch the same information.

Additionally, using the *stencil buffer* allows us to perform the second and third passes only for the pixels which contain an edge, considerably reducing processing times.

4.2 Detecting Edges

We perform edge detection using the depth buffer (or luminance values if depth information is not available). For each pixel, the difference in depth with respect to the pixel on top and on the left is obtained. We can efficiently store the edges

[1] Throughout the article *blending weight* and *area* will be used interchangeably.

for all the pixels in the image this way, given the fact that two adjacent pixels have a common boundary. This difference is thresholded to obtain a binary value, which indicates whether an edge exists in a pixel boundary. This threshold, which varies with resolution, can be made resolution independent [Koonce 07]. Then, the left and top edges are stored, respectively, in the red and green channels of the edges texture, which will be used as input for the next pass.

Whenever using depth-based edge detection, a problem may arise in places where two planes at different angles meet: the edge will not be detected because of samples having the same depth. A common solution to this is the addition of information from normals. However, in our case we found that the improvement in quality obtained when using normals was not worth the increase in execution time it implied.

4.2.1 Using Luminance Values for Edge Detection

An alternative to depth-based edge detection is the use of luminance information to detect image discontinuities. Luminance values are derived from the CIE XYZ (color space) standard:

$$L = 0.2126 \cdot R + 0.7152 \cdot G + 0.0722 \cdot B.$$

Then, for each pixel, the difference in luminance with respect to the pixel on top and on the left is obtained, the implementation being equivalent to that of depth-based detection. When thresholding to obtain a binary value, we found 0.1 to be an adequate threshold for most cases. It is important to note that using either luminance- or depth-based edge detection does not affect the following passes.

Although qualitywise both methods offer similar results, depth-based detection is more robust, yielding a more reliable edges texture. And, our technique takes, on average, 10% less time when using depth than when using luminance values. Luminance values are useful when depth information cannot be accessed and thus offer a more universal approach. Further, when depth-based detection is performed, edges in shading will not be detected, whereas luminance-based detection allows for antialias shading and specular highlights. In general terms, one could say that luminance-based detection works in a more perceptual way because it smoothes *visible* edges. As an example, when dense vegetation is present, using luminance values is faster than using depth values (around 12% faster for the particular case shown in Figure 4.5 (bottom row)), since a greater number of edges are detected when using depth values. Optimal results in terms of quality, at the cost of a higher execution time, can be obtained by combining luminance, depth, and normal values.

Listing 4.1 shows the source code of this pass, using depth-based edge detection. Figure 4.1 (center left) is the resulting image of the edge-detection pass, in this particular case, using luminance-based detection, as depth information is not available.

```
float4 EdgeDetectionPS(float4 position: SV_POSITION,
                       float2 texcoord: TEXCOORD0): SV_TARGET {

  float D = depthTex.SampleLevel(PointSampler,
                                 texcoord, 0);
  float Dleft = depthTex.SampleLevel(PointSampler,
                                     texcoord, 0, -int2(1, 0));
  float Dtop  = depthTex.SampleLevel(PointSampler,
                                     texcoord, 0, -int2(0, 1));

  // We need these for updating the stencil buffer.
  float Dright  = depthTex.SampleLevel(PointSampler,
                                      texcoord, 0, int2(1, 0));
  float Dbottom = depthTex.SampleLevel(PointSampler,
                                      texcoord, 0, int2(0, 1));

  float4 delta = abs(D.xxxx -
                     float4(Dleft, Dtop, Dright, Dbottom));
  float4 edges = step(threshold.xxxx, delta);

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

  return edges;
}
```

Listing 4.1. Edge detection shader.

4.3 Obtaining Blending Weights

In order to calculate the blending weights we first search for the distances to the ends of the line the edgel belongs to, using the edges texture obtained in the previous pass (see Section 4.3.1). Once these distances are known, we can use them to fetch the crossing edges at both ends of the line (see Section 4.3.2). These crossing edges indicate the type of pattern we are dealing with. The distances to the ends of the line and the type of pattern are used to access the precalculated texture (see Section 4.3.3) in which we store the areas that are used as blending weights for the final pass.

