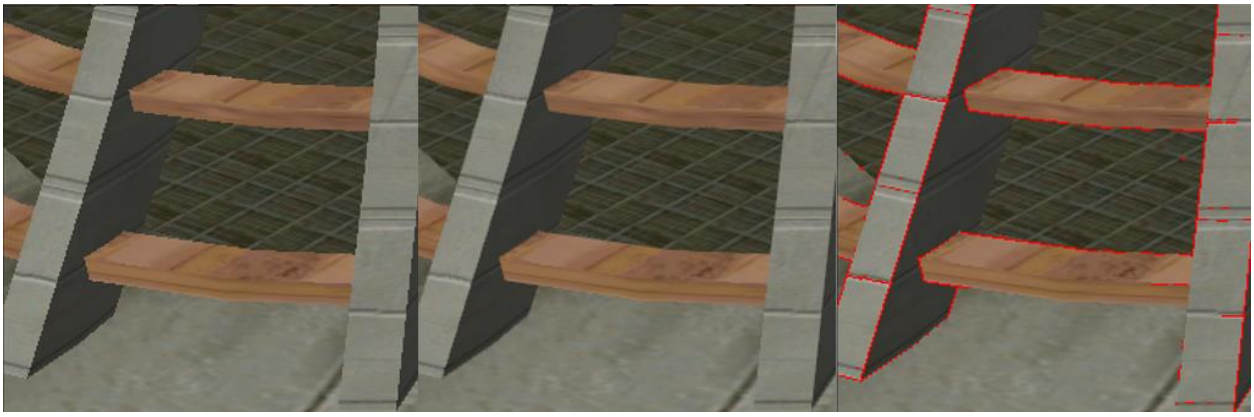
MLAA11

AMD Developer Relations



<left>Without AA,
<middle> With MLAA(edge threshold = 12, using texture filtering),
<right>The detected edges(edge threshold = 12)

Overview

This sample demonstrates a post-process pixel shader technique that applies Full-Screen Anti-Aliasing (FSAA) to an image. Morphological Antialiasing (MLAA) was originally developed by Intel Lab (see reference) but was designed for a CPU based post-process. This sample demonstrates a modified MLAA implementation adapted to run on the GPU.

Recently, deferred rendering has been playing a more and more important role in the game industry. One of the advantages of MLAA is that it works well with deferred or semi-deferred rendering techniques while traditional MSAA is more difficult to implement and much more costly from a performance perspective. Another advantage of MLAA is that it is very simple to integrate into existing rendering systems because it works as a post process operation. Finally post-process FSAA techniques like MLAA work on every pixel on the screen which means that edges resulting from the use of semi-transparent textures also benefit from anti-aliasing.

Implementation

This implementation of MLAA uses three passes to resolve the aliasing. This implementation also allocates two screen-sized render targets to store temporary data. In this implementation, we use DXGI_FORMAT_R8_UINT for edge mask storage and DXGI_FORMAT_R8G8_UINT for edge length storage. The format of the second render target actually depends on the maximum edge length the shader needs to process. The longer the edge, the bigger the render target format should be. In this sample, we use

DXGI_FORMAT_R8G8_UINT for edge length storage because it achieves a good balance between image quality and performance.

Luminance

In this sample, we compute the luminance of each pixel at the scene rendering stage and store it to the alpha channel of a render target. This luminance data is then used as input to the edge detection pass. We recommend developers adopt the same technique because using luminance for edge detection can reduce a lot of the texture fetches required.

Edge detection pass

This is the first pass of MLAA. In this pass, the shader computes the luminance difference between the current pixel and up/right neighbor pixels. Whenever the luminance difference is bigger than a specified threshold value, the shader flags this pixel as an edge pixel in the render target. We use kUpperMask and kRightMask to indicate the direction of the edge detected.

Additionally this sample offers the option to use the stencil buffer to mark which pixels have been detected as edges. For scenes of higher edge complexity this turns out to be a decent, and we will see why later.

Edge length computing pass

This is the second pass of MLAA and this pass uses the output render target from the previous pass. In this pass, the shader first checks the mask value of the current pixel. If the mask value doesn't contain kUpperMask and kRightMask , then the shader discards this pixel since it is not considered to be an edge pixel and therefore wouldn't cause aliasing. Otherwise, the shader needs to compute the edge length of this pixel. The shader computes the edge length in 4 directions, up, down, right and left. For each direction, the shader increases the edge length by one for each iteration until reaching the maximum edge length or a non-edge pixel. We use a stop bit here to stop the edge length computing when reaching a non-edge pixel. After computing the edge length for 4 directions, the shader packs the four edge length values into a uint2 variable then write it into the output render target.

When the application optionally uses the stencil buffer mask, this pixel shader is only instantiated for edge pixels, which can reduce the cost of this pass.

Color blending pass

This is the final pass of MLAA and this pass needs to use the output render target from the previous pass. This pass computes the final pixel color by using a lerp() function to blend the 4 neighbor pixel colors according to the shape and length of an edge. The shader first gets 4 edge lengths from the current, down and left pixel. The shader retrieves the vertical and horizontal edge length of the current pixel, the horizontal edge length of the down pixel and the vertical edge length of the left pixel. Then the shader compares the luminance of end pixels of each edge with the neighbor pixels to determine the shape of the edge. After determining the edge shape, the shader computes a weight value according to its shape then uses this weight value to blend the current pixel color to neighbor pixel color.

References

Morphological Antialiasing, Alexander Reshetov
http://visual-computing.intel-research.net/publications/papers/2009/mlaa/mlaa.pdf

Practical Morphological Anti-Aliasing, Jorge Jimenez, Belen Masia, Jose I. Echevarria, Fernando Navarro, Diego Gutierrez
http://www.iryoku.com/mlaa/

Advanced Micro Devices
One AMD Place
P.O. Box 3453
Sunnyvale, CA 94088-3453

http://www.amd.com
http://developer.amd.com