2.8 Deferred Shading with Multisampling Anti-Aliasing in DirectX 10

NICOLAS THIBIEROZ, ADVANCED MICRO DEVICES, INC.

INTRODUCTION

Deferred shading is a rendering technique now commonly used in games. *Ghost Recon: Advanced Warfighter I* and *II*, *S.T.A.L.K.E.R. Clear Skies*, *Splinter Cell 4*, *Tabula Rasa*, and numerous other titles released and in development have all been seduced by the inherent advantages of deferred shading. Other games have chosen a hybrid approach borrowing elements from deferred shading [Carsten07] [Sweeney07]. Unfortunately, one major drawback of these techniques is their inability to take advantage of the graphic hardware's full-screen multisampling anti-aliasing (MSAA) capabilities with legacy APIs such as DirectX 9 and OpenGL 2.1. As a result, games using these APIs have had to take drastic measures to overcome this limitation. In most cases a full-screen anti-aliasing option is simply not supported, leading to the obvious and dreaded "jaggies" inherent to aliasing issues. In other cases a custom full-screen anti-aliasing filter is proposed [Shishkovtsov05], whereby polygon edges are detected and selectively blurred. Unfortunately, the latter option can be quite a costly process, and is a poor-quality replacement for real MSAA due to using adjacent pixels from the fixed-resolution surface to perform the blurring.

The DirectX 10 graphics API from Microsoft finally includes the tools required to allow MSAA to be used robustly with deferred shading algorithms. In particular, the features introduced in DirectX 10.1 make it a fairly straightforward process, while DirectX 10.0 still requires some extra work to achieve identical results. This article provides details on optimized algorithms enabling MSAA to be used with deferred shading using those APIs.

Readers already familiar with the basic concepts of deferred shading may want to skip the section "Deferred Shading Principles" and start directly at "MSAA Requirements for Deferred Shading." On the other hand, readers wanting to read an in-depth description of the technique on legacy APIs are encouraged to consult previous literature on the subject, e.g., [Thibieroz04] [Calver03] [Pritchard04] [Hargreaves04].

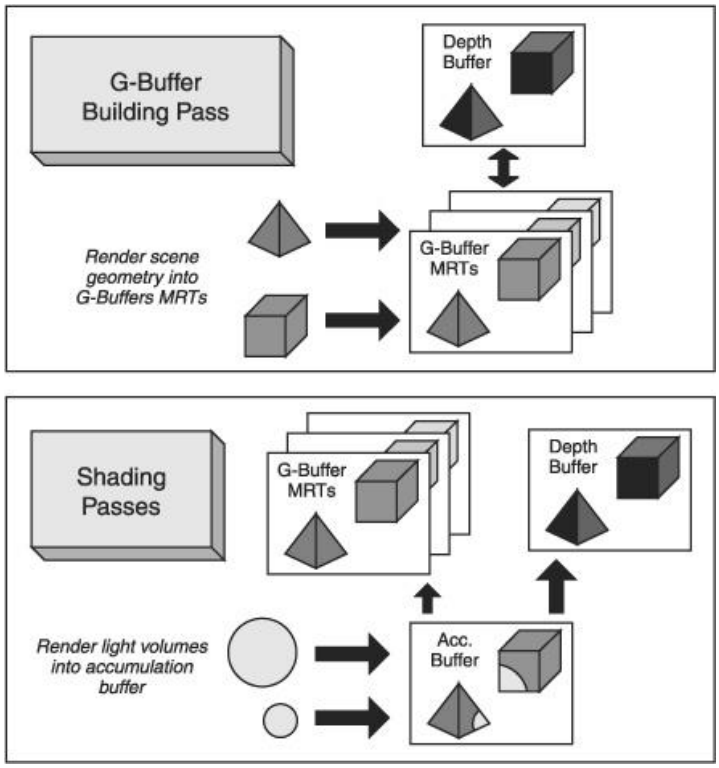
DEFERRED SHADING PRINCIPLES

Deferred shading is the concept of writing out geometric and material properties for every visible pixel in a scene into a collection of textures that are then subsequently fetched during later passes to apply shading operations on a per-pixel basis. The term *deferred shading* originates from the fact that shading operations are only performed once hidden surface removal has been determined and properties for all opaque pixels in the scene have been stored. This is different from a *forward renderer* whereby pixels are typically shaded as objects are being rendered with one or several geometry passes.