As mentioned before, to share calculations between adjacent pixels, we take advantage of the fact that two adjacent pixels share the same boundary, and

```
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.
    float2 d = float2(SearchXLeft(texcoord),
                      SearchXRight(texcoord));

    // Instead of sampling between edges, 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;
    weights.rg = Area(abs(d), e1, e2);
  }

  [branch]
  if (e.r) { // Edge at west.
    float2 d = float2(SearchYUp(texcoord),
                      SearchYDown(texcoord));

    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;
    weights.ba = Area(abs(d), e1, e2);
  }

  return weights;
}
```

Listing 4.2. Blending weights calculation shader.

we perform 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 $(1, 1)$, respectively, in Figure 4.2). The one exception to this is the case in which the pixel lies at the middle of a line of odd length (as pixel $(2, 1)$ in Figure 4.2); in this case both the actual pixel and its opposite have a nonzero value for a. As a consequence, the output of this pass is a texture which, for each pixel, stores the areas at each side of its corresponding edges (by *the areas at each side* we refer to those of the actual pixel and its opposite). This yields two values for north edges and two values for west edges in the final blending weights texture. Finally, the weights stored in this texture will be used in the third pass to perform the final blending. Listing 4.2 shows the source code of this pass; Figure 4.1 (center right) is the resulting image.

4.3.1 Searching for Distances

The search for distances to the ends of the line is performed using an iterative algorithm, which in each iteration 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—it simply encodes whether an edgel exists—and query from positions between pixels using bilinear filtering for fetching two pixels at a time (see Figure 4.3). 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. We stop the search if the returned value is lower than one.[2] By using a simple approach like this, we are introducing two sources of inaccuracy:

1. 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;

2. When the returned value is 0.5 we cannot distinguish which of the two pixels contains an edgel.

Although an error is introduced in some cases, it is unnoticeable in practice—the speed-up is considerable since it is possible to jump two pixels per iteration. Listing 4.3 shows one of the distance search functions.

In order to make the algorithm practical in a game environment, we limit the search to a certain distance. As expected, the greater the maximum length, the better the quality of the antialiasing. However, we have found that, for the majority of cases, distance values between 8 and 12 pixels give a good trade-off between quality and performance.

[2] In practice we use 0.9 due to bilinear filtering precision issues.

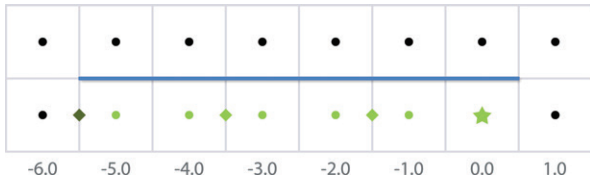

Figure 4.3. Hardware bilinear filtering is used when searching for distances from each pixel to the end of the line. The color of the dot at the center of each pixel represents the value of that pixel in the edges texture. In the case shown here, distance search of the left end of the line is performed for the pixel marked with a star. Positions where the edges texture is accessed, fetching pairs of pixels, are marked with rhombuses. This allows us to travel twice the distance with the same number of accesses.

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 the performance is affected negatively.

```
float SearchXLeft(float2 texcoord) {
  texcoord -= float2(1.5, 0.0) * PIXEL_SIZE;
  float e = 0.0;
  // We offset by 0.5 to sample between edges, 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 return
  // -2 * maxSearchSteps.
  return max(-2.0 * i - 2.0 * e, -2.0 * maxSearchSteps);
}
```

Listing 4.3. Distance search function (search in the left direction case).

Figure 4.4. Examples of the four possible types of crossing edge and corresponding value returned by the bilinear query of the edges texture. The color of the dot at the center of each pixel represents the value of that pixel in the edges texture. The rhombuses, at a distance of 0.25 from the center of the pixel, indicate the sampling position, while their color represents the value returned by the bilinear access.

