FXAA

Timothy Lottes
tfarrar@nvidia.com

February 2009

Document Change History

Version	Date	Responsible	Reason for Change
1.0	25/01/11	Timothy Lottes	Initial release

Introduction

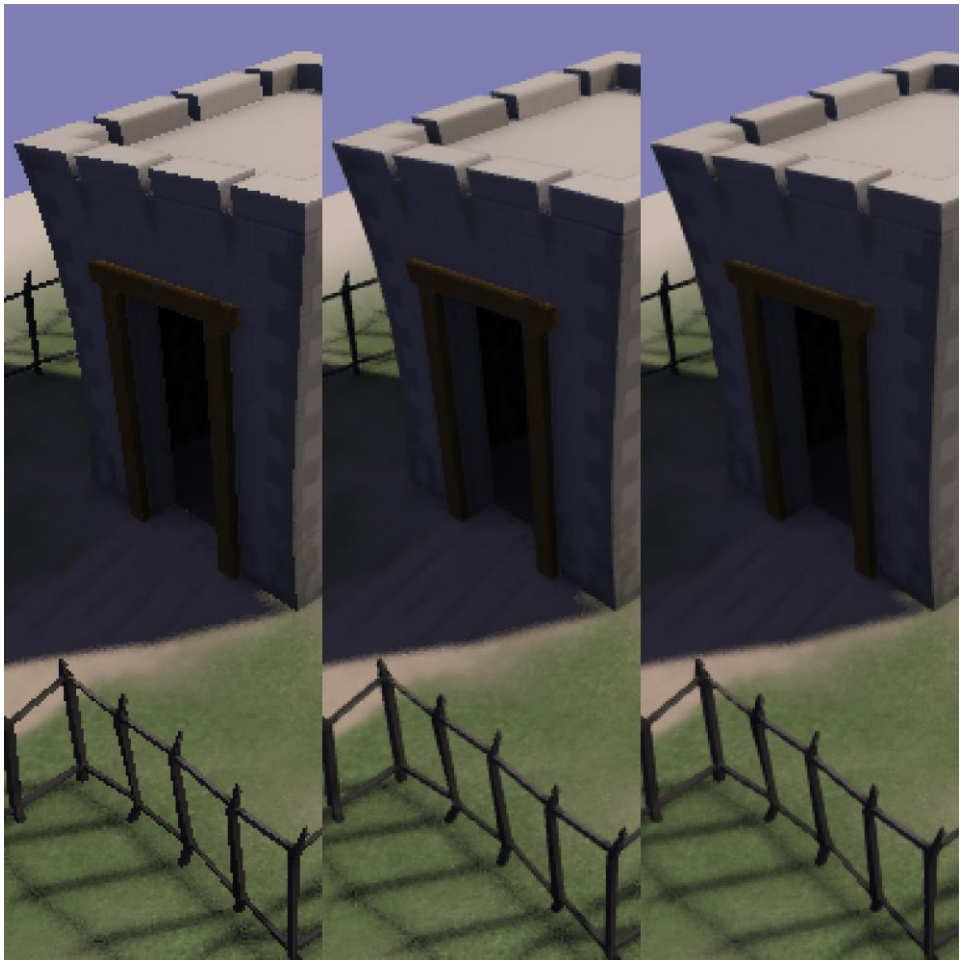


Figure 1: No AA (left) vs 4xMSAA vs FXAA Preset 3 (right)