Deferred shading can be decomposed into two main phases (other common rendering phases such as post-processing are mostly agnostic to the rendering technique used and will therefore not be covered in detail here): the G-Buffer building pass and the shading passes. The G-Buffer building pass has the responsibility of storing the properties of every screen pixel, while the shading passes apply lighting or other shading effects by fetching the G-Buffer data. Each shading pass typically processes an area of pixels corresponding to the contributions of the current light. This allows efficient optimizations whereby the 2D projection of a light volume is rendered in order to limit the processing cost to only the pixels affected by the light. For example, a sphere volume would be rendered for point lights, while spot lights would typically use some kind of conic shape. In contrast, an ambient light affecting all pixels in the scene would require a full-screen quad (or triangle). This approach enables lights to be treated like any other objects in the 3D engine. Also, the decoupling of the G-Buffer creation phase from the lighting phase has beneficial results with regard to reducing pixel processing cost, reducing states changes, improving batch performance, and in general allowing a more elegant structure and management of the 3D engine. Because of those advantages, deferred shading is often considered ideal for scenes composed of a large number of lights. Figure 2.8.1 illustrates the deferred shading phases.

FIGURE 2.8.1 Deferred shading phases. The G-Buffer building phase writes geometric and material properties of the scene into a set of three arbitrary render targets making up the G-Buffer. The shading passes then render 2D projections of light volumes to shade pixels in range of the light. The pixel properties of each lit pixel are fetched from the G-

Buffer textures at a 1:1 mapping ratio, and the light equation is subsequently calculated and output into the accumulation buffer.

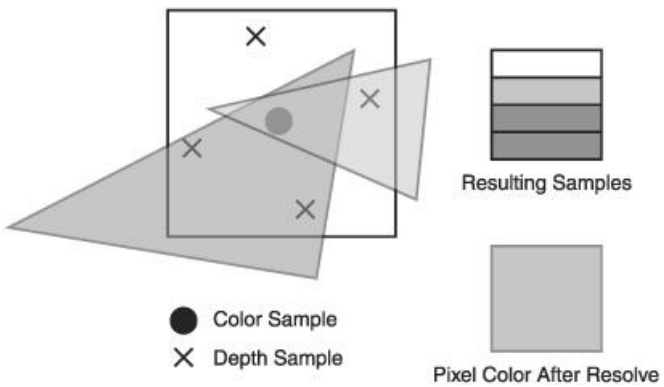


MSAA Requirements for Deferred Shading

MSAA Basics

MSAA allows a scene to be rendered at a higher resolution without having to pay the cost of shading more pixels. MSAA performs pixel shading on a *per-pixel* basis, while depth-stencil testing occurs on a *per-sample* basis. Samples successfully passing the depth-stencil test therefore have the pixel shader output stored into their respective entries in the multisampled render target. Once all rendering has been performed, the color samples for each pixel are averaged together (this process is called the *MSAA resolve* operation) to produce the final, anti-aliased pixel color. Figure 2.8.2 illustrates this concept.

FIGURE 2.8.2 Multisampling anti-aliasing. The pixel represented by a square has two triangles (blue and yellow) crossing some of its sample points. The black dot represents the pixel sample location (pixel center); this is where the pixel shader is executed. The cross symbol corresponds to the location of the multisamples where the depth tests are performed. Samples passing the depth test receive the output of the pixel shader. At resolve time sample colors are averaged together to produce the final, anti-aliased result.



Multisampling Multiple Render Targets

The G-Buffer writing phase of deferred shading algorithms relies on multiple render targets (MRTs) to store all the geometric and material properties for the scene. Typical properties such as depth, diffuse color, normal vector, gloss factor, and so on require a certain amount of destination storage that usually cannot be accommodated with a single render target. Therefore, binding multiple render targets during the G-Buffer writing phase is necessary, and in order to support MSAA with deferred shading, all those render targets must be rendered with MSAA. Unfortunately, real-time graphics APIs prior to DirectX 10 did not allow MSAA to work in conjunction with MRTs. One work-around to this problem is to build each G-Buffer's render target individually by binding it as the sole destination in its own geometry pass. However, the multiple passes (one per render target) required for this work-around have quite a high performance cost and defeat the single geometry pass advantage made possible by deferred shading in the first place. One other work-around would be to pack all required G-Buffer properties into a single "fat" render target format (e.g., 128-bits-per pixel formats). Once again, legacy API or hardware limitations with regard to supported formats and the availability of suitable packing and unpacking instructions impose too many limitations for this solution to be really useful.

DirectX 10 now allows MSAA to be used with up to eight multiple render targets. Although not strictly required by DirectX 10.0, existing DirectX 10 graphic hardware supports MSAA on a wide variety of useful render target formats such as 16-bit per channel floating-point surfaces, allowing greater flexibility for the G-Buffer configuration desired. Note that DirectX 10.1 pushes the minimum requirements further by imposing that four-sample MSAA be supported on 64-bit surfaces for compliant implementations.