4.3.2 Fetching Crossing Edges

Once the distances to the ends of the line are calculated, they are used to obtain the crossing edges. A naive approach for fetching the crossing edge of an end of a line would be to query two edges. A more efficient approach is to use bilinear filtering for fetching both edges at one time, in a manner similar to the way the distance search is done. 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 edges is present. Figure 4.4 shows the crossing edge that corresponds to each of the different values returned by the bilinear query.

4.3.3 The Precomputed Area Texture

With distance and crossing edges information at hand, we now have all the required inputs to calculate the area corresponding to the current pixel. As this is an expensive operation, we opt to precompute it in a four-dimensional table which is stored in a conventional two-dimensional texture (see Figure 4.2 (right)).[3] This texture is divided into subtextures of size 9×9, each of them corresponding to a pattern type (codified by the fetched *crossing edges* $e1$ and $e2$ at each end of the line). Inside each of these subtextures, (u, v) coordinates correspond to distances to the ends of the line, eight being the maximum distance reachable. Resolution can be increased if a higher maximum distance is required. See Listing 4.4 for 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

[3] The code to generate this texture is available in the web material.

```
#define NUM_DISTANCES 9
#define AREA_SIZE (NUM_DISTANCES * 5)

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

Listing 4.4. Precomputed area texture access function.

allows for a simpler and faster indexing. The `round` instruction is used to avoid possible precision problems caused by the bilinear filtering.

Following the same reasoning (explained at the beginning of the section) by 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, one for a pixel and one for its opposite. (Again, a will be zero for one of them in all cases with the exception of those pixels centered on lines of odd length.)

4.4 Blending with the Four-Neighborhood

In this last pass, the final color of each pixel is obtained by blending the actual color with its four neighbors, according to the area values stored in the weights texture obtained in the previous pass. This is achieved by accessing three positions of the blending weights texture:

1. the current pixel, which gives us the north and west blending weights;
2. the pixel at the south;
3. the pixel at the east.

Once more, to exploit hardware capabilities, we use four bilinear filtered accesses to blend the current pixel with each of its four neighbors. Finally, as one pixel can belong to four different lines, we find an average of the contributing lines. Listing 4.5 shows the source code of this pass; Figure 4.1 (right) shows the resulting image.

```
float4 NeighborhoodBlendingPS(
        float4 position: SV_POSITION,
        float2 texcoord: TEXCOORD0): SV_TARGET {
  float4 topLeft = blendTex.SampleLevel(PointSampler,
                                        texcoord, 0);
  float right    = blendTex.SampleLevel(PointSampler,
                                        texcoord, 0,
                                        int2(0, 1)).g;
  float bottom   = blendTex.SampleLevel(PointSampler,
                                        texcoord, 0,
                                        int2(1, 0)).a;
  float4 a = float4(topLeft.r, right, topLeft.b, bottom);

  float sum = dot(a, 1.0);

