SIGGRAPH2011 VANCOUVER

Filtering Approaches for Real-Time Anti-Aliasing

http://www.iryoku.com/aacourse/

SIGGRAPH2011 VANCOUVER

Filtering Approaches for Real-Time Anti-Aliasing

Hybrid CPU/GPU MLAA
(on the Xbox 360)

Pete Demoreuille
pbd@pod6.org

SIGGRAPH2011 VANCOUVER

MLAA

- Algorithm well described by this point
- As well as benefits and drawbacks
- This talk focuses on
  - Shipping hybrid CPU/GPU implementation
  - Assorted edge detection routines
  - Integration and use in engine

Prior presenters have well explained the MLAA algorithm and some implementation approaches, as well as some of the motivations for its use (alternative to MSAA, lower memory, application to deferred shading) and some of the drawbacks (fonts, unnecessary blurring, cost etc).

In this talk we'll describe an implementation that has been used in several titles for the Xbox 360, one that has very low (and fixed) GPU costs and makes MLAA an option on the platform. We'll also touch on some of the techniques developed to adapt the edge filtering to each title's visual style, and general implementation and deployment notes relevant for any real-time application.

SIGGRAPH2011 VANCOUVER

Example Results

Prior presenters have well explained the MLAA algorithm and some implementation approaches, as well as some of the motivations for its use (alternative to MSAA, lower memory, application to deferred shading) and some of the drawbacks (fonts, unnecessary blurring, cost etc).

In this talk we'll describe an implementation that has been used in several titles for the Xbox 360, one that has very low (and fixed) GPU costs and makes MLAA an option on the platform. We'll also touch on some of the techniques developed to adapt the edge filtering to each title's visual style, and general implementation and deployment notes relevant for any real-time application.

SIGGRAPH2011 VANCOUVER

Example Results

Prior presenters have well explained the MLAA algorithm and some implementation approaches, as well as some of the motivations for its use (alternative to MSAA, lower memory, application to deferred shading) and some of the drawbacks (fonts, unnecessary blurring, cost etc).

In this talk we'll describe an implementation that has been used in several titles for the Xbox 360, one that has very low (and fixed) GPU costs and makes MLAA an option on the platform. We'll also touch on some of the techniques developed to adapt the edge filtering to each title's visual style, and general implementation and deployment notes relevant for any real-time application.

SIGGRAPH2011 VANCOUVER

Example Results

Prior presenters have well explained the MLAA algorithm and some implementation approaches, as well as some of the motivations for its use (alternative to MSAA, lower memory, application to deferred shading) and some of the drawbacks (fonts, unnecessary blurring, cost etc).

In this talk we'll describe an implementation that has been used in several titles for the Xbox 360, one that has very low (and fixed) GPU costs and makes MLAA an option on the platform. We'll also touch on some of the techniques developed to adapt the edge filtering to each title's visual style, and general implementation and deployment notes relevant for any real-time application.

SIGGRAPH2011 VANCOUVER

Example Results

Prior presenters have well explained the MLAA algorithm and some implementation approaches, as well as some of the motivations for its use (alternative to MSAA, lower memory, application to deferred shading) and some of the drawbacks (fonts, unnecessary blurring, cost etc).

In this talk we'll describe an implementation that has been used in several titles for the Xbox 360, one that has very low (and fixed) GPU costs and makes MLAA an option on the platform. We'll also touch on some of the techniques developed to adapt the edge filtering to each title's visual style, and general implementation and deployment notes relevant for any real-time application.

SIGGRAPH2011 VANCOUVER

Motivations

- Engine uses variant of deferred rendering
- GPU time at a premium, prefer fixed cost
- SSAA (!) originally used, needed alternative
- Complicated by time pressure
  - Added *right* before shipping Costume Quest
  - Aspects of implementation show this

Briefly, we were forced to abandon Costume Quest's original antialiasing approach of supersampling (at 1520x848), which was originally chosen in the interest of implementation speed. As the game sped towards release, the heavy performance toll placed on the GPU by SSAA, compounded by the increasing quantity of art, heavy use of lights and transparency made the approach no longer tenable (not just the 50% increase in pixels, but additional passes required due to the lack of tiling support). It also didn't improve quality as much as desired (especially compared to the cost), particularly on the PS3. But there was a very strong desire to ship with some sort of antialiasing, and not enough time to get tiling functioning on the XBOX.

A month or two before, I had read about MLAA on the realtimerendering.com blog, and thought about a few ideas that might make the technique feasible for realtime use. Quickly those loose ideas became a risky, last ditch plan to improve rendering performance and hopefully retain good image quality. We hoped to get a pre-release version of Sony's SPU MLAA, to make sure both SKUs had high quality visuals.