Per-Sample Pixel Shader Execution

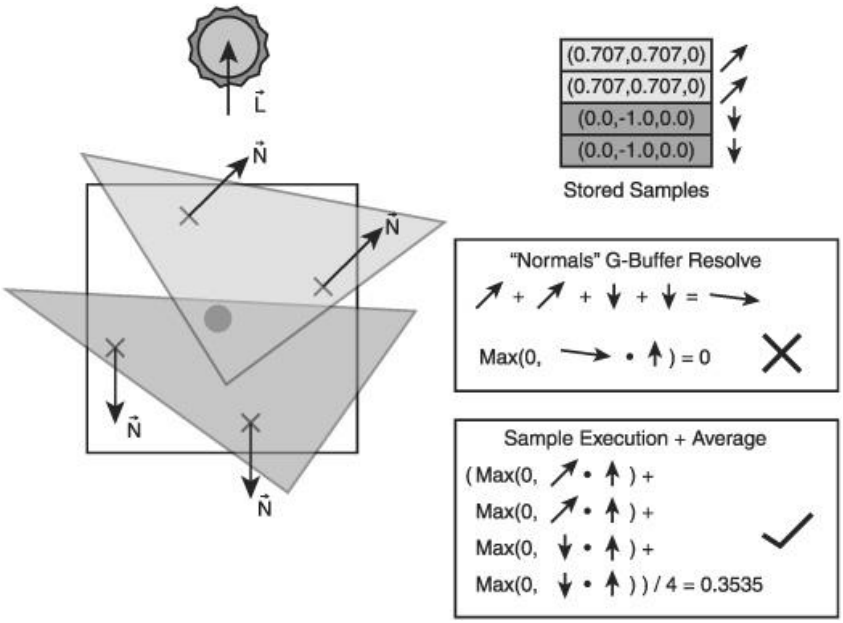
Although one may think that the ability to apply MSAA onto multiple render targets is enough to benefit from gorgeous anti-aliased visuals out of a deferred shading engine, the reality is quite different.

With a forward renderer each object is shaded and rendered directly into a multisampled surface that represents the colors of the completed scene. Once all objects have been rendered, a resolve operation is required to convert the multi-sampled render target into a final, anti-aliased image that can be queued up to the back buffer for the next `Present()` call.

In the case of deferred shading, the G-Buffer's multisampled render targets are simply intermediate storage buffers leading to the construction of the final image. They must therefore be preserved in multisampled form so that the color of each sample can be constructed using its unique per-sample properties stored in the G-Buffer. Once all shading contributions to the scene have been rendered onto the multisampled accumulation buffer, *then* the resolve operation can take place on this buffer to produce the final, anti-aliased image. In other words, the G-Buffer's multisampled render targets should not be resolved into anti-aliased versions of themselves before being used in the shading passes, as doing so will introduce visual artifacts on edge pixels.

Resolving G-Buffer properties like normal vectors or depth makes little sense, and will lead to incorrect results being produced. For example, if half the samples of a given pixel have different depth values stored in the G-Buffer, then the averaged depth produced by the resolve operation no longer has any relevance to the scene, and is very likely to cause incorrect lighting due to a completely new depth value being introduced. A similar case occurs when any samples in a given pixel have different normals: Even if a custom resolve operation is performed (e.g., to preserve normalized vectors through spherical interpolation), the issue remains that the vector produced by the resolve operation is a completely new quantity that did not arise from the G-Buffer writing phase, and therefore does not correspond to the geometric properties of the scene. Figure 2.8.3 provides an illustration of the difference arising when G-Buffer quantities are resolved prior to being fetched for the shading passes.

FIGURE 2.8.3 This figure illustrates what happens when G-Buffer normal vectors are resolved prior to being used in the light equation, instead of averaging the results of the light equation at each sample. Two triangles are rendered over a pixel, with each triangle covering two sample points. Resolving (averaging) the normals of all samples yields a vector that is now facing away from the light, even after renormalization. As a result, the light equation (a simple saturated dot product between the normal and the light vector) returns 0. In contrast, the correct method of applying the light equation on each sample and then averaging the results produces a positive value. Resolving quantities such as normal vectors or depth will result in incorrect colors on polygon edges, preventing any form of effective anti-aliasing.



To guarantee correct MSAA in a deferred shading context, the shading passes must therefore ensure that all calculations are performed on a per-sample basis and that the resulting per-sample output color is accumulated into a multisampled buffer. Because this "accumulation" buffer contains sample *colors* it will be subject to the final resolve operation once all shading and subsequent passes are done. To implement this functionality, the GPU must be able to execute a pixel shader at sample frequency: Input attributes are evaluated at a sample location (as opposed to a pixel location), the pixel shader code runs once for every sample, and each sample output is written out to a multisampled render target. DirectX 10.1 supports this feature directly and provides the fastest and most straightforward implementation. GPUs only supporting DirectX 10.0 can still implement this feature, but with less flexibility and performance, details of which are covered in the next section "Implementation."

