MORPHOLOGICAL ANTIALIASING AND TOPOLOGICAL RECONSTRUCTION

Keywords: antialiasing, mlaa, topology, realtime, gpu.

Abstract: Morphological antialiasing is a post-processing approach which does note require additional samples computation. This algorithm acts as a non-linear filter, ill-suited to massively parallel hardware architectures. We redesigned the initial method using multiple passes with, in particular, a new approach to line length computation. We also introduce in the method the notion of topological reconstruction to correct the weaknesses of postprocessing antialiasing techniques. Our method runs as a pure post-process filter providing full-image antialiasing at high framerates, competing with traditional MSAA.

1 INTRODUCTION

In computer graphics, rendering consists in sampling the color of a virtual scene. The mathematical properties of the sampling operation imply that rendering is inherently bond to aliasing artifacts. Therefore, the subject of antialiasing techniques has been actively explored for the past forty years (Catmull, 1978).

Traditional approaches to handle aliasing involve computing multiple samples for each final sample. Graphics hardware vendors implement various refinements of these algorithms (Akeley, 1993; Schilling, 1991). However antialiasing by supersampling and its refinements for exemple MultiSamplge AntiAliasing (MSAA), can be prohibitively costly notably in terms of memory. This particularity is prohibitive on consoles, therefore a great number of games ship without any antialiasing.

Supersampling, as a geometric technique, does not cope well with image-based lighting techniques such as deferred rendering (Shishkovtsov, 2005) where the rasterization is completely separated from the lighting. Due to the popularity of these lighting approaches in recent real-time rendering engines, a number of filter-based antialiasing techniques appeared. Morphological Antialiasing (Reshetov, 2009) (MLAA) is a relatively recent technique which enables full image antialiasing as a post-process. Figure 1 compares two images corrected by respectively MSAA and MLAA with similar results. However MLAA has its own caveats. As a pure image-based techniques it can not handle subpixel geometry aliasing. Moreover, as a non-linear filter, the technique uses proficiently deep branching and image-wise knowledge thus is a poor candidate to a naive GPU implementation.

We present in this paper both a practical real-time implementation of the MLAA on the GPU and a image-based technique to simulate subpixel geometry antialiasing. Our GPU MLAA implementation is a complete redesign of the original algorithm and takes full advantage of hardware acceleration. We improve results of MLAA on undersampled small scale geometric details. Our algorithm is able to locally reconstruct subpixel missing data. Our method runs as a pure post-process filter providing full-image antialiasing at high framerates.

2 ANTIALIASING TECHNIQUES

2.1 Supersampling

Antialiasing is a widely studied topic in computer graphics from the very start (Catmull, 1978). An-

