2.2

Alpha-to-Coverage in Depth

Kevin Myers, **NVIDIA Corporation**

Realistic simulation of the real world depends on realistic rendering of highly detailed organic objects. Natural-looking vegetation, lifelike fur, as well as complex particle systems for smoke and clouds are among those objects. Physically based descriptions of such objects relying on geometric data are onerous for both artists and efficient hardware implementations. Modern GPUs are excellent at pixel shading, but not exceptionally adept at handling a large number of small triangles. Owing to these limitations, many solutions that stress fragment-level shading have been developed over the years.

A simple solution that is well understood is alpha testing (Figure 2.2.1). With alpha testing a fourth 1-bit color channel, alpha, is used to parameterize the image. With more than 1 bit of alpha we can define a ratio of source color to destination color (Figure 2.2.2). The common algorithm for so-called alpha blending is dest color = source color * source alpha + current dest color * (1 – source alpha). As you can see, this calculation is not commutable; order does matter. This burdens us with maintaining strict back to front rendering, which also forces us to sort all blended objects. Blending also reduces pixels throughput, typically by half, because of the overhead of reading back the current pixel's color.

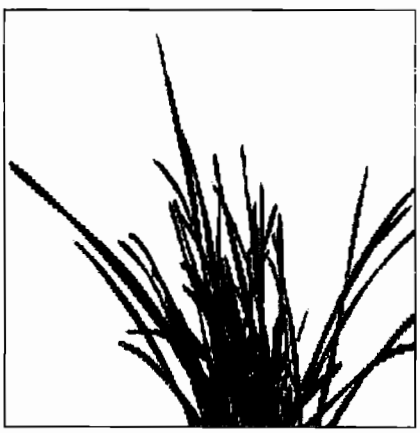

FIGURE 2.2.1 Alpha testing results.

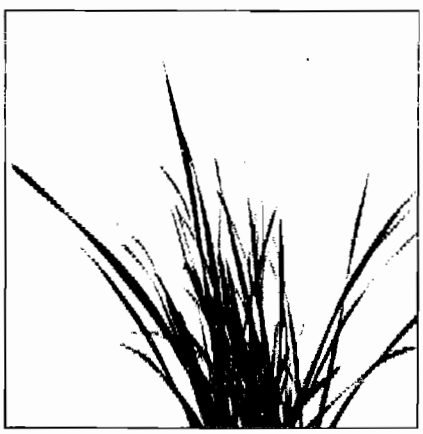

FIGURE 2.2.2 Alpha Blending results.

A simpler idea that does not rely on in-order rendering is screen door transparency. This technique gets its name from the observation that a screen covering a window will darken the view through the window as the eye interpolates the color of the screen with the color behind the screen. Screen door transparency relies on the human eye's ability to fill in the gaps between discrete samples. The more solid the screen is, the less background color gets through. If the screen varies in transparency, we can achieve different levels of blending. This is the basis for modern alpha-to-coverage.

Alpha-to-coverage is screen door transparency applied at the subpixel level through multisampling. API support exists in both OpenGL and DirectX 10. As a reminder, multisampling rasterizes more than one sample (typically four) per pixel, while shading once for all subsamples. This produces smooth polygon edges without the overhead of additional shading. When using alpha-to-coverage, multisampling is handled as usual, but the alpha value is used to decrease coverage when alpha is less than 1. The result can be seen in Figure 2.2.3. Notice the softly dithered edges compared to the hard edge in the case of simple alpha testing. Figure 2.2.4 shows a close-up view of one of the blades. One can clearly see the hard edge that occurs if alpha is interpreted as 0 or 1. In Figure 2.2.5 alpha is used to determine coverage. The black holes represent where alpha was determined to be below a particular threshold, therefore preventing a sample from being generated. Remember, this is being done at the antialiased resolution. Blending occurs during the multisample resolve, where the image is down-sampled to the display's resolution. Figure 2.2.6 shows a blown up picture of the final leaf.

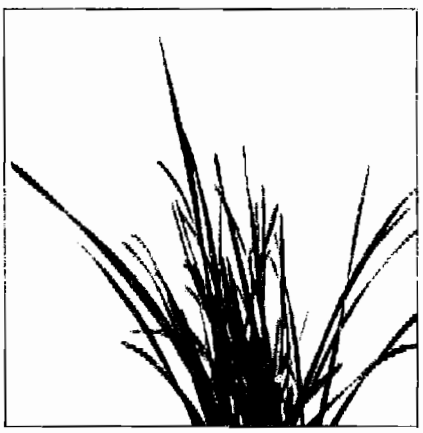

FIGURE 2.2.3 Alpha to coverage result.