Many of the details of the implementation bear the mark of this time pressure – efficient time use was of the essence, since by this time the studio's total graphics programmer bandwidth was 50% of one person, responsible for 4 games in production!

SIGGRAPH2011 VANCOUVER

Starting Points

- Reference implementation from paper
  - Took unspeakable amount of time on PPC
- First-pass fixed-point implementation
  - Took ~90ms (not include untiling!)
  - ~21ms for edges, ~4.5ms blending
  - ~65ms for blend weight calculations

The original implementation using the reference code took a very, very long time (though on my U7300 1.3ghz laptop, it only took ~40-50ms).

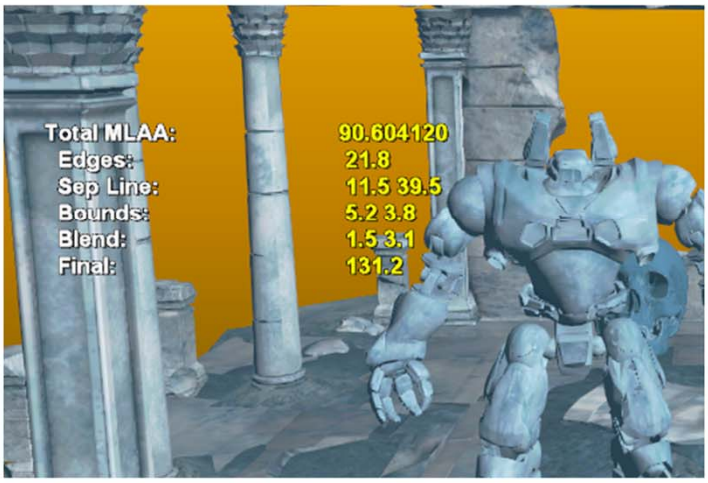
A first-pass at converting the code to a more PPC/Console friendly form was done, and though it was still quite slow, it showed some promise.

At this point the idea to split the implementation and do the edge and blending calculations on the GPU was first seriously considered, while the remainder would run on the CPU.

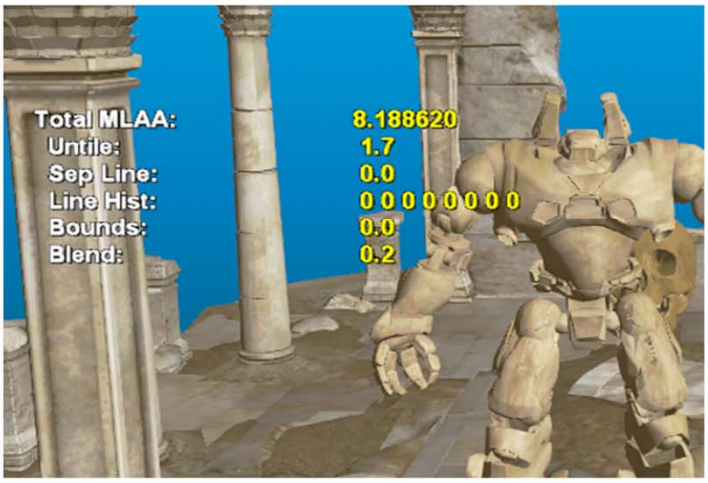
I freely admit the idea may have been swayed by the often enjoyable challenges encountered when making the CPU and GPU communicate in such a way!