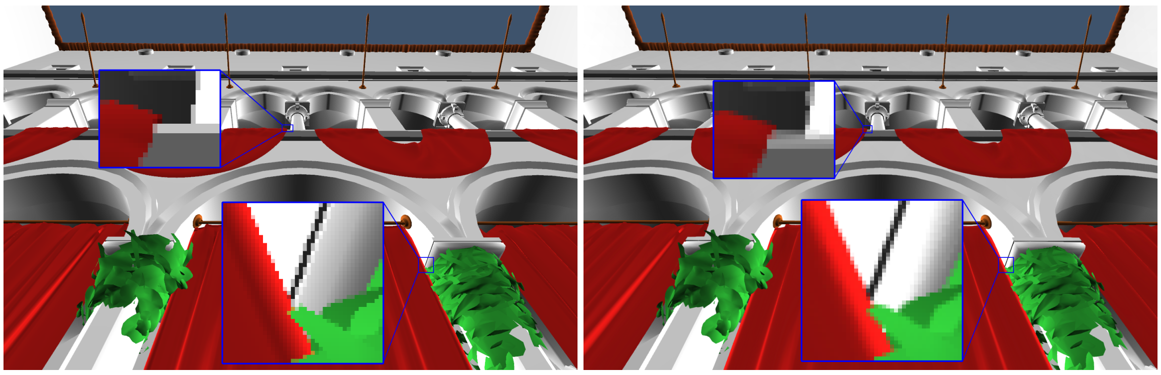

Figure 1: Left : Antialiasing with MSAA 4x. Right : with MLAA

tialiasing is built in most graphics hardware thanks to refinements of the supersampling technique. In this technique, named SSAA for Supersampling AntiAliasing, rendering is done at higher resolution then undersampled to produce the final image. Though this technique gives good results, it can be particularly costly in terms of rasterization, shading and memory. Framerates can decrease rapidly in case of complex shading or raytracing.

The most common hardware supersampling approach is multisampling or MSAA (Akeley, 1993; Schilling, 1991). In the naive supersampling, each sample is shaded, therefore a pixel is shaded multiple times. MSAA decreases the cost by decoupling fragment shading and sampling. For each sample in a pixel, the covering triangle is stored, shading is then performed once for each covering triangle then pixel color is extrapolated. CSAA or Coverage Sampling Antialiasing reduces even more the number of shaded samples by handling separately primitive coverage, depth, stencil and color. Amortized Supersampling (Yang et al., 2009) uses temporal coherence between frames to reduce computed samples but is limited to procedural shading.

Multisampling techniques are still tightly coupled to fixed parts of the graphic pipeline, but fixed functions tend to disappear. A recent technique by (Iourcha et al., 2009) takes advantage of new hardware MSAA capabilities to acquire more accurate coverage values using neighbors of samples.

In deferred shading(Koone, 2007), the shading stage is performed in image-space on G-Buffers. Therefore using hardware multisampling with deferred shading is usually considered challenging (Engel, 2009).

2.2 Image-based antialiasing

In order to totally remove the dependency to geometry, it is possible to directly filter the final image. Full-image filtering offers the advantage of handling triangle coverage aliasing as well as shadows or textures indifferently. With the recent gain in popularity of deferred shading techniques and the limited MSAA capabilities of the current generation of game console these approaches flourished. The combination of edge detection and blur is a common and inexpensive technique (Shishkovtsov, 2005), however the results contain lots of false positive resulting generally in overblurring. In (Lau, 2003), 5x5 masks are used to detects patterns to blend.

MLAA (Reshetov, 2009) is an image-based algorithm providing full-image antialiasing independently of the rendering pipeline. Therefore it can be used in rasterization as well as in raytracing. Moreover it is perfectly adapted to deferred rendering techniques. The initial algorithm is CPU based and was successfully ported to SPU for the PS3 console (Hoffman, 2010) with results comparable to MSAA4x.

The algorithm can be coarsely divided in two passes. First we detect discontinuity segments along lines and columns of the image. Crossing segments are then detected and form L shapes of varying orientations and sizes. Then the pixels of the frame are blended with their opposite neighbor along theses shapes (cf. figure 2). Concerned samples are those contained in the L, they are covered by a trapeze which is a sub-part of the triangle formed by the L. Blending weight depends of the trapeze area.

Though this algorithm is naively parallelizable it is ill-suited to massively parallel architectures such as GPU. Indeed L shapes detection imply deep dynamic branching and quasi-random access patterns in the frame which cripples cache efficiency (Nvidia, 2007). Moreover, as a post-processing technique, MLAA is

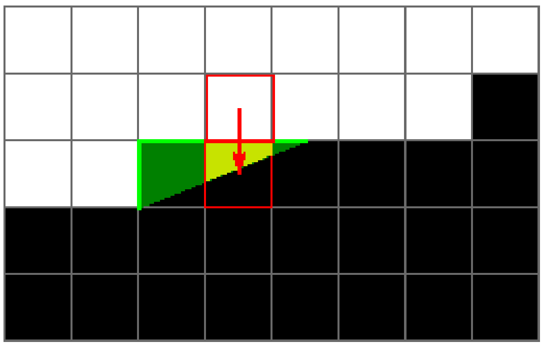

Figure 2: In MLAA, the bottom red pixel blends with the top red pixel weighted by the area of the yellow trapeze. The technique consists in detecting L shapes in green and blending pixels along the green triangle with pixels opposing the L.

unable to handle small scale geometry aliasing, and as a relatively local filter can suffer from poor temporal coherence, which induces flickering artifacts in animation. Finally, using luminance for discontinuity computation can lead to poor quality on RGB rendering.