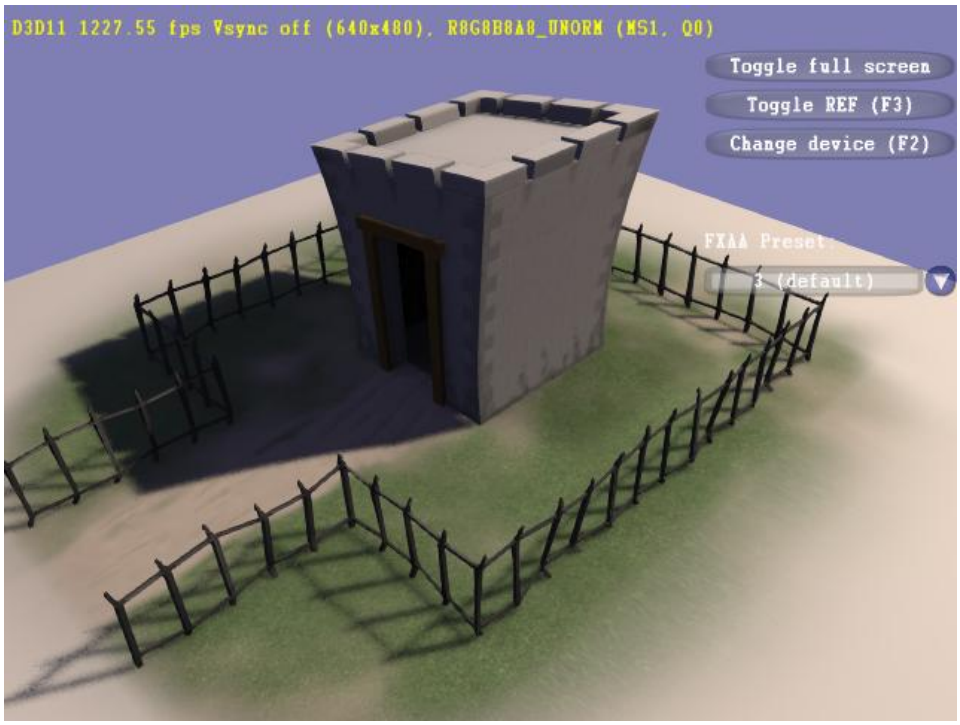
This sample presents a high performance and high quality screen-space software approximation to anti-aliasing called FXAA.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com

FXAA

1. Reduces visible aliasing while maintaining sharpness and keeping within a practical ms/frame cost for a typical game engine. Cost to process a 1920x1200 frame on GTX480 is under 1ms for FXAA preset 2.
2. Targets aliasing both on triangle edges and in shader results. FXAA has logic to reduce single-pixel and sub-pixel aliasing: see the reduction in stipple aliasing in the jittered sampling shadow area in Figure 1.
3. Is easy to integrate into a single pixel shader. FXAA runs as a single-pass filter on a single-sample color image. FXAA can provide a memory advantage over MSAA, especially on stereo and multi-display render targets or back buffers.
4. Can provide a performance advantage for deferred rendering over using MSAA and shading multiple samples.

Running the Sample



The left button controls the light position and the right button the camera position. The mouse wheel zooms in and out. Use the "FXAA Preset:" drop down to select the preset or to turn FXAA off.

The "Change device" interface can be used to select MSAA modes to mix FXAA and MSAA. However performance with mixed MSAA and FXAA is not representative of a proper integration: this sample outputs FXAA filtered results back into a MSAA render target.

Integration

Preset Selection

Fastest integration is to simply select from one of the example presets. See "Implementation" below for fine tuning.

0. Preset 0 has the highest performance and lowest quality. It simply breaks the aliasing on edges and applies a lower quality sub-pixel aliasing removal filter. This preset is not designed to be practical, but rather to show the lower limits of the tunable parameters.
1. Preset 1 extends the end-of-edge search radius, adds the high quality sub-pixel aliasing removal filter, and increases the effected local contrast range. This preset is designed for highly performance constrained situations like high resolution mixed with laptop GPUs.
2. Preset 2 is a good default for high performance but still shows some artifacts do to the end-of-edge search acceleration. This preset decreases the search acceleration, increases the end-of-edge search radius, and increases the effected local contrast range.
3. Preset 3 is the default, good performance and high quality without artifacts: end-of-edge search acceleration is turned off.
4. Preset 4 and higher continue to increase the end-of-edge search radius. The improved quality is marginal for the increase in cost. These presets are designed to show the upper limits of the tunable parameters.

Single Full Screen Pass

FXAA runs as a single full-screen triangle pixel shader post processing pass which can easily be integrated with the following shader shell,

```
#define FXAA_PRESET 3
#define FXAA_HLSL_4 1
#include "FxaaShader.h"

cbuffer cbFxaa : register(b1) {
    float4 rcpFrame : packoffset(c0); };

struct FxaaVS_Output {
    float4 Pos : SV_POSITION;
    float2 Tex : TEXCOORD0; };

FxaaVS_Output FxaaVS(uint id : SV_VertexID) {
    FxaaVS_Output Output;
```

```
    Output.Tex = float2((id << 1) & 2, id & 2);
    Output.Pos = float4(Output.Tex * float2(2.0f, -2.0f) +
        float2(-1.0f, 1.0f), 0.0f, 1.0f);
    return Output; }

SamplerState anisotropicSampler : register(s0);
Texture2D    inputTexture       : register(t0);

float4 FxaaPS(FxaaVS_Output Input) : SV_TARGET {
    FxaaTex tex = { anisotropicSampler, inputTexture };
    return float4(FxaaPixelShader(
        Input.Tex.xy, tex, rcpFrame.xy), 1.0f); }
```

Note, FXAA presets 0 through 2 require an anisotropic sampler with max anisotropy set to 4, and for all presets, there is a required rcpFrame constant which supplies the reciprocal of the inputTexture size in pixels,

```
{ 1.0f/inputTextureWidth, 1.0f/inputTextureHeight, 0.0f, 0.0f }
```

Alias sRGB as UNORM

FXAA is engineered to be applied towards the end of engine post processing after conversion to low dynamic range and conversion to the sRGB color space for display.

