MLAA: Efficiently Moving Antialiasing from the GPU to the CPU

Introduction
Efficient antialiasing techniques are an important tool of high-quality, real-time rendering. MSAA (Multisample Antialiasing) is the standard technique in use today, but comes with some serious disadvantages:

- Incompatibility with deferred lighting, which is used more and more in real-time rendering;
- High memory and processing cost, which makes its use prohibitive on some widely available platforms (such as the Sony Playstation* PS3* [Perthuis 2010]). This cost is also directly linked to the complexity of the scene rendered;
- Inability to smooth non-geometric edges unless used in conjunction with alpha to coverage.

A new technique developed by Intel Labs called Morphological Antialiasing (MLAA) [Reshetov 2009] addresses these limitations. MLAA is an image-based, post-process filtering technique which identifies discontinuity patterns and blends colors in the neighborhood of these patterns to perform effective antialiasing. It is the precursor of a new generation of real-time antialiasing techniques that rival MSAA [Jimenez et al., 2011] [SIGGRAPH 2011].

This sample is based on the original, CPU-based MLAA implementation provided by Reshetov, with improvements to greatly increase performance. The improvements are:

- Integration of a new, efficient, easy-to-use tasking system implemented on top of Intel® Threading Building Blocks (Intel® TBB).
- Integration of a new, efficient, easy to use pipelining system for CPU onloading of graphics tasks.
- Improvement of data access patterns through a new transposing pass.
- Increased use of Intel® SSE instructions to optimize discontinuities detection and color blending.

The MLAA algorithm
In this section we present an overview of how the MLAA algorithm works; cf. [Reshetov 2009] for the fully detailed explanation. Conceptually, MLAA processes the image buffer in three steps:

1. Find discontinuities between pixels in a given image.
2. Identify U-shaped, Z-shaped, L-shaped patterns.
3. Blend colors in the neighborhood of these patterns.
The first step (find discontinuities) is implemented by comparing each pixel to its neighbors. Horizontal discontinuities are identified by comparing a pixel to its bottom neighbor, and vertical discontinuities by comparing with the neighbor on the right. In our implementation we compare color values, but any other approach that suits the application’s specificities is perfectly valid.

At the end of the first step, each pixel is marked with the horizontal discontinuity flag and/or the vertical discontinuity flag, if such discontinuities are detected. In the next step, we “walk” the marked pixels to identify discontinuity lines (sequences of consecutive pixels marked with the same discontinuity flag), and determine how they combine to form L-shaped patterns, as illustrated below:

![Figure 1: MLAA processing of an image, with Z, U, and L-shapes shown on the original image on the left](image1.png)

The third and final step is to perform blending for each of the identified L-shaped patterns.

The general idea is to connect the middle point of the primary segment of the L-shape (horizontal green line in the figure below) to the middle point of the secondary segment (vertical green line—the connection line is in red). The connection line splits each pixel into two trapezoids; for each pixel the area of the corresponding trapezoid determines the blending weights. For example, in the figure below, we see that the trapezoid’s area for pixel c5 is 1/3; so the new color of c5 will be computed as 2/3 * (color of c5) + 1/3 * (color of c5’s bottom neighbor).

![Figure 2: Computing blending weights](image2.png)
In practice, to ensure a smooth silhouette look, we need to minimize the color differences at the stitching positions of consecutive L-shapes. To achieve this, we slightly adjust the connection points on the L-shape segments around the middle position based on the colors of the pixels around the stitching point.

**Sample usage**

The camera can be moved around the scene using drag-and-click, and the mouse wheel can be used to zoom in or out.