3 MLAA ON THE GPU

3.1 Overview

The initial algorithm was entirely redesigned to fit with a GPU GLSL implementation thus allowing direct usage in any rendering engine. GPU detection and storage of L shapes can be avoided since the main objective is to compute the area of the covering trapeze on each pixel along discontinuity lines. This area can be computed using the pixel position on the L shape. Our algorithm will therefore detect discontinuity segments, determine relative positions of pixels along these segments, compute covering trapeze areas and then do the final blending. The figure 3 shows an overview of the algorithm and samples for each of these stages.

3.2 Discontinuities detection

In this stage, we seek color discontinuities between two neighboring pixels. For greyscale images, we detect discontinuity presence if the difference between greyscale values exceeds a user defined value called the discontinuity factor, chosen between 0. and 1. For color images we could use this factor as a threshold in the luminance of the pixels. However in RGB, red and blue share the same luminance leading to aliasing artifacts on textures. Therefore we switch to CIELAB color space where we are able to compute a color difference directly based on human perception (Kang, 2006). This discontinuity detection pass creates a texture containing, for each pixel, the existence of a discontinuity at its bottom and its right border. These two boolean are stored in two different texture channels as show in Algorithm 1

We also use this first stage to initialize the texture storing discontinuity line lengths. We simply write 1 to the left and right distance of any pixel belonging to horizontal lines and 1 to up and down distance to the ones belonging to vertical lines. This is done simply using MRT.

3.3 Distance propagation

In this stage, we want to compute the distance in number of pixel, from any pixel along discontinuity lines to their two extremities. In fact, we will propagate distance from the extremities. This is done in left and right direction for horizontal lines and on up and down directions for vertical ones. Adding these two lengths also gives us the total length of the horizontal (resp vertical) discontinuity line minus one. And of course, it gives us the position of the pixel relatively