Attempting to apply FXAA to a HDR image prior to tone-mapping will result in the same artifacts as resolving an MSAA surface prior to tone-mapping: anti-aliasing will effectively be turned off on edges which have both in-LDR-range and out-of-LDR-range intensity. For example a 0.0 intensity pixel and a 16.0 intensity pixel will average to a 8.0 intensity pixel which later might get tone-mapped to 1.0 intensity (which results in no blend between the two pixels).

FXAA requires the non-linear (perceptually encoded) RGB input and texture filtering: as a performance optimization, the algorithm's end-of-edge search step samples halfway between a pair of texels, and it is important that the fetch returns the perceptual blend of the two texels instead of the linearly correct blend.

When integrating FXAA to sample from a sRGB texture, modify the texture resource to be DXGI_FORMAT_R8G8B8A8_TYPELESS, and use a DXGI_FORMAT_R8G8B8A8_UNORM shader resource view for FXAA, and DXGI_FORMAT_R8G8B8A8_UNORM_SRGB shader resource views for the rest of the engine. Make sure to apply the same type aliasing when FXAA needs to write into a sRGB render target.

In the case of FXAA writing into a sRGB back-buffer, DX11 does not support type aliasing, so the conversion to linear will have to be done in the shader. Use the FXAA_SRGB_ROP define to enable this functionality in the included FXAA shader. This way FXAA can be applied and then UI elements can be composited into the back-buffer with linear correct blending.

Apply FXAA Prior to HUD/UI

FXAA should be applied prior to drawing the HUD or UI elements.

Optimized Integration

Depending on the engine, there may be a performance advantage in merging FXAA into an existing post processing pass. For example, the following can be done in a single full screen pass: FXAA + composite bloom results + color grading + adding film noise.

If the engine already has the ability to selectively apply post processing to only the regions which need a given effect, anti-aliasing, and thus FXAA, is likely not needed in areas under strong motion blur or depth of field.

Implementation

Algorithm Overview

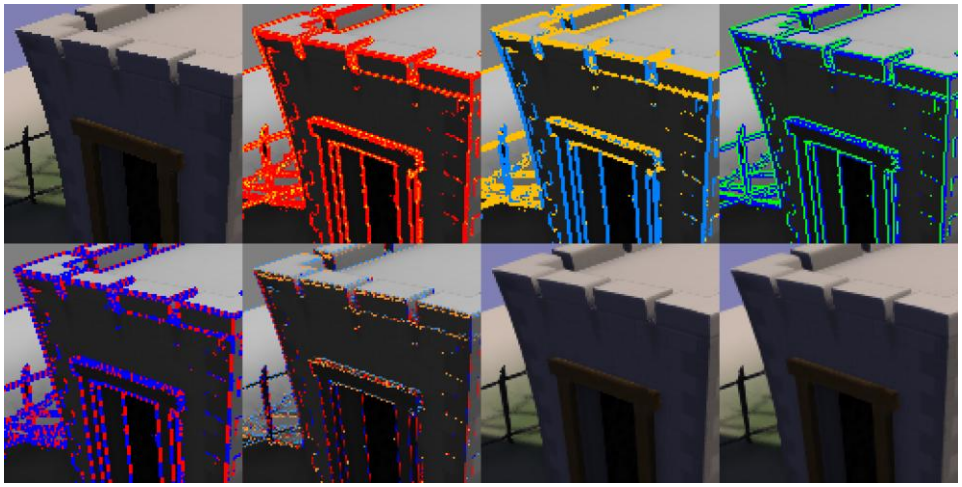


Figure 1: FXAA algorithm from right to left, top to bottom.

1. FXAA takes as input non-linear RGB color data which it converts internally into a scalar estimate of luminance for shader logic.
2. FXAA checks local contrast to avoid processing non-edges. Detected edges are in red, with blending towards yellow to represent the amount of detected sub-pixel aliasing (drawn using FXAA_DEBUG_PASSTHROUGH shader define).