In addition, there are three blocks of controls on the right side of the sample’s window:

![Figure 3: A screenshot of the sample in action](image)

- **Pause Scene** toggles the scene’s animations on/off;
- **Show zoom box** toggles the zoom box feature on/off, which allows the user to take a closer look at a pixel area to better compare the antialiasing techniques. The area of interest can be changed by right-clicking to the new area to examine;
- **Scene complexity** simulates the effect of increasing scene complexity by using overdraw. The value (between 1 and 100, adjusted by the slider) indicates how many times the scene is rendered per frame (with overdraw).
The second block selects which antialiasing technique to apply: MLAA, MSAA (4x) or no antialiasing. This is of course to allow comparison of the techniques’ performance and quality (the default choice is “MLAA”);

The last block of controls is only available if the antialiasing technique used is MLAA, and controls how the algorithm should be run:

- *Copy/Map/Unmap only* copies the color buffer from GPU memory to CPU memory and back, but won’t perform any MLAA processing between the two copy operations. This allows measurement of the impact of the copy operations on the overall performance of the entire algorithm;
- *CPU tasks pipelining* turns the pipelining system on/off for CPU onloading of graphics tasks (the default is on) so that it is easy to see the benefit of pipelining;
- *Show found edges* runs the first part of the algorithm (find discontinuities between pixels), but the blending passes are replaced by a debug pass, where a pixel is:
  - changed to solid green if a horizontal discontinuity has been found with its neighbor;
  - changed to solid blue if a vertical discontinuity has been found with its neighbor;
  - changed to solid red if both horizontal and vertical discontinuities have been found with its neighbors;
  - unchanged if no discontinuities have been found.

*Figure 4. Gear tower scene with MLAA and the “show found edges” option enabled*
Sample architecture
Without the pipelining optimizations, the sequence of events for each frame is:

<table>
<thead>
<tr>
<th>ANIMATE AND RENDER TEST SCENE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLAA STAGE (if MLAA is enabled)</td>
</tr>
<tr>
<td>• Copy the color buffer where the scene was rendered to a staging buffer</td>
</tr>
<tr>
<td>• Map the staging buffer for CPU-side access.</td>
</tr>
<tr>
<td>• MLAA post-processing (the staging buffer is both input and output)</td>
</tr>
<tr>
<td>• Unmap the staging buffer</td>
</tr>
<tr>
<td>• Copy staging buffer back to the GPU-side color buffer</td>
</tr>
<tr>
<td>• Render the zoombox (if applicable)</td>
</tr>
</tbody>
</table>

| RENDER THE SAMPLE’s UI, PRESENT FRAME |

Except for the “Perform the MLAA post-processing work” step (and again not considering the pipeline for now), all of these steps are implemented using standard Microsoft DirectX* methods.

The tasking API
By nature the MLAA algorithm is easy to parallelize. For both the discontinuities detection and blending passes, we can process the color buffer in independent chunks (blocks of contiguous rows or columns). MLAA can be implemented using a task-based solution that automatically makes full use of all available CPU cores, while keeping the code core count agnostic.

This sample uses a simple C-based tasking API that is implemented on top of the Intel® Threading Building Blocks (Intel® TBB) scheduler. This wrapper API was created to simplify the integration of the technique into existing codebases which already expose a similar tasking API (e.g. cross-platform game engines). An added benefit is the increased readability of the main source file MLAA.cpp.

The two important functions of the wrapper APIs are:

```c
BOOL CreateTaskSet(
    TASKSETFUNC pFunc,   // Function pointer to the
    VOID* pArg,          // Taskset callback function
    UINT uTaskCount,  // App data pointer (can be NULL)
    TASKSETHANDLE* pDepends, // Number of tasks to create
    // Array of TASKSETHANDLEs that
    // this taskset depends on. The
    // taskset will not be scheduled
    // until all tasksets in this list
    // complete.
    UINT uDepends, // Count of the depends list
    OPTIONAL LPCSTR szSetName, // [Optional] name of the taskset
    // the name is used for profiling
    OUT TASKSETHANDLE* pOutHandle // [Out] Handle to the new taskset
);
```
And:

```c
//  WaitForSet will yield the main thread to the tasking system and return
//  only when the taskset specified has completed execution.
VOID WaitForSet(
    TASKSETHANDLE               hSet     // Taskset to wait for completion
);  
```

The callback function has the following signature:

```c
typedef VOID (*TASKSETFUNC)(VOID* TaskInfo, INT iContext, UINT uTaskId, UINT uTaskCount);
```

MLAA requires a dependency graph of three consecutive tasksets:

- The first taskset finds the discontinuities between pixels in the color buffer;
- The second taskset performs the horizontal blending pass, and depends on the completion of the first taskset for the discontinuities information;
- The third taskset performs the vertical blending pass, which depends on the completion of the first taskset for the discontinuities information, but also on the completion of the second taskset because of the transpose optimization (cf. corresponding section for details).