Implementation

The following describes the basic steps of the algorithm allowing MSAA to be used with deferred shading and how they compare with the usual steps of rendering in single sample mode (i.e., without multisampling).

Creation of Multisampled Resources

The following resources need to be created as multisampled.

- The G-Buffer render targets
- The accumulation buffer receiving the contribution of shading passes and further rendering
- The depth-stencil buffer

The creation of multisampled render targets is done through the `CreateTexture2D()` API, by setting the `SampleDesc` field of the `D3D10_TEXTURE2D_DESC` structure to the desired multisampling configuration.

G-Buffer Writing Phase

There is very little difference between writing to single-sampled G-Buffer render targets and writing to multisampled ones. The only change required is to simply enable multisampling rasterization by creating and setting an appropriate rasterizer state. This can be done by enabling the `MultisampleEnable` field of the `D3D10_RASTERIZER_DESC` structure used during the creation of the state object. With the rasterizer state object set, all rendering is written out to the G-Buffer render targets in multisampled mode.

Edge Detection Phase

In order to produce accurate results for MSAA, it is essential that the pixel shaders used during the shading passes are executed at per-sample frequency. Executing a pixel shader for every sample has a significant impact on performance, though, especially if a high number of shading passes are rendered or a high number of multisamples are used. Instead of running the pixel shader code once per pixel, the code is now run as many times as there are samples (e.g., with 4x MSAA the pixel shader would execute four times: one for each sample). In order to avoid paying such a high performance cost, a sensible optimization is to detect pixels whose samples have different values and only perform per-sample pixel shader execution on those "edge" pixels. Remaining pixels whose samples share the same value would only be operated on at pixel frequency. By definition, MSAA produces differing samples on triangle edges; however, other rendering operations such as alpha-to-coverage or transparency anti-aliasing also output results on a per-sample granularity. In any case, the number of edge pixels is typically only a fraction of the total number of pixels in the scene, so the performance savings of only executing per-sample pixel shader code on those are fully justified.

The detection of edge pixels relies on the rendering of a full-screen quad (or triangle) with a pixel shader used to determine whether samples fetched from the G-Buffer are identical or not. Different methods can be used for pixel edge detection. For a perfect result, one would have to fetch and compare all samples of all G-Buffer textures. However, this approach can cause a performance concern due to the numerous texture fetches it requires.

The output of the edge detection pass is written out to the stencil buffer. After being cleared to 0, the stencil buffer sets the top stencil bit of every sample if a pixel edge is detected. For this to work, the pixel shader must discard non-edge pixels, and "let through" the ones detected as edges. Note that no color buffer is bound to this render pass, as the only output required is stored in the stencil buffer. Listing 2.8.1 shows the code used to detect pixel edges.

Listing 2.8.1 Pixel shader code used for pixel edge detection

```
// Pixel shader to detect pixel edges

// Used with the following depth-stencil state values:

// DepthEnable =    TRUE

// DepthFunc =      Always

// DepthWriteMask = ZERO

// StencilEnable =  TRUE

// Front/BackFaceStencilFail =      Keep

// Front/BackfaceStencilDepthFail = Keep

// Front/BackfaceStencilPass =      Replace;

// Front/BackfaceStencilFunc =      Always;

// The stencil reference value is set to 0x80

float4 PSMarkStencilWithEdgePixels( PS_INPUT input ) : SV_TARGET

{

    // Fetch and compare samples from GBuffer to determine if pixel

    // is an edge pixel or not
      bool bIsEdge = DetectEdgePixel(input);



      // Discard pixel if non-edge (only mark stencil for edge pixels)

      if (!bIsEdge) discard;



      // Return color (will have no effect since no color buffer bound) return

      float4(1,1,1,1);

}
```