SIGGRAPH2011 VANCOUVER

Starting Points



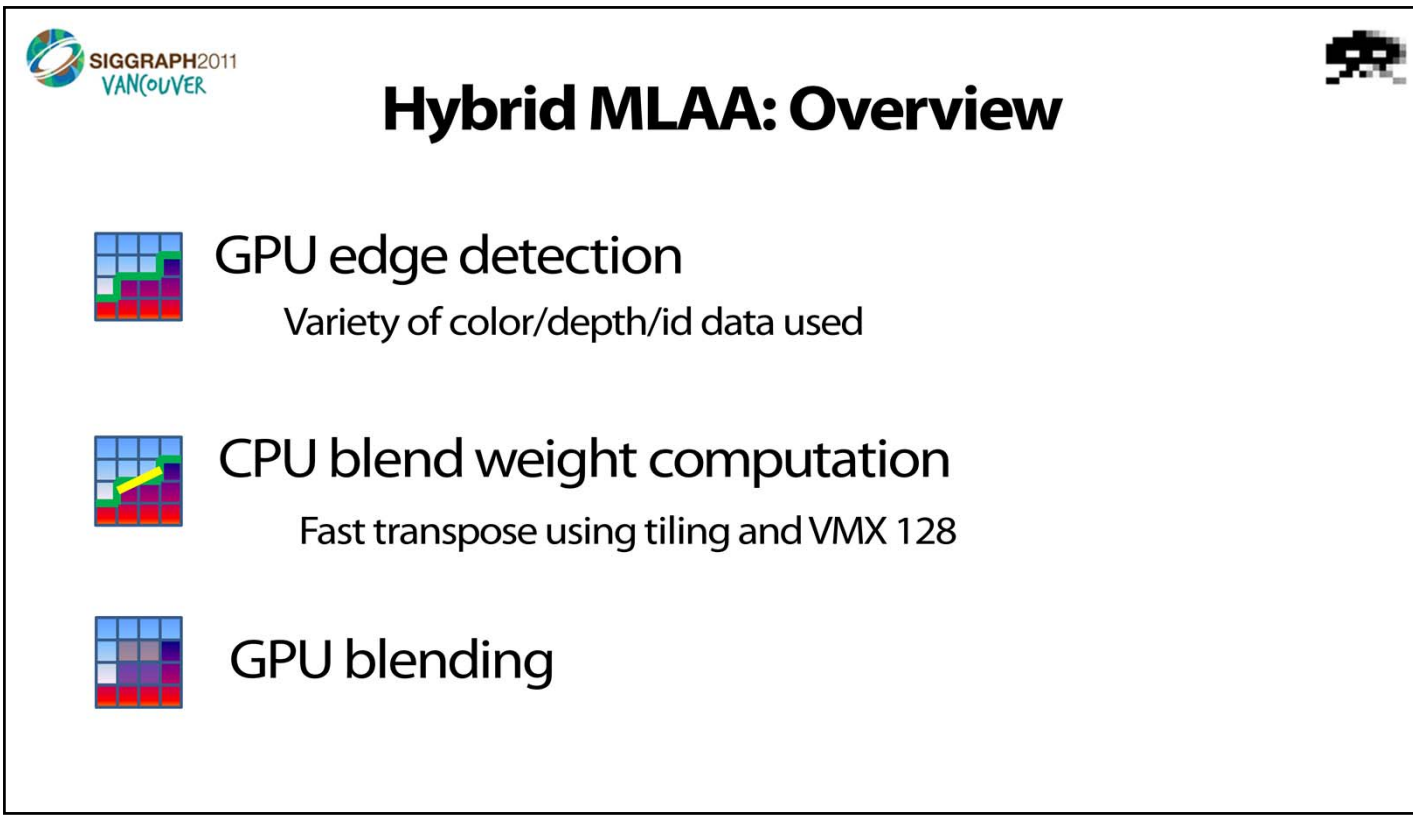
Fully on cpu, fixed point
~4fps (90ms aa)



Gpu edges + rest on Cpu, optimized masks
~68fps (8ms aa+untile)

Sample Art Courtesy of Microsoft

Some key points in the prototype: a first pass at making the code run more than 1fps (not counting tiling), and after the edge detection had moved to the GPU and blend weight computation had improved a great deal. After this the implementation moved in-engine and further optimizations and threading was done.