This dependency graph is expressed in `MLAA.cpp` as a sequence of three calls to `CreateTaskSet`; the taskset callback functions (implemented in `MLAAPostProcess.cpp`) being respectively `MLAAFindDiscontinuitiesTask`, `MLAABlendHTask`, `MLAABlendVTask`.

If pipelining is not enabled, we wait for the last taskset to complete by calling `WaitForSet` with the handle of the last taskset. When the call returns, the MLAA work for the frame is complete. Things are just a little bit more complicated when using the pipeline.

**The CPU/GPU pipeline**

To get the maximum performance from our implementation, we have to keep both CPU and GPU sides fully utilized. Due to the data dependencies between the tasksets, full utilization can be achieved by interleaving the processing of multiple frames, as shown in the diagram below:

![Diagram of the CPU/GPU pipeline](image)

*Figure 5: Frames moving through the pipeline. The red blocks on the CPU main thread timeline illustrate that the main thread will take MLAA work if the application gets CPU-bound*
In other words, we have to build a pipelining system, where a pipeline is a sequence of CPU stages (workloads) and GPU stages, and be able to run multiple instances of said pipeline at the same time.

In our case, each instance of the pipeline corresponds to the processing of one frame. We have three steps:

Step 1 (GPU stage):

- Animate and render test scene;
- Copy color buffer to staging buffer (using asynchronous GPU-side CopyResource).

Step 2 (CPU stage):

- Map the staging buffer;
- Perform the MLAA post-processing work (staging buffer is both input and output).

Step 3 (GPU stage):

- Unmap staging buffer and copy it back to the GPU-side color buffer;
- Finish frame rendering (zoombox, UI) and present.

To implement this concept, we designed a simple Pipeline class. Each stage is represented by a PipelineFunction structure that specifies the function to be called, and the stage type. The callback function must have the following signature:

```c
typedef void (*PPPIPELINEFUNC)( UINT uInstance, ID3D11DeviceContext* pContext )
```

Where uInstance indicates which instance of the pipeline the function is being called from.

A GPU stage uses a DirectX* query (of type D3D11_QUERY_EVENT) to signal its completion, but a CPU stage has to explicitly call the Pipeline's class CompleteCPUWait method to signal completion:

```c
void CompleteCPUWait( UINT uInstance );
```

Creating the pipeline instances requires a single call to the Init method. In our case, the code is:

```c
PipelineFunction Func[ 2 ];
Func[0].Type = PIPELINE_EVENT_COMPLETE;
Func[0].pFunc = AnimateAndRenderScene;
Func[1].Type = PIPELINE_CPU_COMPLETE;
Func[1].pFunc = MLAAPostProcessing;

g_Pipeline.Init(g_NumPipelineInstances, ARRAYSIZE( Func ), Func, g_pPipelineQueries );
```

Where g_NumPipelineInstances is the number of pipeline instances to create (3 in this case).
To run the pipelines, we call the Start method each frame:

```c
UINT Start(    ID3D11DeviceContext* pContext,    // Context to issue Event query on.    BOOL bFlush = FALSE    // If bFlush is TRUE, the Pipeline    // will not create new instances.    // Specify TRUE if you want to flush    // the Pipeline.
);
```

Because the Start method returns the index of a completed instance of the pipeline, the third and last step doesn’t have to be called through the Pipeline class. The code of the last step executes right after the call to the Start method, indexing the data structures with the index returned by Start.

Pipelining does not rely on the tasking API WaitForSet call, since it is blocking and so does not allow pipelining to occur. The solution is to use a completion taskset—that is, a task that depends on the completion of all the MLAA tasks, and whose only work will be to call the CompleteCPUWait method.

**Intel® SSE optimizations**

The first pass of the MLAA algorithm identifies discontinuities between pixels. Conceptually, each pixel checks its color and compares it with its bottom neighbor (when looking for horizontal discontinuities) or right neighbor (when looking for vertical discontinuities) [Reshetov09, section 2.2.1].