  [branch]
  if (sum > 0.0) {
    float4 o = a * PIXEL_SIZE.yyxx;
    float4 color = 0.0;
    color = mad(colorTex.SampleLevel(LinearSampler,
                texcoord + float2( 0.0, -o.r), 0), a.r, color);
    color = mad(colorTex.SampleLevel(LinearSampler,
                texcoord + float2( 0.0,  o.g), 0), a.g, color);
    color = mad(colorTex.SampleLevel(LinearSampler,
                texcoord + float2(-o.b,  0.0), 0), a.b, color);
    color = mad(colorTex.SampleLevel(LinearSampler,
                texcoord + float2( o.a,  0.0), 0), a.a, color);
    return color / sum;
  } else {
    return colorTex.SampleLevel(LinearSampler, texcoord, 0);
  }
}
```

Listing 4.5. Four-neighborhood blending shader.

4.5 Results

Qualitywise, our algorithm lies between $4x$ and $8x$ MSAA, requiring a memory consumption of only $1.5x$ the size of the backbuffer on a PC and of $2x$ on Xbox 360.[4] Figure 4.5 shows a comparison between our algorithm, $8x$ MSAA, and no antialiasing at all on images from Unigine Heaven Benchmark. A limitation of our algorithm with respect to MSAA is the impossibility of recovering subpixel

[4]The increased memory cost in the Xbox 360 is due to the fact that two-channel render targets with 8-bit precision cannot be created in the framework we used for that platform, forcing the usage of a four-channel render target for storing the edges texture.

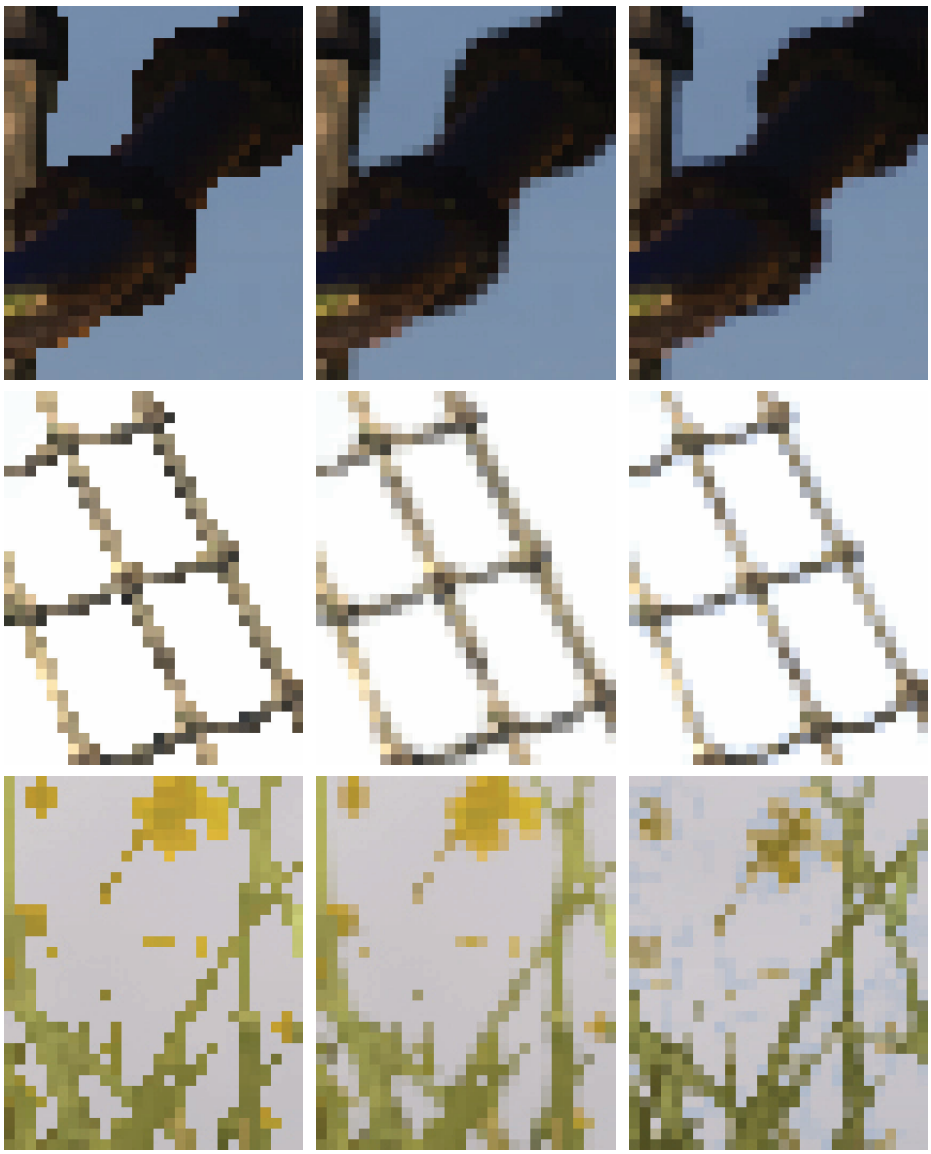

Figure 4.5. Examples of images without antialiasing, processed with our algorithm, and with $8x$ MSAA. Our algorithm offers similar results to those of $8x$ MSAA. A special case is the handling of alpha textures (bottom row). Note that in the grass shown here, alpha to coverage is used when MSAA is activated, which provides additional detail, hence the different look. As the scene is animated, there might be slight changes in appearance from one image to another. (Images from Unigine Heaven Benchmark courtesy of Unigine Corporation.)

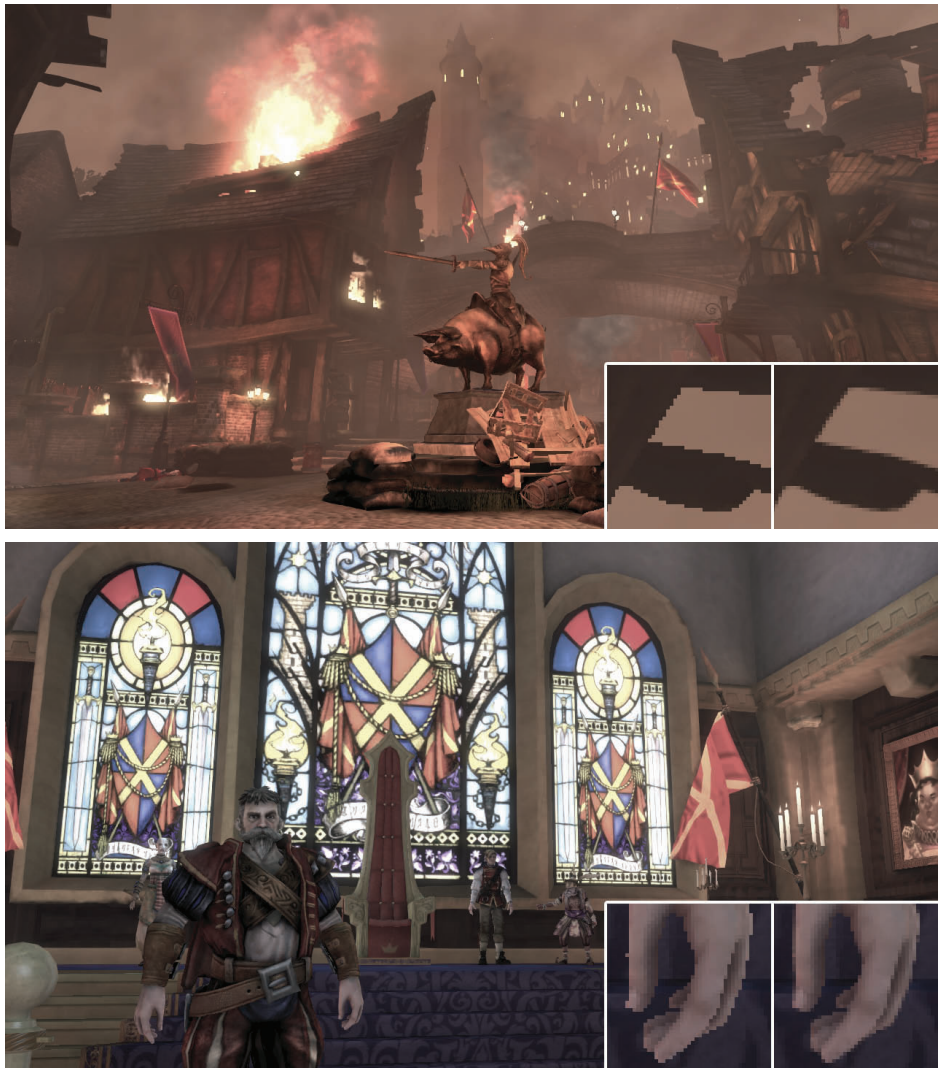

Figure 4.6. Images obtained with our algorithm. Insets show close-ups with no antialiasing at all (left) and processed with our technique (right). (Images from *Fable III* courtesy of Lionhead Studios.)