3. Pixels passing the local contrast test are then classified as horizontal, in gold, or vertical, in blue (FXAA_DEBUG_HORZVERT).
4. Given edge orientation, the highest contrast pixel pair 90 degrees to the edge is selected, in blue/green (FXAA_DEBUG_PAIR).
5. The algorithm searches for end-of-edge in both the negative and positive, red/blue, directions along the edge. Checking for a significant change in average luminance of the high contrast pixel pair along the edge (FXAA_DEBUG_NEGPOS).
6. Given the ends of the edge, pixel position on the edge is transformed into to a sub-pixel shift 90 degrees perpendicular to the edge to reduce the aliasing, red/blue for -/+ horizontal shift and gold/skyblue for -/+ vertical shift (FXAA_DEBUG_OFFSET).
7. The input texture is re-sampled given this sub-pixel offset.
8. Finally a lowpass filter is blended in depending on the amount of detected sub-pixel aliasing.

Luminance Conversion

As an optimization, luminance is estimated strictly from Red and Green channels using a single fused multiply add operation. In practice pure blue aliasing rarely appears in typical game content.

```
float FxaaLuma(float3 rgb) {
    return rgb.y * (0.587/0.299) + rgb.x; }
```

Local Contrast Check

The local contrast check uses the pixel and its North, South, East, and West neighbors. If the difference in local maximum and minimum luma (contrast) is lower than a threshold proportional to the maximum local luma, then the shader early exits (no visible aliasing). This threshold is clamped at a minimum value to avoid processing in really dark areas.

```
float3 rgbN  = FxaaTextureOffset(tex, pos.xy, FxaaInt2( 0,-1)).xyz;
float3 rgbW  = FxaaTextureOffset(tex, pos.xy, FxaaInt2(-1, 0)).xyz;
float3 rgbM  = FxaaTextureOffset(tex, pos.xy, FxaaInt2( 0, 0)).xyz;
float3 rgbE  = FxaaTextureOffset(tex, pos.xy, FxaaInt2( 1, 0)).xyz;
float3 rgbS  = FxaaTextureOffset(tex, pos.xy, FxaaInt2( 0, 1)).xyz;
float lumaN  = FxaaLuma(rgbN);
float lumaW  = FxaaLuma(rgbW);
float lumaM  = FxaaLuma(rgbM);
float lumaE  = FxaaLuma(rgbE);
float lumaS  = FxaaLuma(rgbS);
float rangeMin = min(lumaM, min(min(lumaN, lumaW), min(lumaS, lumaE)));
```

```
float rangeMax = max(lumaM, max(max(lumaN, lumaW), max(lumaS, lumaE)));
float range = rangeMax - rangeMin;
if(range <
max(FXAA_EDGE_THRESHOLD_MIN, rangeMax * XAA_EDGE_THRESHOLD)) {
    return FxaaFilterReturn(rgbM); }
```

Tuning Defines

These defines can be used to optimize algorithm by allowing the algorithm to early exit and avoid processing.

FXAA_EDGE_THRESHOLD

The minimum amount of local contrast required to apply algorithm.

1/3 – too little

1/4 – low quality

1/8 – high quality

1/16 – overkill

FXAA_EDGE_THRESHOLD_MIN

Trims the algorithm from processing darks.

1/32 – visible limit

1/16 – high quality

1/12 – upper limit (start of visible unfiltered edges)

Sub-pixel Aliasing Test

Pixel contrast is estimated as the absolute difference in pixel luma from a lowpass luma (computed as the average of the North, South, East and West neighbors). The ratio of pixel contrast to local contrast is used to detect sub-pixel aliasing. This ratio approaches 1.0 in the presence of single pixel dots and otherwise begins to fall off towards 0.0 as more pixels contribute to an edge. The ratio is transformed into the amount of lowpass filter to blend in at the end of the algorithm.

```
float lumaL = (lumaN + lumaW + lumaE + lumaS) * 0.25;
float rangeL = abs(lumaL - lumaM);
float blendL = max(0.0,
    (rangeL / range) - FXAA_SUBPIX_TRIM) * FXAA_SUBPIX_TRIM_SCALE;
blendL = min(FXAA_SUBPIX_CAP, blendL);
```