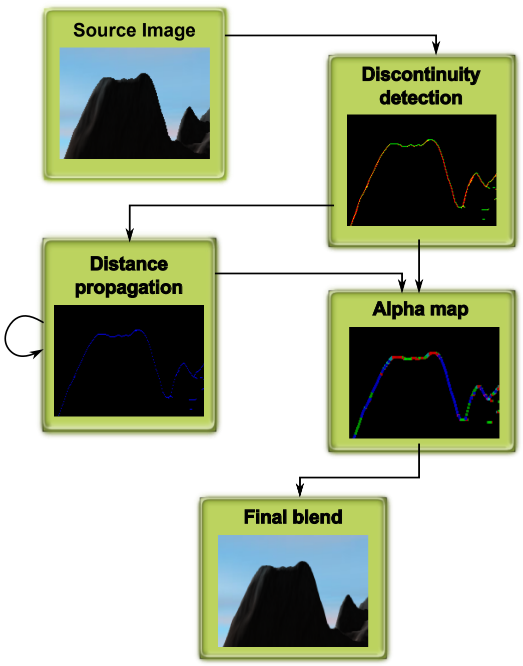

Figure 3: Algorithm overview

Algorithm 1: Pseudo code of shader for discontinuity detection

```
Data: basetex : source image
Data: texCoord : tex coord of fragment
Data: discFct : threshold
Result: discontinuity on bottom and right pixel
Result: init distance texture
1  begin
2      vec4 col = texture2D(basetex,texCoord.st);
3      glFragData[0] = vec4(0.);
4      glFragData[1] = vec4(0.);
5      vec3 colLab = XYZtoLAB(RGBtoXYZ * col.rgb);
       /* Test right */
6      vec3 colRight = texture2D(basetex,rightTexCoord[0].xy);
7      vec3 colCompLab= XYZtoLAB(RGBtoXYZ * colRight);
8      if distanceLAB(colLab,colCompLab) > discFct then
9          glFragData[0].r = 1.;
10         glFragData[1].ba = vec2(1./255.);
       /* Test bottom */
11     colCompLab = XYZtoLAB(RGBtoXYZ * texture2D(basetex,upTexCoord[1].zw).rgb);
12     if distanceLAB(colLab,colCompLab) > discFct then
13         glFragData[0].g = 1.0 ;
14         glFragData[1].rg = vec2(1./255.);
```

to this line which will be useful in the next step. All lengths will be computed in pixel distance.

To propagate the distance to the extremities of lines, we will use a modification of the recursive doubling approach (P and Rodrigue, 1977; Hensley et al., 2005). The computation will be done in up to four steps. The first one will compute length distance up to four pixels, the second one up to 16, the third one up to 64 and the final one, up to 256. At each step, for a given pixel X, we look at its distance $d(X)$ (for instance for left direction), fetch distance of the texel at $d(X)$ distance (at left) and add it to the current distance $d(X)$. Updating the distance is given by :

$$d(X) := d(X) + d(X.x - d(X).x, X.y) \qquad (1)$$

We do the same for right, up and down direction. We also do that three times at each step. Figure 4 illustrates computation of left distance for the two first steps of this stage. The reason we split the computation in several step is that, you can check, for each step and pixel, if the pixel's distance is below the maximum length we can obtain from the previous step. If

Algorithm 2: Pseudo code of shader for distance computations

```
Data: texLgth : previous length tex
Data: texCoord : tex coord of fragment
Data: vec2 : coeff_image.zw : 255./size of image.xy
Result: new length tex
1  begin
2      vec4 curLength;
3      curLength = texture2D(texLgth,texCoord);
4      vec4 curDelta = curLength;
5      vec2 curTCoord;
6      float oneDelta;
7      curDelta *= curDelta.zzww;
       /* Left propagation */
8      if (curLength.r ≥ threshold) then
           /* Scan left (3 times) */
9          curTCoord = owncoordinates.st - vec2(curDelta.r,0.);
10         for ($k = 0; k < 3; ++k$) do
11             oneDelta = texture2D(texLength, curTCoord).r;
12             curLength.r += oneDelta;
13             curTCoord.x -= oneDelta * curDelta.z;
       /* Do the same for right, up and down */
```

so, the pixel distance is already computed and can be discarded.

Since distance will be stored in 8-bit precision, we will restrict ourselves to 255 line length (in pixel). The interest is that the four distance can be stored in a regular RGBA texture. You need two such textures in order to update distance, one texture for reading (in the shader), and one for writing.

3.4 Computing blending factors

To this point we have computed the discontinuity segments, their lengths and the relative position of any pixel along these lines. Therefore, for each pixel bordering a discontinuity segment, we need to identify which type of L shape it belongs to. Each pixel can belong to up to four L shapes and will be blended accordingly. For an identified L shape, the area of the trapeze A can be precomputed and depends from segment length L in pixels, and from the relative position p of the pixel in the segment. Trapeze area can be