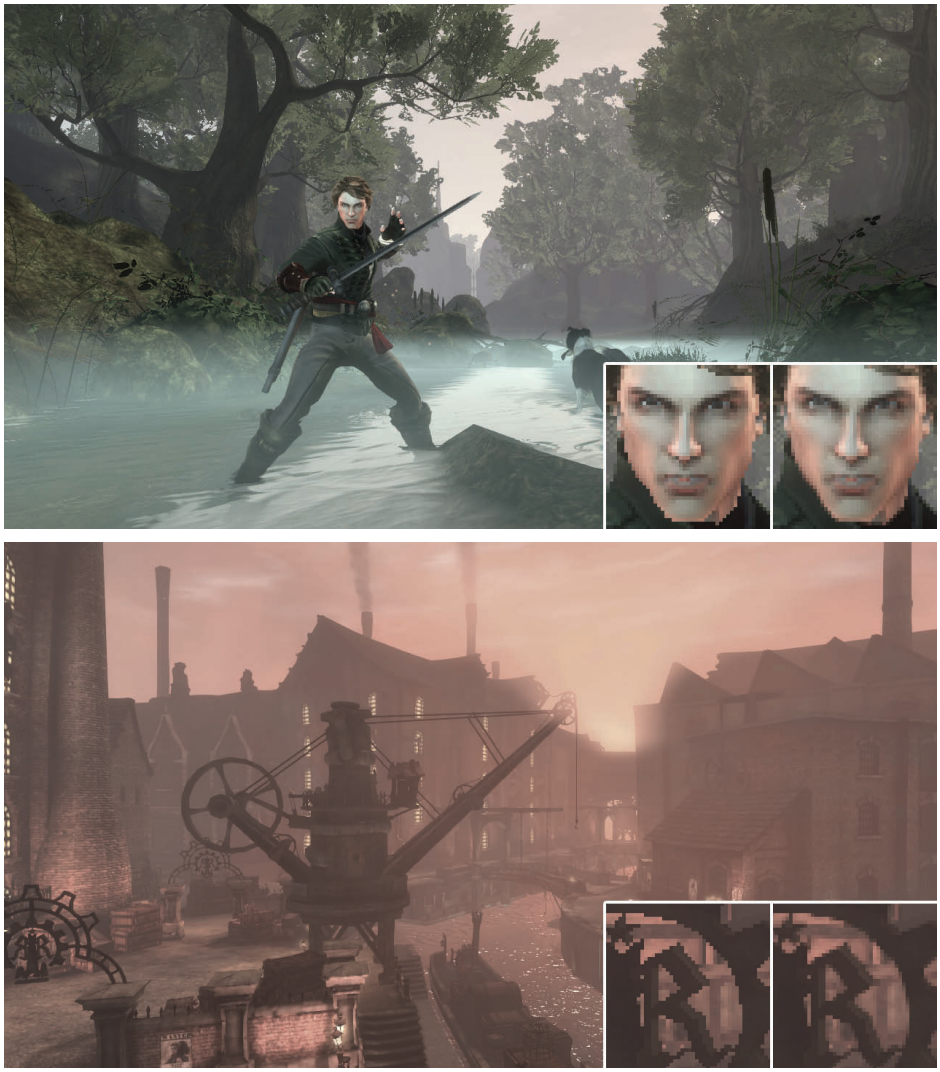

Figure 4.7. More images showing our technique in action. Insets show close-ups with no antialiasing at all (left) and processed with our technique (right). (Images from *Fable III* courtesy of Lionhead Studios.)

	Xbox 360		GeForce 9800 GTX+		
	Avg.	Std. Dev.	Avg.	Std. Dev.	Speed-up
Assassin's Creed	4.37 ms	0.61 ms	0.55 ms	0.13 ms	$6.31x^\star$
Bioshock	3.44 ms	0.09 ms	0.37 ms	0.00 ms	n/a
Crysis	3.92 ms	0.10 ms	0.44 ms	0.02 ms	$14.80x$
Dead Space	3.65 ms	0.45 ms	0.39 ms	0.03 ms	n/a
Devil May Cry 4	3.46 ms	0.34 ms	0.39 ms	0.04 ms	$5.75x$
GTA IV	4.11 ms	0.23 ms	0.47 ms	0.04 ms	n/a
Modern Warfare 2	4.38 ms	0.80 ms	0.57 ms	0.17 ms	$2.48x^\star$
NFS Shift	3.54 ms	0.35 ms	0.42 ms	0.04 ms	$14.84x$
Split/Second	3.85 ms	0.27 ms	0.46 ms	0.05 ms	n/a
S.T.A.L.K.E.R.	3.18 ms	0.05 ms	0.36 ms	0.01 ms	n/a
Grand Average	3.79 ms	0.33 ms	0.44 ms	0.05 ms	$11.80x$

Table 4.1. Average times and standard deviations for a set of well-known commercial games. A column showing the speed-up factor of our algorithm with respect to $8x$ MSAA is also included for the PC/DirectX 10 implementation. Values marked with * indicate $4x$ MSAA, since $8x$ was not available, and the grand average of these includes values only for $8x$ MSAA.

features. Further results of our technique, on images from *Fable III*, are shown in Figures 4.6 and 4.7. Results of our algorithm in-game are available in the web material.

As our algorithm works as a post-process, we have run it on a batch of screenshots of several commercial games in order to gain insight about its performance in different scenarios. Given the dependency of the edge detection on image content, processing times are variable. We have noticed that each game has a more or less unique "look-and-feel," so we have taken a representative sample of five screenshots per game. Screenshots were taken at 1280×720 as the typical case in the current generation of games. We used the slightly more expensive luminance-based edge detection, since we did not have access to depth information. Table 4.1 shows the average time and standard deviation of our algorithm on different games and platforms (Xbox 360/DirectX 9 and PC/DirectX 10), as well as the speed-up factor with respect to MSAA. On average, our method implies a speed-up factor of $11.80x$ with respect to $8x$ MSAA.