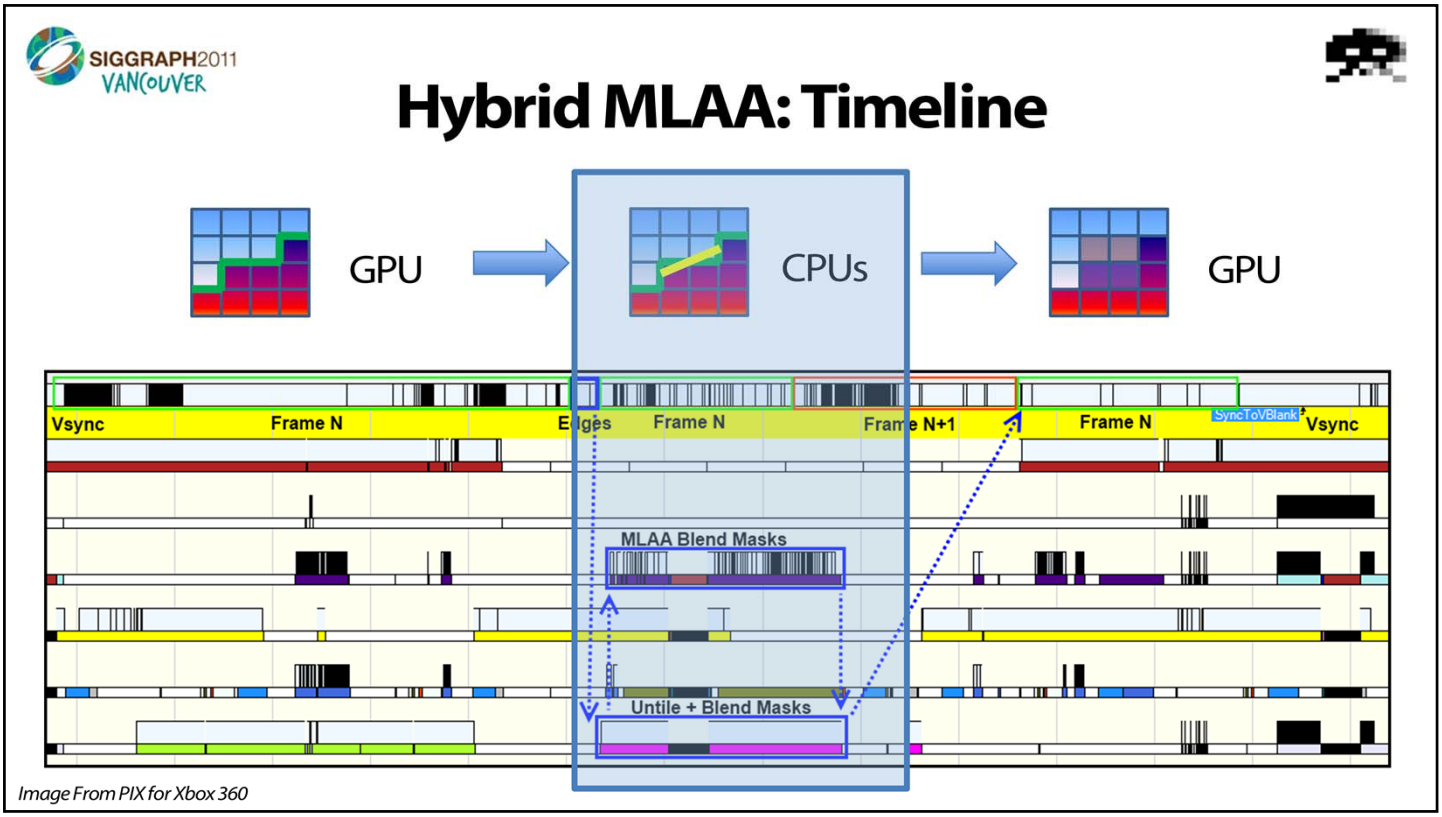


Our implementation consists of three major steps.

Edge detection, using a variety of depth, normal, and id buffers in conjunction with raw color data is performed on the GPU. When complete, the GPU notifies a CPU via an interrupt that the data is ready.

This edge data is then read on the CPU to form two blend masks (horizontal and vertical). The major issue in doing this quickly is reducing the amount of bandwidth used by the algorithm, in part by avoiding a vertical edge scan by performing a fast transpose of the input edge data. Once complete, the CPUs will notify the GPU that the blend data is available using an asynchronous resource lock (or async command buffer).

These final buffers (blend masks) containing blending weights are then used by the GPU to filter the near-final image.



This example timing capture from Stacking (on a reasonably light frame, to help make this legible) illustrates the various stages of the algorithm and the relative time used.

Each of the horizontal bars below represent the various hardware threads on the Xbox 360, and the two blue boxes in the center represent MLAA processing.

The dotted blue arrows show the flow of data.

Along the top, surrounded in green and red boxes, is the GPU workload. The timing starts and stops at vertical blanks, as this image shows a single frame.

Note there is some amount of time left before the GPU would stall on CPU results, even though this frame is already somewhat heavy on blend mask time, making it very unlikely the GPU would stall on edge data.

SIGGRAPH2011 VANCOUVER

Blend Mask

- CPU *blend mask* generation
  - Contains weights for GPU use when filtering
- GPU to provide input data
  - Flags for horizontal / vertical edges
  - Intensity values for blending calculations
- Hypothesis: calculation bearable, bandwidth not
  - Fist reduce size as much as possible

The two images which hold horizontal and vertical blending weights for neighbor pixels we will call the blend masks.

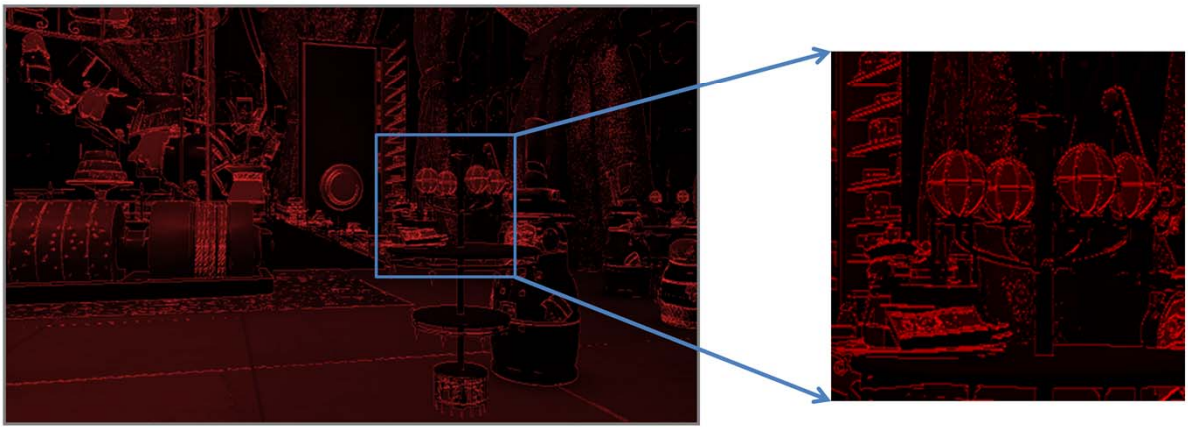
These are computed from edge and luminance data provided by the GPU.