The lowpass value used to filter sub-pixel aliasing at the end of the algorithm is a box filter of the complete 3x3 pixel neighborhood.

```
float3 rgbL = rgbN + rgbW + rgbM + rgbE + rgbS;
// ...
float3 rgbNW = FxaaTextureOffset(tex, pos.xy, FxaaInt2(-1,-1)).xyz;
float3 rgbNE = FxaaTextureOffset(tex, pos.xy, FxaaInt2( 1,-1)).xyz;
float3 rgbSW = FxaaTextureOffset(tex, pos.xy, FxaaInt2(-1, 1)).xyz;
float3 rgbSE = FxaaTextureOffset(tex, pos.xy, FxaaInt2( 1, 1)).xyz;
rgbL += (rgbNW + rgbNE + rgbSW + rgbSE);
rgbL *= FxaaToFloat3(1.0/9.0);
```

Tuning Defines

Other than turning the feature off or fully on, these defines do not effect performance. By default the amount of sub-pixel aliasing removal is limited. This enables fine features to remain, but at a low enough contrast so they are not distracting to the eye. Turing on full will blur the image.

FXAA_SUBPIX

Toggle subpix filtering.

0 – turn off

1 – turn on

2 – turn on force full (ignore FXAA_SUBPIX_TRIM and CAP)

FXAA_SUBPIX_TRIM

Controls removal of sub-pixel aliasing.

1/2 – low removal

1/3 – medium removal

1/4 – default removal

1/8 – high removal

0 – complete removal

FXAA_SUBPIX_CAP

Insures fine detail is not completely removed.

This partly overrides FXAA_SUBPIX_TRIM.

3/4 – default amount of filtering

7/8 – high amount of filtering

1 – no capping of filtering

Vertical/Horizontal Edge Test

Edge detect filters like Sobel fail on single pixel lines which pass through the center of a pixel. FXAA takes a weighted average magnitude of the high-pass values for rows and columns of the local 3x3 neighborhood as an indication of local edge amount.

```
float edgeVert =
    abs((0.25 * lumaNW) + (-0.5 * lumaN) + (0.25 * lumaNE)) +
    abs((0.50 * lumaW ) + (-1.0 * lumaM) + (0.50 * lumaE )) +
    abs((0.25 * lumaSW) + (-0.5 * lumaS) + (0.25 * lumaSE));
float edgeHorz =
    abs((0.25 * lumaNW) + (-0.5 * lumaW) + (0.25 * lumaSW)) +
    abs((0.50 * lumaN ) + (-1.0 * lumaM) + (0.50 * lumaS )) +
    abs((0.25 * lumaNE) + (-0.5 * lumaE) + (0.25 * lumaSE));
bool horzSpan = edgeHorz >= edgeVert;
```

End-of-edge Search

Given the local edge direction, FXAA pairs the pixel with its highest contrast neighbor 90 degrees to the local edge direction. The algorithm searches along the edge in the positive and negative directions until a search limit is reached or the average luminance of the pair moving along the edge changes by enough to signify end-of-edge.

A single loop searches in both the negative and positive direction in parallel on the chosen horizontal or vertical orientation. This is done to avoid divergent branching in the shader.

When search acceleration is enabled (presets 0, 1, and 2) the search is accelerated using anisotropic filtering as a box filter to check more than just a single pixel pair.

```
for(uint i = 0; i < FXAA_SEARCH_STEPS; i++) {
    #if FXAA_SEARCH_ACCELERATION == 1
        if(!doneN) lumaEndN = FxaaLuma(FxaaTexture(tex, posN.xy).xyz);
        if(!doneP) lumaEndP = FxaaLuma(FxaaTexture(tex, posP.xy).xyz);
    #else
        if(!doneN) lumaEndN = FxaaLuma(
            FxaaTextureGrad(tex, posN.xy, offNP).xyz);
        if(!doneP) lumaEndP = FxaaLuma(
            FxaaTextureGrad(tex, posP.xy, offNP).xyz);
    #endif
    doneN = doneN || (abs(lumaEndN - lumaN) >= gradientN);
    doneP = doneP || (abs(lumaEndP - lumaN) >= gradientN);
    if(doneN && doneP) break;
    if(!doneN) posN -= offNP;
    if(!doneP) posP += offNP; }
```

Tuning Defines

These defines have the greatest impact on performance. Note using search acceleration has the potential of causing some dithering in the edge filtering.

FXAA_SEARCH_STEPS

Controls the maximum number of search steps.

Multiply by FXAA_SEARCH_ACCELERATION for filtering radius.

FXAA_SEARCH_ACCELERATION

How much to accelerate search using anisotropic filtering.