4.6 Discussion

This section includes a brief compilation of possible alternatives that we tried, in the hope that it would be useful for programmers employing this algorithm in the future.

Edges texture compression. This is perhaps the most obvious possible optimization, saving memory consumption and bandwidth. We tried two different alternatives: a) using 1 bit per edgel, and b) separating the algorithm into a vertical and a horizontal pass and storing the edges of four consecutive pixels in the RGBA

channels of each pixel of the edges texture (vertical and horizontal edges separately). This has two advantages: first, the texture uses less memory; second, the number of texture accesses is lower since several edges are fetched in each query. However, storing the values and—to a greater extent—querying them later, becomes much more complex and time consuming, given that bitwise operations are not available in all platforms. Nevertheless, the use of bitwise operations in conjunction with edges texture compression could further optimize our technique in platforms where they are available, such as DirectX 10.

Storing crossing edges in the edges texture. Instead of storing just the north and west edges of the actual pixel, we tried storing the crossing edges situated at the left and at the top of the pixel. The main reason for doing this was that we could spare one texture access when detecting patterns; but we realized that by using bilinear filtering we could also spare the access, without the need to store those additional edges. The other reason for storing the crossing edges was that, by doing so, when we searched for distances to the ends of the line, we could stop the search when we encountered a line perpendicular to the one we were following, which is an inaccuracy of our approach. However, the current solution yields similar results, requires less memory, and processing time is lower.