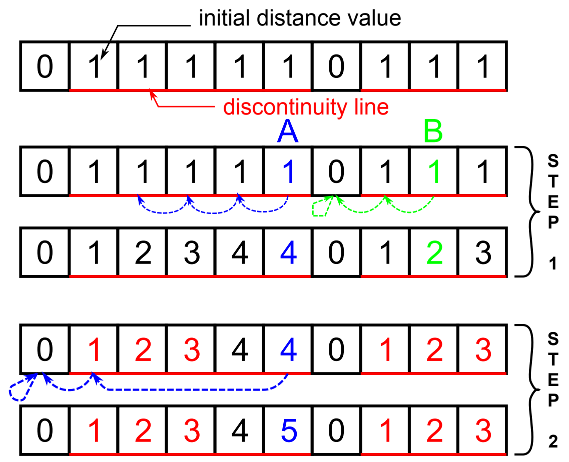

Figure 4: Overview of the recursive doubling approach to compute the length of discontinuity lines. Example here shows the computation of the relative distance to the left horizontal end line. In the first step, pixel A checks its three closest neighbor, each one making him "jump" to the left. Pixel B checks its closest left neighbor then its left pixel which does not belong to a discontinuity line (distance is zero then). So the computation will add 0 for the two last times. In the second step, pixel B has a distance lesser than 4, so it is discarded as all pixels marked in red. Pixel A will check the fourth - its distance - left neighbor, add one, check the left neighbor and stop here.

computed by the formula :

$$A = \frac{1}{2}\left(1 - \frac{2*p+1}{L}\right) \tag{2}$$

This equation is true if $p < L/2$, else we have a triangle. If L is odd, the area of this triangle is $1/(2L)$, else it is $1/(8L)$. Therefore, given the relative position computed in previous step, and the length of the L shape, we can fetch the blending factor in a precomputed 2D texture. Given that a pixel may belong to up to for L shapes, four blending factors are stored, one for each neighbor, on a RGBA texture which is used in the final step.

3.5 Blending

At the end of the previous step, each pixel of the textures contains blending factors with its four neighbors. Therefore the blending consists in using them as local convolution kernels.

4 TOPOLOGICAL RECONSTRUCTION

An inherent problem to image based antialiasing algorithm are their incapacity to "regenerate" geometry in case of wrong sampling as in figure 5. We present in this section a reconstruction technique for this geometry, particularly in simple but frequent cases where only one sample is missing. We use discrete geometry and topology approaches to compute, for each pixel, its connexity number (Kong and Rosenfeld, 1989) and a characterization of its neighborhood. Using these properties it is possible to determine whether a pixel could reconnect two homogeneous areas and then "reconstruct" missing geometry.

4.1 Neighborhood and connexity

In the case of binary images, we remind some elements of digital topology (Kong and Rosenfeld, 1989). A point $A \in \mathbb{Z}^2$ is defined as (A_1, A_2). We consider neighborhood relationships N_4 and N_8 defined by, for each point $A \in \mathbb{Z}^2$:

$$\begin{array}{l} N_4(A) = \left\{B \in \mathbb{Z}^2;\ |B1 - A1| + |B2 - A2| \leq 1\right\} \\ N_8(A) = \left\{B \in \mathbb{Z}^2;\ \max(|B1 - A1|, |B2 - A2|) \leq 1\right\} \end{array} \tag{3}$$

Let $\alpha \in \{4, 8\}$, we define $N^*_\alpha(A) = \frac{N_\alpha(A)}{A}$. A point B will be α-adjacent at point A if $B \in N^*_\alpha(A)$. An α-path is a sequence of points $A_0 \ldots A_k$ so that A_i is adjacent to A_{i-1} for $i = 1 \ldots k$.

Let $X \subseteq \mathbb{Z}^2$, two points A, B of X are α-connected in X if an α-path exists in X between these two points. This defines an equivalence relation. Equivalency classes for this relationship are α-connected components of X. A subset X of $\mathbb{Z}^2$ is said α-connected if it is constituted by exactly one α-connected component.

The set constituted by all α-connected components of X is noted $C_\alpha(X)$. A subset Y of $\mathbb{Z}^2$ is said α-adjacent to a point $A \in \mathbb{Z}^2$ if it exists a point $B \in Y$ adjacent to A. The set of α-connected components of X α-adjacent to point A is noted $C^A_\alpha(X)$. Formally, the connexity number for a point A in a subset X of $\mathbb{Z}^2$ are defined by :

$$T_\alpha(A, X) = \left|C^A_\alpha[N^*_8(A) \cap X]\right| \tag{4}$$

Using the number of connexities we can measure efficiently the number of connected components adjacent to a given pixel.