1 – no acceleration

2 – skip by 2 pixels

3 – skip by 3 pixels

4 – skip by 4 pixels (hard upper limit)

FXAA_SEARCH_THRESHOLD

Controls when to stop searching.

1/4 – seems best quality wise

Porting to DX9 and OpenGL

The included "FxaaShader.h" file can also be used directly with DX9 and OpenGL.

DX9

Use the FXAA_HLSL_3 define for DX9, and adjust for the differences in Pixel Coordinate System for DX9 and DX11. FXAA pixel shader input "pos" needs to represent the texel center for a given pixel. Also to alias between sRGB and non-sRGB textures and render targets, use the following.

```
// sRGB->linear conversion when fetching from TEX
SetSamplerState(sampler, D3DSAMP_SRGBTEXTURE, 1); // on
SetSamplerState(sampler, D3DSAMP_SRGBTEXTURE, 0); // off

// linear->sRGB conversion when writing to ROP
SetRenderState(D3DRS_SRGBWRITEENABLE, 1); // on
SetRenderState(D3DRS_SRGBWRITEENABLE, 0); // off
```

OpenGL

For OpenGL 2.0 the GL_EXT_gpu_shader4 is required for the texture2DGrad() function. If necessary to support older versions of GL, the code could be modified to work without texture2DGrad() by removing the FXAA_SEARCH_ACCELERATION != 1 cases which use anisotropic filtering.

```
#version 120
#extension GL_EXT_gpu_shader4 : enable
#define FXAA_GLSL_120 1
```

For GLSL versions 130 and later, use the following FXAA_GLSL_130 define which does not require GL_EXT_gpu_shader4.

```
#define FXAA_GLSL_130 1
```

Frame buffer sRGB to non-sRGB aliasing is provided by the EXT_framebuffer_sRGB extension.

```
// linear->sRGB conversion when writing to ROP
glEnable(GL_FRAMEBUFFER_SRGB_EXT); // on
glDisable(GL_FRAMEBUFFER_SRGB_EXT); // off (default)
```

Texture sRGB to non-sRGB aliasing is provided by the EXT_texture_sRGB_decode extension, which can enable/disable for both textures and samplers. Texture enable/disable below.

```
// sRGB->linear conversion when fetching from TEX
glTexParameteri(
  GL_TEXTURE_2D, GL_TEXTURE_SRGB_DECODE_TEX,
  GL_DECODE_EXT); // on (default)
glTexParameteri(
  GL_TEXTURE_2D, GL_TEXTURE_SRGB_DECODE_TEX,
  GL_SKIP_DECODE_EXT); // off
```

Conclusion

FXAA provides another high performance and high quality option to combat aliasing in graphics engines.

Inspiration

FXAA was inspired by the AA work and up-coming work of many others,

"**Subpixel Reconstruction Antialiasing**", Matthäus G. Chajdas (Technische Universität München and NVIDIA), Morgan McGuire (NVIDIA), David Luebke (NVIDIA), to appear in i3D Febuary 2011

"**Morphological Antialiasing**", Alexander Reshetov (Intel Labs)
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, to appear in GPU Pro 2
http://www.iryoku.com/mlaa/

"**DLAA – DirectionaL Anti-Aliasing**", Dmitry Andreev (LucasArts), to appear in the "Anti-aliasing from a Different Perspective" talk at GDC 2011
http://schedule.gdconf.com/session/12294

"**Hybrid Anti-Aliasing**", Anton Kaplanyan (Crytek), presented in "CryENGINE 3: reaching the speed of light" at SIGGRAPH 2010
http://advances.realtimerendering.com/s2010/Kaplanyan-CryEngine3%28SIGGRAPH%202010%20Advanced%20RealTime%20Rendering%20Course%29.pdf

"**Deferred Rendering in Frameranger**", Matt Swoboda (SCEE), blog post which includes section on a reconstruction AA method for deferred rendering
http://directtovideo.wordpress.com/2009/11/13/deferred-rendering-in-frameranger/

"**DX11 Perspective Matrix Jittering Temporal AA**", blog post on Yakiimo3D
http://www.yakiimo3d.com/2010/09/28/dx11-perspective-matrix-jittering-temporal-aa/

"**Deferred MSAA**", Matt Pettineo (Ready At Dawn Studios)

http://mynameismjp.wordpress.com/2010/08/16/deferred-msaa/

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA, the NVIDIA logo, GeForce, and NVIDIA Quadro are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2010 NVIDIA Corporation. All rights reserved.