From the original test implementation, we knew that bandwidth would be as large an issue, if not more, than the amount of calculations going on, so reducing it was a primary goal.

SIGGRAPH2011 VANCOUVER

Blend Mask Input

- Use 8bpp: 6-bit luminance, 2-bit H+V edge
  - Our implementation actually uses 16bpp, 1 channel for other stuff



The first step in doing this was passing less data from the GPU to the CPU. We use 8bpp, which is split into a 2 bits of horizontal and edge flags, and 6 bits of luminance data.

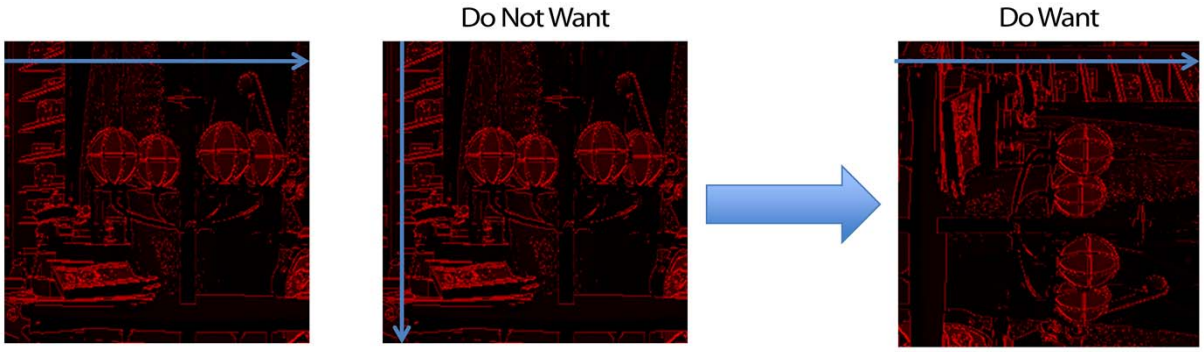
In our actual implementation, we actually use a 16bpp target which contains the edge data along with other unrelated data into the second channel. This was mainly to save a little GPU time.

It should be noted that this data is passed from the GPU in a tiled format, making it unsuitable for use on the CPU. After receiving the data, a (very slow) function from the XDK was originally used to untile the image, which it was assumed could be made faster later.