4.2 Image reconstruction

Our objective is to work on the particular case of a single pixel missing for small scale geometry as shown in

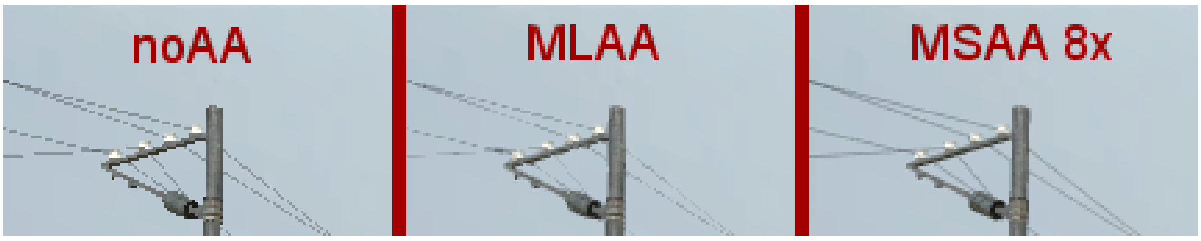

Figure 5: Comparison between no antialiasing (left), MLAA (middle) and MSAA8x (right). Missing subpixel data MLAA can't close holes in the electrical lines.

Figure 6. In this case, post-processing techniques due to missing data can not handle this artifact. Nevertheless, local image structure can indicate the probable presence of a missing element.

We work locally on each image pixel on a 3x3 mask. On each mask we identify potential shapes which are cut in two connected component by the current pixel. We focus on pixels linking the two connected components. The shape is constituted by pixels sharing the same color and different from the current pixel color. Reciprocally, the complement of the reconstructed shape has to be a unique connected component, thus sharing the same color as the current pixel, in order to avoid removing straight lines. If X is the shape to be reconstructed, we select pixels A so $T_8(A,X) = 2$ and $T_8(A,\overline{X}) = 1$. Therefore we can replace their color by the color of the shape. Figure 7 gives an overview of the results.

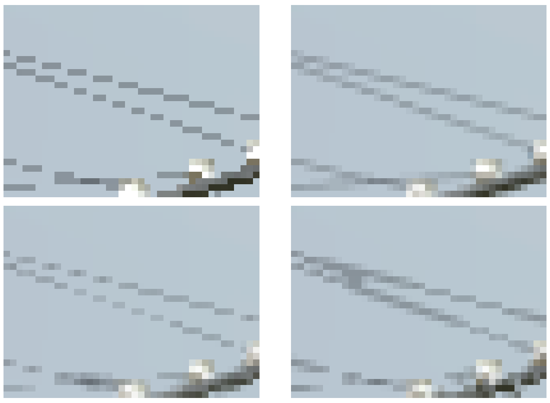
Figure 6: Detail of the reconstruction along electric lines. Top left, original image. Top right, MSAA8x. Bottom left, MLAA without reconstruction. Bottom right, MLAA with topological reconstruction.

4.3 Integration

The correction takes place before discontinuity lines detection. On the whole image, we compute the connexity number for each pixel. In order to do this, we compare the color in CIELAB space with its 8 neighbors. Each neighbor with a different color than the current pixel color is marked as belonging to X, the others are marked as belonging to the set $\overline{X}$. Using the gathered data, the number of connexities is fetched using a lookup-table. If the number of connexities $T_8(A,X) = 2$ and $T_8(A,\overline{X}) = 1$, the pixel is then replaced by the mean color of X pixels.

The Table 2 compares antialiasing techniques on a forward renderer. In this case MLAA is equivalent of MSAA4x.