In this sample, we have kept the simple discontinuity detection kernel of the reference implementation. A discontinuity exists between two pixels if at least one of their RGB color components differs by at least 16 (on the 0-255 scale of the RGBA8 format).

This definition works well in the sample and allows a very compact and efficient SIMD implementation. However, more complex approaches are possible, and sometimes necessary, to get the best possible results. For example, a pixel’s luminance could be used instead of directly comparing color values, and a variable threshold recomputed each frame to take into account the scene’s overall luminance and contrast [Luminance]. Depth values could be used to assist with edge detection or any kind of custom data to exclude specific zones of the color buffer from being processed.

Because this step is independent from the rest of the algorithm, and we are running on the CPU, any data from the program can be used directly as input data. The only limits to the detection algorithm are:
- The fact that the algorithm must output a bit for each pixel indicating whether a discontinuity is detected;
- The tradeoff between performance and quality/complexity;
- The programmer's imagination.

As with the original reference implementation [Reshetov09, section 2.4], we work with a RGBA8 render target format and the “vertical discontinuity” and “horizontal discontinuity” bit flags computed by MLAA are stored in-place in the two high bits of the pixel data (i.e., the two high bits of the alpha component which is unaffected by the MLAA blending operations). This keeps the implementation simple while helping with memory footprint, and allowing optimizations in the next steps of the algorithms (cf. “the transposing optimization” section below).

Because each pixel is 32 bits of data, and each pixel can be processed independently of the others in this step, we can process 4 pixels at a time using Intel® SSE intrinsics. The detection code is very short.

```c
#include <immintrin.h>

// Given a row of pixels, check for color discontinuities between a pixel and its neighbor. Intel SSE implementation, so we process all 4 pixels in the row at a time.
// If a discontinuity is detected, the corresponding flag is set in the alpha component of the pixel.

inline void ComparePixelsSSE(__m128i vPixels0, __m128i vPixels1, const INT EdgeFlag) {
    __m128i vDiff = _mm_sub_epi8(_mm_max_epu8(vPixels0, vPixels1), _mm_min_epu8(vPixels0, vPixels1));
    // We are only interested if the diff. is >16, and not interested in alpha differences.
    const INT Threshold = 0x00F0F0F0;
    __m128i vThresholdMask = _mm_set1_epi32(Threshold);
    vDiff = _mm_and_si128(vDiff, vThresholdMask);
    // Each of the four lanes of the selector is 0 if no discontinuity detected RGB, 0xFFFFFFFF else.
    __m128i vSelector = _mm_cmpeq_epi32(vDiff, _mm_setzero_si128());
}
```

The reference implementation then proceeds to update the alpha component of each pixel, but used inefficient serial code to do so. We can optimize this sequence by using the following Intel® SSE code:

```c
__m128i vEdgeFlag = _mm_set1_epi32(EdgeFlag);
vEdgeFlag = _mm_andnot_si128(vSelector, vEdgeFlag);
// vEdgeFlag now contains EdgeFlag for the pixels where a discontinuity was detected, 0 otherwise.
vPixels0 = _mm_or_si128(vPixels0, vEdgeFlag);
```

We also replaced the MixColors function (which computes a linear interpolation of two colors) to use a full Intel® SSE implementation.
The Transposing Optimization

The next task of the algorithm is to find “discontinuity lines”, i.e., sequences of consecutive pixels which are marked with the same discontinuity flag (horizontal flag for horizontal blending pass, vertical flag for vertical blending pass) while walking rows and columns of our color buffer [Reshetov09, section 2.1].

Because discontinuity lines tend to be short, intuition suggests (and profiling data confirms) that the most expensive part of this operation is scanning between discontinuity lines, i.e., scanning the large areas of consecutive pixels where no discontinuity flag is set.

The good news is that we can check 4 pixels at a time using the _mm_movemask_ps Intel® SSE intrinsic when the following conditions are met:

1. We are scanning 4 pixels stored at consecutive addresses;
2. The address of the first pixel is “Intel® SSE-aligned” (16 bytes alignment);
3. The discontinuity flag is stored as the high bit of the 32 bits of pixel data.