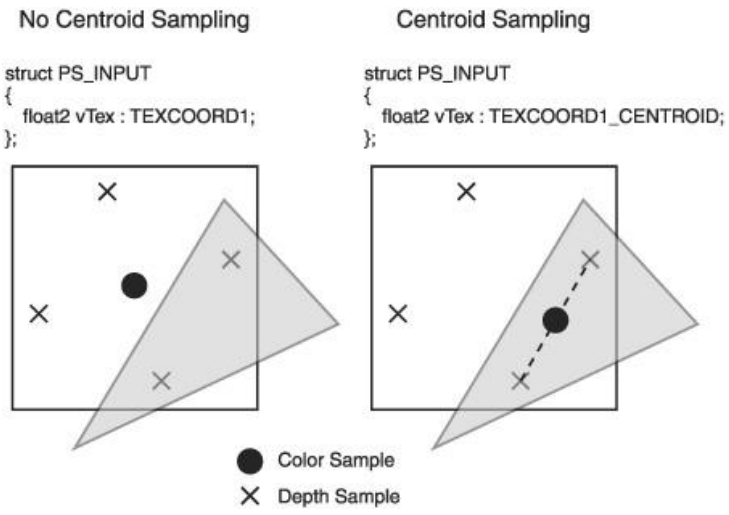
`DetectEdgePixel()` has the responsibility of determining whether the current pixel is an edge pixel. If the function returns FALSE (non-edge), then all samples are equal, and the pixel will be discarded, leaving the default value of 0 in all stencil samples for this pixel. If the function returns TRUE, then the pixel continues its way in the graphics pipeline, causing a 0×80 value to be written out to the stencil samples.

Centroid-Based Edge Detection

An optimized way to detect edges is to leverage the GPU's fixed function resolve feature. Because this process is usually hard-coded in the hardware, it produces a resolved output faster than the equivalent "custom" approach requiring individual samples to be fetched through the use of dedicated instructions. The edge detection method used relies on the centroid functionality that was introduced in DirectX 9. Centroid sampling is used to adjust the sample position of an interpolated pixel shader input so that it is contained within the area defined by the multisamples covered by the triangle. Centroid sampling is generally useful in cases where sampling could generate incorrect results on edge pixels when the triangle does not cover the pixel center. DirectX 10 allows pixel shader inputs to be evaluated at a centroid location by appending the keyword `_CENTROID` to the semantics declaration. Figure 2.8.4 illustrates the concept of declaring an iterated pixel shader input as centroid and how it compares with the default mode.

Centroid sampling can be used to determine whether a sample belongs to an edge pixel or not. A vertex shader writes a variable unique to every pixel (e.g., position data) into two outputs, while the associated pixel shader declares two inputs: one without and one with centroid sampling enabled. During the G-Buffer building phase the pixel shader then compares the centroid-enabled input with the one without it: Differing values mean that samples were only partially covered by the triangle, indicating an edge pixel. A "centroid value" of 1.0 is then written out to a selected area of the G-Buffer (previously cleared to 0.0) to indicate that the covered samples belong to an edge pixel. Once the G-Buffer writing phase is complete, a fixed-function resolve is performed to average the G-Buffer render target containing the centroid value. If this value is not exactly 0, then the current pixel is an edge pixel.

FIGURE 2.8.4 On the left the iterated pixel shader input vTex is interpolated normally: This pixel shader input will always be evaluated at the center of the pixel regardless of whether it is covered by the triangle. On the right the same input is declared with the _CENTROID keyword, forcing a centroid evaluation of the input. Since the two rightmost depth samples are covered by the triangle, the centroid sample location is contained within this area (typically midway between the two points).



This MSAA edge detection technique is quite fast, especially compared to a custom method of comparing every G-Buffer normal and depth samples. It only requires a few bits of storage in a G-Buffer render target. The number of bits reserved for this purpose must be large enough to accommodate the result of the fixed-function resolve operation without triggering a loss of precision that could lead to false negatives. The number of storage bits required is therefore as large as the bit encoding needed to represent the number of multisamples used, plus one bit to avoid any rounding-to-zero behavior. Thus, 4 bits of storage in the G-Buffer is enough to store edge detection data for up to 8x multisampling anti-aliasing.

For example, if only one sample out of eight was to receive the centroid value of 1.0 due to being the only one covered by a triangle, then the resolved (averaged) value of all samples belonging to this pixel would be 1 + 0 + 0 + 0 + 0 + 0 + 0 + 0 = 1 / 8 = 0.125. This result quantizes to a value between 0001b and 0010b in 4-bit binary representation, enough to guarantee a non-zero result. Since only a few bits are required to store the centroid value for each sample, the remaining bits available in the selected G-Buffer render target channel can be used to pack other properties, for example, a bit field indicating various attributes of the pixel material. In this case the resolve process still safeguards the result of the centroid value resolve, and all that's needed is to adjust the edge test accordingly. For example, a pixel detected as an edge would add 240/255 to an 8-bit destination channel so that only the four top bits are used. After resolve, the test simply has to determine whether the averaged

centroid value is above 15 to indicate an edge.

Unfortunately, this method cannot be used to detect non-polygonal edges, so it will not work with techniques like alpha-to-coverage or multisampled alpha testing. If such features are required, then an alternative edge detection algorithm relying on pure detection of differing samples should be used on such primitives.