Two-pass implementation. As mentioned in Section 4.1, a two-pass implementation is also possible, joining the last two passes into a single pass. However, this would be more inefficient because of the repetition of calculations.

Storing distances instead of areas. Our first implementation calculated and stored only the distances to the ends of the line in the second pass, and they were then used in the final pass to calculate the corresponding blending weights. However, directly storing areas in the intermediate pass allows us to spare calculations, reducing execution time.

4.7 Conclusion

In this chapter, we have presented an algorithm crafted for the computation of antialiasing. Our method is based on three passes that detect edges, determine the position of each pixel inside those image features, and produce an antialiased result that selectively blends the pixel with its neighborhood according to its relative position within the line it belongs to. We also take advantage of hardware texture filtering, which allows us to reduce the number of texture fetches by half.

Our technique features execution times that make it usable in actual game environments, and that are far shorter than those needed for MSAA. The method presented has a minimal impact on existing rendering pipelines and is entirely implemented as an image post-process. Resulting images are between $4x$ and

$8x$ MSAA in quality, while requiring a fraction of their time and memory consumption. Furthermore, it can antialias transparent textures such as the ones used in alpha testing for rendering vegetation, whereas MSAA can smooth vegetation only when using alpha to coverage. Finally, when using luminance values to detect edges, our technique can also handle aliasing belonging to shading and specular highlights.

The method we are presenting solves most of the drawbacks of MSAA, which is currently the most widely used solution to the problem of aliasing; the processing time of our method is one order of magnitude below that of $8x$ MSAA. We believe that the quality of the images produced by our algorithm, its speed, efficiency, and pluggability, make it a good choice for rendering high quality images in today's game architectures, including platforms where benefiting from antialiasing, together with outstanding techniques like deferred shading, was difficult to achieve. In summary, we present an algorithm which challenges the current gold standard for solving the aliasing problem in real time.