SIGGRAPH2011 VANCOUVER

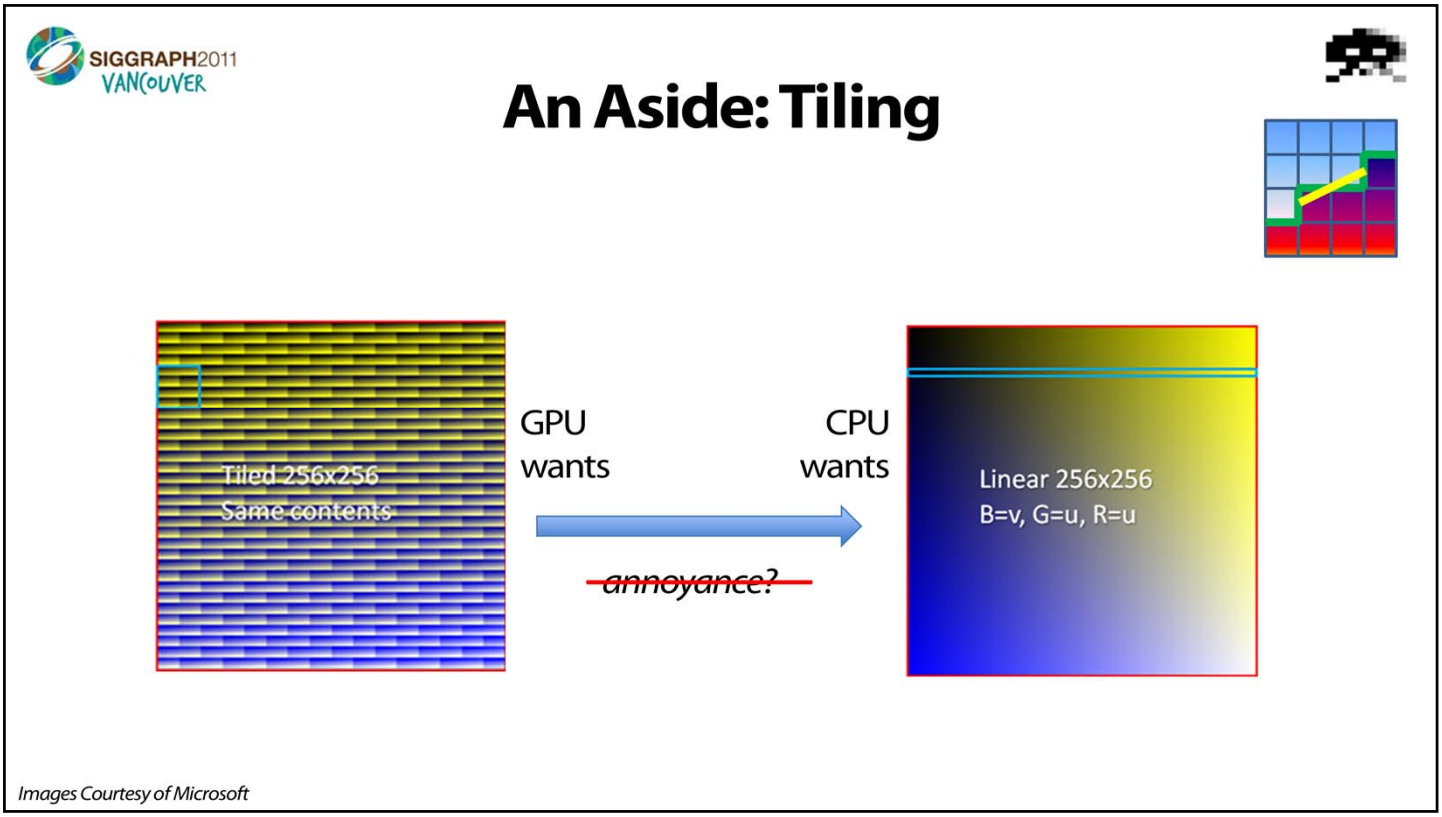
Blend Mask Input

- Large reduction, bandwidth still an issue
  - Vertical edges *obliterate* cache
  - Transpose image!



This greatly reduces the cost for horizontal blend mask generation (and enables some additional optimizations), the vertical blend mask generation is still horribly slow due to the cache-unfriendly access pattern.

An obvious solution would be to transpose the image, but exactly how to accomplish that in an efficient manner was slightly less obvious.



What was originally just an annoyance ended up providing an excellent solution to the problem.

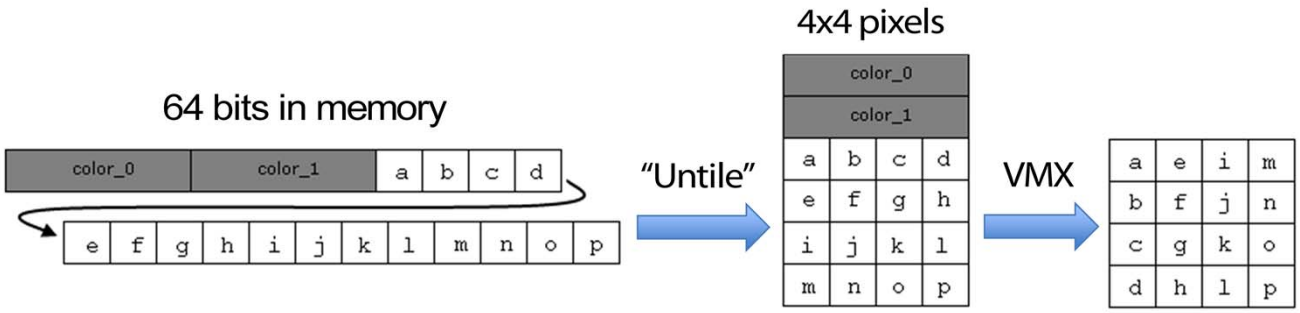
We could utilize the fact that a tiled image has blocks of the image arranged linearly in memory, so we could read in small square regions, transpose them, and write out the blocks in a transposed pattern.

Doing this over the whole image allows us to transpose it in a very cache and CPU efficient manner.

SIGGRAPH2011 VANCOUVER

Not an Annoyance

- DXT blocks
  - One read for 4x4 pixels



Instead of going into the intricacies of Xbox texture tiling, we'll use the familiar example of DXT blocks.