During the horizontal blending pass, (1) is true (we scan horizontal rows of pixels in the color buffer represented as a 2D linear array of pixel data); (2) is true almost all the time (remember that 16 bytes alignment is equivalent to “the index of the starting pixel in the buffer is a multiple of 4” as the buffer is properly aligned); and (3) is true as we chose bit 31 to represent the horizontal discontinuity flag.

If all conditions are true, we compute the flags:

```c
__m128 PixelFlags = *(reinterpret_cast<__m128*>(WorkBuffer + OffsetCurrentPixel));
// Creates a 4-bit mask from the most significant bits
int HFlags = _mm_movemask_ps(PixelFlags);
```

And five outcomes are possible depending on the value of HFlags:

- 0 (the most common case by far): no discontinuity flag set in this group of 4 pixels, move to the next group of 4 pixels.
- Bit 0 is set: discontinuity line starts at first pixel of the group.
- Bit 1 is set: discontinuity line starts at second pixel of the group.
- Bit 2 is set: discontinuity line starts at third pixel of the group.
- Bit 3 is set: discontinuity line starts at fourth pixel of the group.

This optimization is part of the reference implementation and works well for the horizontal blending pass, but was impossible to apply for the vertical blending pass. As the code scans the buffer vertically, (1) is false (adjacent pixels in a column are not stored at consecutive addresses), and (3) is false as well (the vertical discontinuity flag is stored at bit 30 of the pixel data).

The problem with (3) is easy to work around by introducing a simple “shift left by one bit” operation if we are processing the vertical flags, transforming the code above to:
int Shift = (EdgeFlag == EdgeFlagV) ? 1 : 0;
__m128 PixelFlags = *(reinterpret_cast<__m128*>(WorkBuffer + OffsetCurrentPixel));
PixelFlags = _mm_castsi128_ps(_mm_slli_epi32(_mm_castps_si128(PixelFlags), Shift));
int HFlags = _mm_movemask_ps(PixelFlags);

But (1) is still a problem. In addition, vertical scanning is extremely cache-unfriendly. Because of both of these issues, the vertical blending pass is about 3 times (300%) as expensive as the horizontal blending pass in the reference implementation (as shown by profiling data).

Our solution to this issue is to make the vertical blending pass use the cache and Intel® SSE-friendly data access patterns of the horizontal blending pass by considering the color buffer as a matrix of pixels and transposing it between passes:

- Perform horizontal blending pass
- Transpose the (horizontally blended) color buffer
- Perform vertical blending pass
- Transpose back color buffer

This way the code for both blending passes is exactly the same (which adds the advantage of simplicity/readability), the only difference being which flag to scan for, and we benefit from all the optimizations and cache-friendliness of the horizontal pass.

In practice, the transpose operations are not implemented as separate passes/tasksets, but as the last part of their respective blending pass. This allows us to benefit from the cache “warmness”.

The two transpose operations are not free. The cost is the extra code execution time, one extra work buffer and a synchronization point between the horizontal and vertical passes (we must wait for all horizontal tasks to be done before we can start any of the vertical tasks, because we must wait for the color buffer to be fully transposed before starting the vertical pass work).

Even with these extra costs, the overall performance is significantly better than the previous approach. As expected, profiling data shows that both blending passes have equivalent performance.

**Performance Results**

MLAA performance was measured on the following two configurations:

- Code name “Huron River” : Intel® Core™ i7-2820QM Processor (Intel® microarchitecture code name Sandy Bridge, 4 cores 8 threads @2.30 GHz) with GT2 processor graphics, 4 GB of RAM, Windows 7 Ultimate 64-bit Service Pack 1
- Code name “Kings Creek”: Intel® Core™ i5-660 Processor (codename “Clarkdale”, 2 cores 4 threads @ 3.33 Ghz), with GMA HD Graphics (codename “Ironlake”), 2 GB of RAM, Windows 7 Ultimate 64-bit Service Pack 1
We measured the average frame rendering time of our sample as a function of scene complexity for the different antialiasing settings. We also measured the rendering time for the Copy/Map/Umap only mode to highlight the impact of the color buffer copy operations on the overall performance of the algorithm.