4.8 Acknowledgments

Jorge would like to dedicate this work to his eternal and most loyal friend Kazán. The authors would like to thank the colleagues at the lab for their valuable comments, and Christopher Oat and Wolfgang Engel for their editing efforts and help in obtaining images. Thanks also to Lionhead Studios and Microsoft Games Studios for granting permission to use images from *Fable III*. We are very grateful for the support and useful suggestions provided by the Fable team during the production of this work. We would also like to express our gratitude to Unigine Corporation, and Denis Shergin in particular, for providing us with images and material for the video (available in the web material) from their Unigine Heaven Benchmark. This research has been funded by a Marie Curie grant from the 7^{th} Framework Programme (grant agreement no.: 251415), the Spanish Ministry of Science and Technology (TIN2010-21543) and the Gobierno de Aragón (projects OTRI 2009/0411 and CTPP05/09). Jorge Jimenez and Belen Masia are also funded by grants from the Gobierno de Aragón.

Bibliography

[Koonce 07] Rusty Koonce. "Deferred Shading in Tabula Rasa." In *GPU Gems 3*, pp. 429–457. Reading, MA: Addison Wesley, 2007.

[Reshetov 09] Alexander Reshetov. "Morphological Antialiasing." In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pp. 109–116. New York: ACM, 2009. Available online (http://visual-computing.intel-research.net/publications/papers/2009/mlaa/mlaa.pdf).

[Shishkovtsov 05] Oles Shishkovtsov. "Deferred Shading in S.T.A.L.K.E.R." In *GPU Gems 2*, pp. 143–166. Reading, MA: Addison Wesley, 2005.

[Sousa 07] Tiago Sousa. "Vegetation Procedural Animation and Shading in Crysis." In *GPU Gems 3*, pp. 373–385. Reading, MA: Addison Wesley, 2007.

[Thibieroz 09] Nicolas Thibieroz. "Deferred Shading with Multisampling Anti-Aliasing in DirectX 10." In *ShaderX*^{7}, pp. 225–242. Hingham, MA: Charles River Media, 2009.