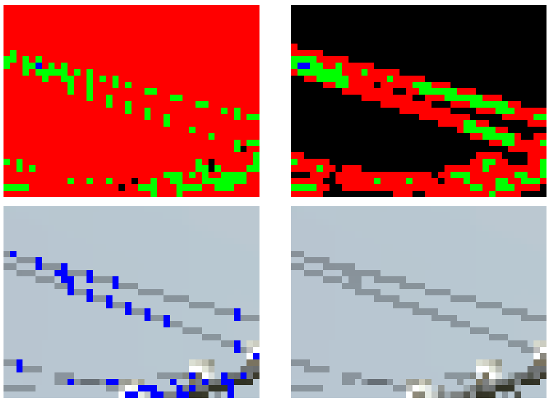
Figure 7: Top : connexity number for X and $\overline{X}$, black = 0, red = 1, green = 2, other > 2. Pixels are selected if $X \geq 2$ and $\overline{X} = 1$. Bottom left, selected pixels. Bottom right reconstructed pixels.

5 RESULTS

The algorithm was implemented on an Intel Core i7 920 with a NVidia GeForce GTX 295 graphic card. Measurements were done on a Sponza-like scene at resolution of 1920x1080. The Table 1 sums up the results in terms of frames per second on our deferred renderer. In this case MLAA performs relatively well against brute force SSAA both in terms of quality and performance cost.

Table 1: FPS and additional cost of antialiasing techniques on a deferred shading renderer.

Method	No AA	MLAA	SSAAx2
Rendering time (ms)	17.4	18.3	31.2
Additional cost (ms)	0	0.9	13.8

Table 2: FPS and additional costs of antialiasing methods in a forward renderer

Method	No AA	MSAAx2	MSAAx4	MLAA4
Rendering time (ms)	10.3	10.9	11.8	11.2
Additional cost	0	0.6	1.2	0.9

In terms of quality, in Figure 1 MLAA performs generally well against MSAA, however figure 5 illustrates a pathological case for MLAA. Small scale geometry details lead to holes between samples when the projected geometry is smaller than pixel size. In next session we present a technique to address this problem.

The topological reconstruction does have an additional cost of 0.55ms on our previously described testing configuration. The Figure 8 we show MLAA filtering with and without correction on the whole image.

The Figure 9 shows other use cases of our method given different scenes and shading. Our implementation is currently used on the in-house real-time rendering engine at DuranDuboi Studio notably for asset previewing.

6 CONCLUSION

In this paper, we presented a practical real-time implementation of the MLAA algorithm suited to the GPU. We introduced a new method using topological reconstruction to handle pathological cases for image-based antialiasing approaches. Our method improves behavior of these algorithms on small scale geometric details.

We intend to continue working on G-Buffer and MLAA integration, particularly how to use additional data in the G-Buffer such as depths and normals to improve general MLAA behavior. We also aim to improve the topological reconstruction to handle bigger gaps and to improve temporal coherence between frames.

REFERENCES

Akeley, K. (1993). Reality engine graphics. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 109–116, New York, NY, USA. ACM.

Catmull, E. (1978). A hidden-surface algorithm with anti-aliasing. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, page 11. ACM.

Engel, W. (2009). Deferred Shading with Multisampling Anti-Aliasing in DirectX 10. *in ShaderX7, W. Engel, Ed. Charles River Media, March.*

Hensley, J., Scheuermann, T., Coombe, G., Singh, M., and Lastra, A. (2005). Fast summed-area table generation and its applications. *Computer Graphics Forum*, 24(3):547–555.

Hoffman, N. (2010). Morphological Antialiasing in God of War III.

Iourcha, K., Yang, J., and Pomianowski, A. (2009). A directionally adaptive edge anti-aliasing filter. In *Proceedings of the 1st ACM conference on High Performance Graphics*, pages 127–133. ACM.

Kang, H. (2006). *Computational color technology*. Society of Photo Optical.