**Results**

The data shows that for the Huron River machine, the extra cost of MSAA 4x goes up linearly with the scene complexity (in fact, for all resolutions, the frame time when using MSAA 4x to render our test scene appears to be approximately the double of the frame time when no antialiasing is used). In contrast, the cost of MLAA appears more or less constant (around 4 ms/frame at 1280x800). This is consistent with our expectations, as unlike MSAA 4x, MLAA is executed only once per frame, regardless of scene complexity / number of draw calls.

We also observe that except for very low complexity values (i.e. less than ~5) MLAA always outperforms MSAA 4x, regardless of resolution, and that the difference in performance grows with complexity (because as noted above, the cost of MSAA 4x grows linearly with complexity when the cost of MLAA does not).

In the case of the Kings Creek machine, we can’t compare the cost of MLAA vs. MSAA 4x as the latter is not provided by the Ironlake hardware. The goal is then to determine if MLAA allows us to provide software antialiasing as an alternative with acceptable performance. Our measurements at the 1280x800 resolution show that again the cost of MLAA is largely independent of scene complexity, and the average value is ~7.5 ms (discarding the outlier data point at complexity = 100).

Interestingly, if we compare this result to a hypothetical MSAA 4x implementation with approximately the same performance profile than the Huron River one (frame time with MSAA 4x ~ 2x frame time with no antialiasing), we notice that again MLAA would outperform MSAA 4x for almost all complexity values (≥ 4 in this case).

<table>
<thead>
<tr>
<th>Kings Creek II/GFX</th>
<th>Scene Complexity between 1-100</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resolution 1280x800</strong></td>
<td></td>
</tr>
<tr>
<td>No AA</td>
<td>5.54 6.72 8.73 13.34 16.63 25.16 35.64 46.12 56.61 66.63 77.74 88.06 95.81 100.71</td>
</tr>
<tr>
<td>MLAA with pipeline on</td>
<td>13.00 14.26 16.63 20.16 22.48 33.27 43.90 54.13 64.24 74.82 85.23 95.96 106.02 116.59</td>
</tr>
<tr>
<td>MLAA with pipeline off</td>
<td>17.03 18.33 20.83 24.94 26.59 37.24 49.91 58.73 67.82 78.17 88.76 99.79 109.22 119.63</td>
</tr>
<tr>
<td>MLAA in &quot;copymap/unmap only&quot; mode</td>
<td>11.16 11.33 13.49 17.77 20.00 29.95 41.65 50.81 60.98 71.13 83.17 92.35 102.77 112.93</td>
</tr>
<tr>
<td>MLAA in &quot;copymap/unmap only&quot; mode, pipeline on</td>
<td>9.98 11.08 13.43 17.20 19.42 29.96 39.91 50.36 61.01 72.15 83.17 92.17 102.52 113.89</td>
</tr>
<tr>
<td>MLAA with pipeline on - No AA</td>
<td>7.46 7.54 7.90 8.82 9.85 11.11 11.26 13.81 16.63 18.39 19.49 21.54 23.02 24.58</td>
</tr>
</tbody>
</table>

Table 1. Rendering times of our test scene on the King's Creek machine.
Table 2. Rendering times of our test scene on the Huron River machine.
Figure 6: Rendering times of our test scene for each of the antialiasing techniques, as a function of scene complexity. The bottom, flat curve is the difference between the “MLAA with pipeline on” curve, and the “No antialiasing” curve, measuring the cost of using MLAA [Huron River, res. 1280x800]
Figure 7: Rendering times of our test scene for each of the antialiasing techniques, as a function of scene complexity. The bottom, flat curve is the difference between the “MLAA with pipeline on” curve, and the “No anti aliasing” curve, measuring the cost of using MLAA [Huron River, res. 1600x1200].
Figure 8: Rendering times of our test scene with MLAA on and off, as a function of scene complexity. The bottom, flat curve is the difference between the “MLAA with pipeline on” curve, and the “No antialiasing” curve, measuring the cost of using MLAA [Kings Creek, res. 1280x800]
References


[Luminance] Definition of luminance for CRT-like devices:


And for LCDs:

ITU-R Rec. BT.709-5. 2008. Page 18, items 1.3 and 1.4