Shading Passes

The shading passes are the “meat” of the algorithm, as this is where most of the changes to enable MSAA are required. The purpose of the shading passes is to add light contributions to the accumulation buffer. Light volumes are typically rendered as 2D projections in screen space so that only pixels affected by the light are processed, enabling significant performance savings. One way to do this is by using a two-sided stencil to mark the pixels inside a volume in a first “pre-pass” and then applying the light equation onto only those marked pixels. This technique has the advantage of generating a perfect “mask” of all pixels affected by the light. Another method commonly used is to detect whether the camera is inside or outside a convex light volume. If outside, the front faces of the volume are rendered; if inside, then the back faces of the volume are rendered with an inverted depth test. The latter approach has the advantage of not requiring an additional pass or any stencil testing, but it may be less effective when the camera is outside the volume due to the possibility of false positives being generated.

Regardless of the method used to render light volumes, the use of MSAA with deferred shading relies on the ability to apply lighting equations on a per-sample basis. In order to avoid redundant and costly processing, this operation is only performed on samples belonging to pixel edges, as detected in the previous phase. Such samples are therefore processed individually by the pixel shader, with their output written out to their respective samples in the accumulation buffer. Because the remaining non-edge pixels share the same sample values, they are then rendered at pixel frequency, which means the result of the pixel shader calculation is output to all samples passing the depth-stencil test.

Per-Sample Pixel Shader Execution: The DX10.1 Method

Shader model 4.1 as available in DX10.1 exposes a feature allowing the execution of a pixel shader at sample frequency, enabling single-pass processing of all samples belonging to edge pixels. To request a pixel shader to run at sample frequency, the sample keyword must be declared on at least one of the interpolated pixel shader inputs, or the SV_SAMPLEINDEX semantic must be used. Below is an example pixel shader input structure that will trigger a per-sample pixel shader execution.

```
struct PS_INPUT_EDGE_SAMPLE

{
    float4 Pos   : SV_POSITION;

    uint uSample : SV_SAMPLEINDEX;

};
```

The uSample input declared with the SV_SAMPLEINDEX semantic is a *system value* that returns a zero-based index of the sample being executed by the pixel shader. This index is used by the pixel shader code to fetch data from the G-Buffers corresponding to the sample being processed. The pixel shader code therefore executes once for every sample using the inputs provided at each iteration. Access to an individual texture sample is achieved by using the Load() shader instruction, which takes the pixel screen coordinates and the sample number to retrieve as input parameters. Listing 2.8.2 shows the pixel shader used during the shading passes to process edge samples.

To ensure that only samples belonging to edge pixels are processed, the stencil test has to be set up accordingly before running the aforementioned pixel shader. This can be done by simply setting the stencil test properties so that the test passes if the stencil buffer value equals the value written to the stencil buffer during the edge detection pass (0×80 was used in Listing 2.8.2). The depth test to use depends on whether the camera is inside or outside the light volume.

Listing 2.8.2 The pixel shader applies the light equation on a per-sample basis for all edge samples. The sample index is used to fetch sample properties from the multisampled G-Buffer textures. The pixel shader output is written to the sample corresponding to the current sample being processed

```
// Multisampled G-Buffer textures declaration

Texture2DMS <float4, NUM_SAMPLES> txMRT0;

Texture2DMS <float4, NUM_SAMPLES> txMRT1;
```

```
Texture2DMS <float4, NUM_SAMPLES> txMRT2;
  // Pixel shader for shading pass of edge samples in DX10.1

  // This shader is run at sample frequency


  // Used with the following depth-stencil state values so that only

  // samples belonging to edge pixels are rendered, as detected in

  // the previous stencil pass.

  // StencilEnable =   TRUE

  // StencilReadMask = 0x80

  // Front/BackFaceStencilFail =      Keep

  // Front/BackfaceStencilDepthFail = Keep

  // Front/BackfaceStencilPass =      Keep;

  // Front/BackfaceStencilFunc =      Equal;

  // The stencil reference value is set to 0x80

  float4 PSLightPass_EdgeSampleOnly( PS_INPUT_EDGE_SAMPLE input ) : SV_TARGET

  {

      // Convert screen coordinates to integer

      int3 nScreenCoordinates = int3(input.Pos.xy, 0);


     // Sample G-Buffer textures for current sample

     float4 MRT0 = txMRT0.Load( nScreenCoordinates, input.uSample);

     float4 MRT1 = txMRT1.Load( nScreenCoordinates, input.uSample);

     float4 MRT2 = txMRT2.Load( nScreenCoordinates, input.uSample);


     // Apply light equation to this sample

     float4 vColor = LightEquation(MRT0, MRT1, MRT2);


    // Return calculated sample color

    return vColor;

}
```

Per-Sample Pixel Shader Execution: The DX10.0 Method

DirectX 10.0 does not support the concept of running a pixel shader at sample frequency. However, it is still possible to achieve an identical result by adopting a multi-pass approach. The idea is to render the light volumes as many times as the number of samples, only enabling output to a single sample each pass. Although the pixel shader is then run per-pixel, the OMSetBlendState() API is set up to ensure that only one sample is written out. The pixel shader to execute at each pass fetches the G-Buffer samples corresponding to the sample currently being processed. Listing 2.8.3 shows the pixel shader code to achieve this result.