The key insight is that encoding patterns like this allow a single aligned, linear, easily prefetchable read to load a rectangular block of pixels. One can then take care to load a square block using VMX128/Altivec/ your vector instruction set of choice, transpose that, and write out both an untiled and untiled+transposed image simultaneously.

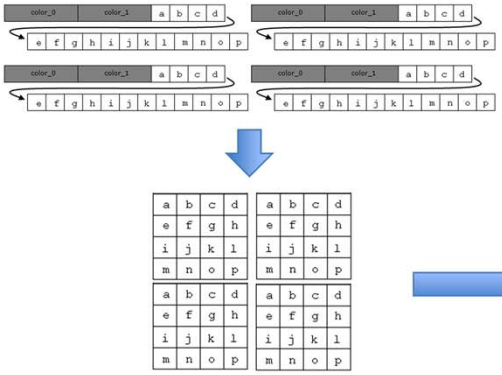
Prefetching carefully and trying to make the best tradeoff of block size, instructions count to transpose each block, read bandwidth per instruction, etc is simply labor from that point on. The untile in our implementation also discards 8 bits of the 16bpp texture produced by the GPU, and operates on 32x32 blocks which are processed as 4 8x8 sub-blocks, two of which fit into a set of vector registers at once.

The new FastUntile sample application in the XDK is an excellent starting point for performing the untile quickly.

SIGGRAPH2011 VANCOUVER

Multiple tiles

Read from few cache lines



Into blocks of vector registers

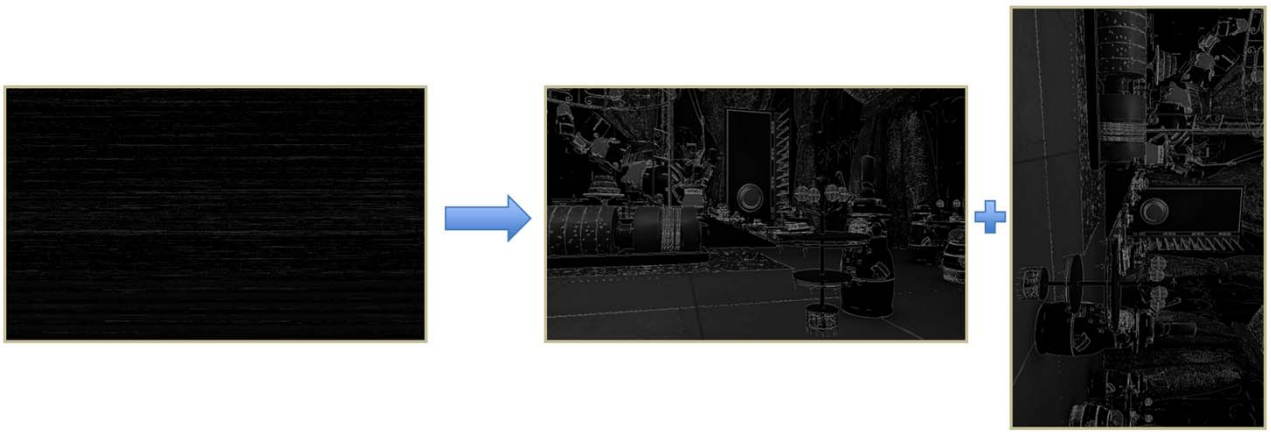
Store original and transpose

A visual example of progressing blocks of pixels and how they might be written out.

SIGGRAPH2011 VANCOUVER

Blend Mask: Transpose

- Untile creates horizontal, vertical images
  - We convert from 16bpp -> 8bpp here as well



In summary, we take something which looks nothing like an image and convert it into two 8bpp images containing horizontal and vertical edges.

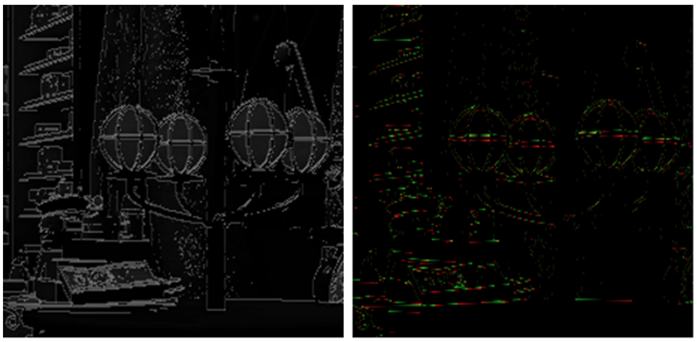
We then run the MLAA algorithm on these two images.