Kong, T. Y. and Rosenfeld, A. (1989). Digital topology: introduction and survey. *Comput. Vision Graph. Image Process.*, 48(3):357–393.

Koone, R. (2007). Deferred Shading in Tabula Rasa, GPU Gems 3, Nguyen, H.

Lau, R. (2003). An efficient low-cost antialiasing method based on adaptive postfiltering. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(3):247–256.

Nvidia, C. (2007). Compute Unified Device Architecture Programming Guide. *NVIDIA: Santa Clara, CA.*

P, D. and Rodrigue, G. (1977). An analysis of the recursive doubling algorithm. *High Speed Computer and Algorithm Organization*, pages 299–305.

Reshetov, A. (2009). Morphological antialiasing. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 109–116, New York, NY, USA. ACM.

Schilling, A. (1991). A new simple and efficient antialiasing with subpixel masks. *ACM SIGGRAPH Computer Graphics*, 25(4):141.

Shishkovtsov, O. (2005). Deferred shading in stalker. *GPU Gems*, 2 ch 9:143–166.

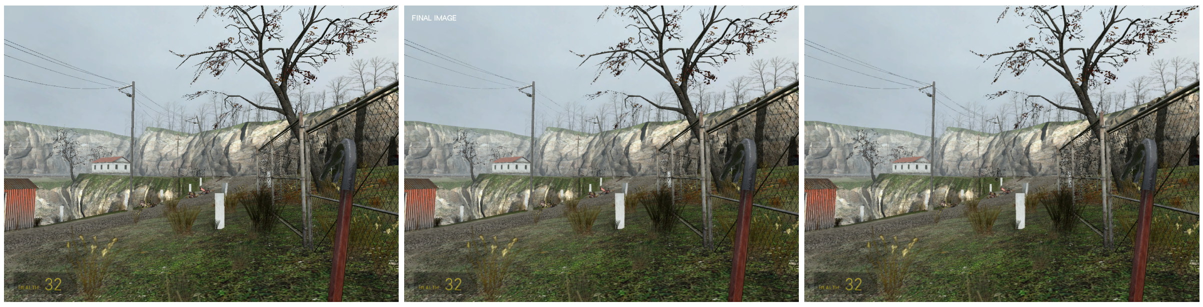

Figure 8: Original image (left), Standard MLAA (middle). MLAA with topological reconstruction (right)

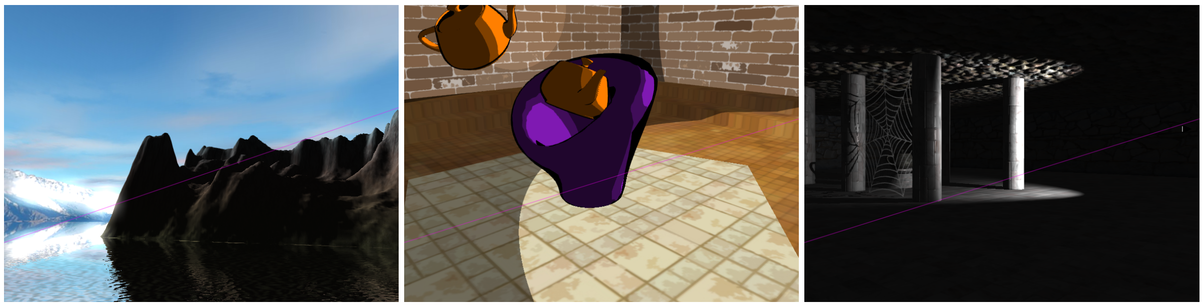

Figure 9: Various use case for MLAA : bloom, toon-shading, shadows

Yang, L., Nehab, D., Sander, P., Sitthi-amorn, P., Lawrence, J., and Hoppe, H. (2009). Amortized supersampling. In *ACM SIGGRAPH Asia 2009 papers*, pages 1–12. ACM.