This method produces results identical to the DX10.1 method detailed in the previous section, at the cost of extra render passes and slightly reduced texture cache effectiveness due to repeated access of the same G-Buffer areas multiple times.

Listing 2.8.3 Pixel shader code run as part of a multi-pass rendering operation to emulate per-sample execution in DX10.0

```
// Multisampled G-Buffer textures declaration

Texture2DMS <float4, NUM_SAMPLES> txMRT0;

Texture2DMS <float4, NUM_SAMPLES> txMRT1;

Texture2DMS <float4, NUM_SAMPLES> txMRT2;
```

```
// Pixel shader for shading pass of edge samples in DX10.0

// This shader is run at pixel frequency and is executed once for

// each sample as part of a multipass operation.

// A single sample will be output each pass through the use of:
// pDev10->OMSetBlendState(&BS, &BlendFactor, 1<<nCurrentSample);


// Used with the following depth-stencil state values so that only

// samples belonging to edge pixels are rendered, as detected in

// the previous stencil pass.

// StencilEnable =   TRUE

// StencilReadMask = 0x80

// Front/BackFaceStencilFail =      Keep

// Front/BackfaceStencilDepthFail = Keep

// Front/BackfaceStencilPass =      Keep;

// Front/BackfaceStencilFunc =      Equal;

// The stencil reference value is set to 0x80

float4 PSLightPass_SingleSampleOnly( float4 Pos : SV_POSITION,
                                     uniform uint nSample )
                                    : SV_TARGET
  {

    // Convert screen coordinates to integer

    int3 nScreenCoordinates = int3(Pos.xy, 0);


    // Sample G-Buffer textures for input sample

    float4 MRT0 = txMRT0.Load( nScreenCoordinates, nSample);

    float4 MRT1 = txMRT1.Load( nScreenCoordinates, nSample);

    float4 MRT2 = txMRT2.Load( nScreenCoordinates, nSample);


    // Apply light equation to this sample

    float4 vColor = LightEquation(MRT0, MRT1, MRT2);


    // Return calculated color

    return vColor;

}
```

Each light volume from the shading passes is rendered as many times as there are samples in the chosen multisampling configuration. The global sample mask is set up so that only the sample corresponding to the current pass is output into the multisampled accumulation buffer. The light equation is only applied to the selected sample by fetching the appropriate G-Buffer samples. Note that the Load() API requires the sample index to be a literal value, so several versions of this shader should be compiled (one for every sample index).

Non-Edge Pixels

Regardless of the method used (DX10.1 or DX10.0), once all samples belonging to edge pixels have been rendered into the accumulation buffer, it is the turn of non-edge pixels to add their contribution. By definition, non-edge pixels share the same data across all samples, so the pixel shader need only fetch G-Buffer data for the first sample. The pixel shader code is therefore fairly straightforward and is a simple modification to Listing 2.8.3 whereby the sample to fetch is hard-coded to be sample 0. To ensure that only non-edge pixels are processed, the stencil test is set up differently so that the stencil test passes if bit 8 is *not* equal to the reference value (still set to 0×80).

Once all light volumes have been rendered, the multisampled accumulation buffer contains color samples representing the scene; it is therefore suited to any further operations required by the graphic engine and can be resolved as normal prior to being used in some post-processing operations or before the final copy to the back buffer.

Assessment of Alternative Implementation

On-the-Fly Resolves

It can be tempting to think that per-sample execution frequency is not strictly required for correct multisampling of deferred shading engines and that instead, an "on-the-fly" resolve approach could be taken to add per-pixel color contributions during the shading passes. The on-the-fly resolve approach consists in fetching all the samples belonging to a pixel, calculating their respective color contributions, and then averaging (resolving) the results so that a single pixel color is accumulated into a render target. This technique presents the advantage that only per-sample access to multisampled resources is required for it to work (which is supported on all DirectX 10.0 implementations), but it can also be subject to significant inefficiencies and even errors.

First, the render target used for accumulation of the shading passes' output will still need to be multisampled despite writing whole pixel color values. This is because the depth buffer itself is multisampled, and it is not possible to mix a multisampled depth buffer with a non-multisampled color render target. The depth buffer is quite essential to improve the performance of shading passes through Z culling optimizations, so not binding it is not an option.