SIGGRAPH2011 VANCOUVER

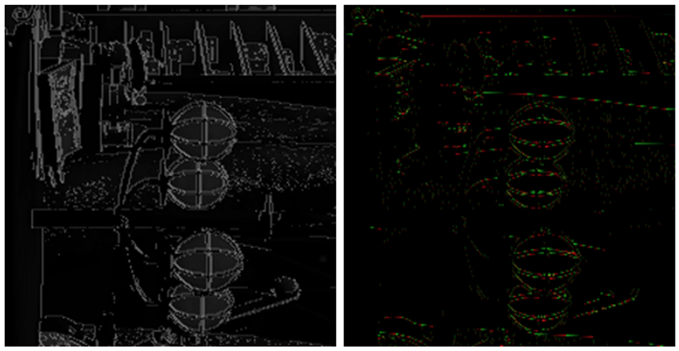
Blend Mask

- Now run horizontal mask code twice
  - Massive bandwidth reduction

Horizontal edges



Vertical edges



Our implementation uses an optimized version of the reference code – not many functional modifications were originally made besides a few changes to blend weight calculations.

The fact that we only perform horizontal blend mask generation makes additional optimizations possible, and makes the code much easier to maintain.

SIGGRAPH2011 VANCOUVER

Blend Mask: Threading

- Last major step: threads
  - Process interleaved horizontal blocks
  - Jobs kicked as untiling of blocks completes
    - L2 still warm when mask generation starts
  - Wait for complete untile before vertical mask
- Final cost ~4-8ms per thread
  - Tons of potential optimizations left

Threading was the one of the last steps, which does further complicate the careful balancing of cache prefetching and bandwidth usage. The speedup was certainly not 1:1 with each thread added, and titles so far end up using two threads total.

The threading pattern arrived at is similar to the reference code's, interleaved horizontal blocks of the image are processed in a way that two adjacent blocks are not used simultaneously. A single thread is responsible for locking the GPU resources, untiling the image and submitting work, and will then processes blocks once untiling is complete.

It should be noted that the core MLAA code has many remaining opportunities for optimizations, both with respect to the speed of the code and additional preprocessing that could be cheaply done on the edge data to allow more efficient skipping of empty space, line length searches, etc. Many of the ideas presented in the GPU implementations could be adapted to optimize the CPU code (particularly those that limit the total size of processed edges).

SIGGRAPH2011 VANCOUVER

Blending

- GPU reads horizontal and vertical masks
  - Linear textures, but shader is ALU bound
  - Performed with color correction, etc

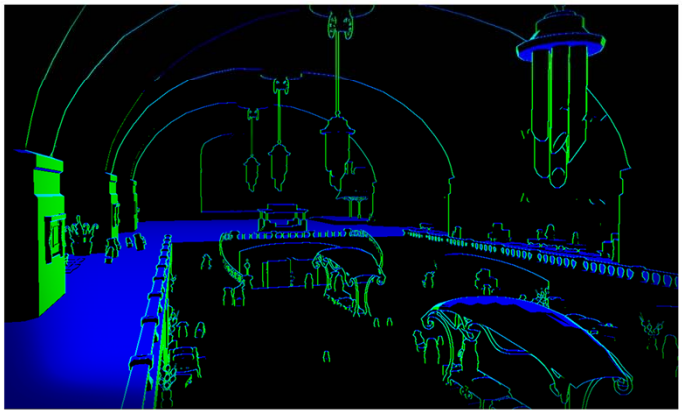
Blending is straightforward. In one of our final post-processing shaders (that is ALU heavy), we read the blend masks and blend between adjacent pixels in the image to produce the final, antialiased image.

We currently blend all four neighbors, but based on preliminary results from Jimenez's MLAA, using bilinear filtering to blend only the dominant horizontal and vertical neighbor may produce similar results while reducing total blending time.

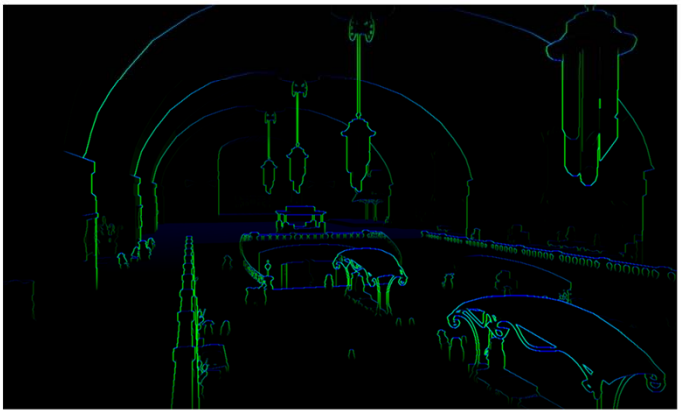
SIGGRAPH2011 VANCOUVER

Depth-based Edges

Absolute Tolerance



Relative Tolerance



Flat areas can