The second issue requires a good knowledge of DirectX 10 multisampling rules to be understood. It is important to point out that a notable difference between having multisampling enabled or not is the value of the incoming depth used as input to the depth-stencil test. With multisampling disabled, the same interpolated depth value (evaluated at the pixel center) is input to the depth test of each sample. With multisampling enabled, the interpolated depth is evaluated at each sample, providing a unique input depth to the depth test of each sample. In *both* cases depth tests, stencil tests, and backend operations such as blending operate on a *persample* basis. When rendering a light volume with depth or stencil optimizations, there will be cases where some samples of a pixel pass the depth-stencil tests while others do not (this will happen along the edges of objects inside the volume and regardless of whether MSAA is enabled or not). Unfortunately, the pixel shader used with the on-the-fly resolve method averages all samples regardless of whether they actually pass the depth-stencil test, and output the resolved value into the samples that passed the depth test. The multisampled accumulation buffer may therefore contain different values per sample when it was supposed to contain a unique value representing the average of all samples. This requires a resolve operation at the backend to take into account the value of all samples; unfortunately, this process creates lower color intensities on edge pixels (by averaging some samples containing values that are already the result of an averaging operation), resulting in a considerable reduction in anti-aliasing effectiveness as illustrated on the right-hand side of Figure 2.8.5.

FIGURE 2.8.5 Comparison of rendering method used during the shading passes. On the left the pixel shader outputs an individual per-sample color for every sample belonging to edge pixels, resulting in efficient anti-aliasing. On the right the shading is performed per-pixel by averaging lighting contributions of all samples (on-the-fly resolve). Notice the reduced effectiveness of anti-aliasing for the latter method due to darker colors being produced.

It is possible to overcome this problem by ensuring that pixel edge samples are always fully covered by a light volume; in this case the result of the on-the-fly resolve will be written out to

all samples, and the final resolve need only take a single sample to produce the final output. Unfortunately, ensuring that pixel edge samples are always covered means that early depth or stencil culling optimizations may not be used (as they will produce per-sample outputs), so a light volume would have to be rendered integrally (e.g., using an alternative technique such as scissoring). Naturally, this will increase the fill-rate and pixel processing cost and should therefore not be considered a good solution.

The last issue with this method is the inability to perform further drawing operations requiring per-sample access after the shading passes. This limitation can be drastically restrictive, for example, when high-dynamic-range-correct rendering needs to be applied to the scene [Persson08].

Conclusion

This article has shown how to implement a correct deferred shading implementation in combination with multisampling anti-aliasing. Because of the significant memory cost associated with the creation of multisampled resources, it pays off to be economical on the G-Buffer configuration chosen. Reducing the number of render targets and their bit depth will not only provide savings in video memory but will also result in higher performance. The use of a proper edge detection filter is also essential to the implementation since executing shading passes on too many samples rapidly results in unacceptable performance issues.

References

[Thibieroz04] Nicolas Thibieroz, "Deferred Shading with Multiple Render Targets," *ShaderX^2: Shader Programming Tips & Tricks with DirectX^9*, Wolfgang Engel, ed., Wordware Publishing, 2004, pp. 251–269.

[Calver03] Dean Calver, "Photo-Realistic Deferred Lighting," white paper, available at http://www.beyond3d.com/content/articles/19

[Shishkovtsov05] Oles Shishkovtsov, "Deferred Shading in S.T.A.L.K.E.R.," *GPU Gems 2*, Matt Pharr, ed., Addison Wesley, 2005, pp. 143–166.

[Pritchard04] Rich Geldreich, Matt Pritchard, & John Brooks, "Deferred Lighting and Shading," GDC 2004 presentation, available at: http://www.gdconf.com/conference/archives/2004/pritchard_matt.ppt

[Hargreaves04] Shawn Hargreaves, "Deferred Shading," presentation, GDC 2004, available at http://ati.amd.com/developer/gdc/D3DTutorial_DeferredShading.pdf

[Carsten07] Carsten Wenzel, "Real-time Atmospheric Effects in Games Revisited," GDC 2007 presentation, available at http://developer.amd.com/assets/D3DTutorial_Crytek.pdf

[Sweeney07], EVGA Gaming, "Q&A with Tim Sweeney," available at http://www.evga.com/gaming/ gaming_news/gn_100.asp

[Persson08] Emil Persson, "Post-Tonemapping Resolve for High-Quality HDR Anti-aliasing in D3D10," *ShaderX^6: Advanced Rendering Techniques*, Wolfgang Engel, ed., Course Technology CENGAGE Learning, 2008, pp. 161–164.