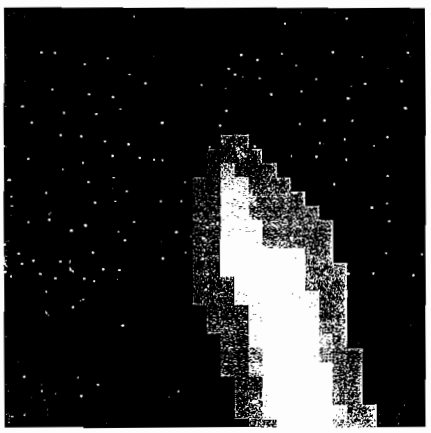

FIGURE 2.2.4 Close up view of one blade.

Alpha-to-coverage is then limited in quality by the number of subsamples. With only four subsamples per pixel, it would seem we could only have five discrete levels of coverage (zero to four samples covered). Fortunately we can do better. Figure 2.2.7

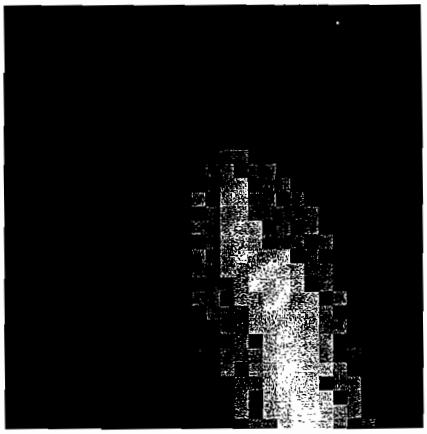

FIGURE 2.2.5 Alpha is used to determine coverage.

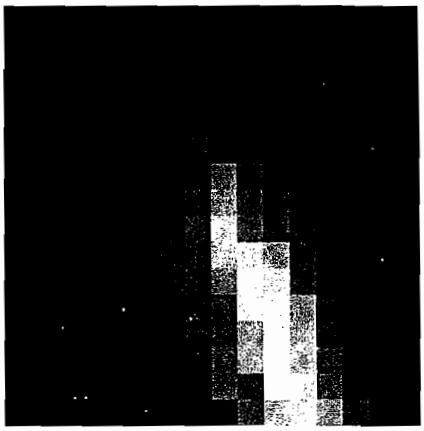

FIGURE 2.2.6 Alpha to coverage result of one blade.

was produced by linearly increasing alpha from right to left with 4x MSAA and alpha-to-coverage on. As we can see, there are clearly more than five levels of coverage. In order to get more levels of coverage than existing subsamples, an implementation must dither coverage levels at the pixel level and not the subsample level. Alpha-to-coverage as dictated by the OpenGL specification simply says coverage must be 0 when alpha == 0 and 1 when alpha == 1. Between 0 and 1 the specification indicates that coverage must increase, but it does not specify how it must increase. Various implementations then have freedom in how this is implemented as long as these three rules are followed.

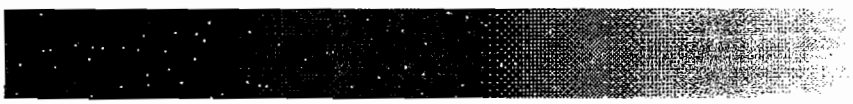

FIGURE 2.2.7 Result of linearly increasing alpha from right to left with 4x MSAA.

What can be gleaned from the spec is that alpha-to-coverage differs from normal screen-door transparency in its inability to handle multiple layers of transparency effectively. Because the spec only states that coverage should increase with increasing alpha, two triangles with the same alpha value will occupy the same samples. In Figure 2.2.8 two triangles with the same alpha are rasterized to the same pixel, represented by the four subsamples inside the square. We first render the pink triangle and then the blue triangle. Since both have an alpha value of 0.5, only two samples will be written, but they are the same two each time. This is why we only see blue and the

background color in our subsamples. This is a serious limitation if we are trying to use alpha-to-coverage to simulate general blending. Particle simulations that depend heavily on alpha blending will not benefit from alpha-to-coverage because only the most recently written layer's color will survive.

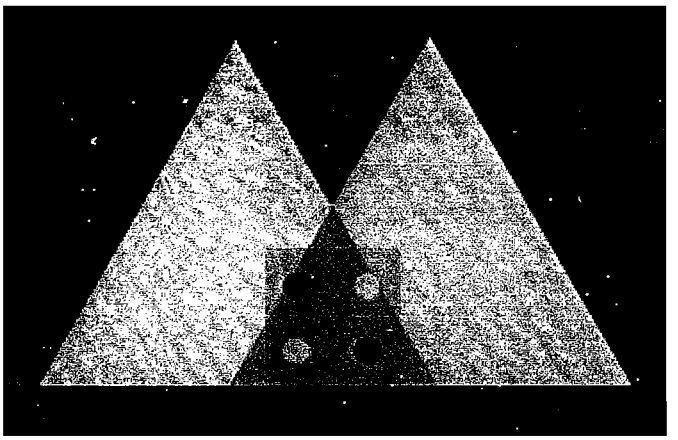

FIGURE 2.2.8 Two triangles with the same alpha.

This doesn't mean alpha-to-coverage is useless (or we wouldn't have this article). Alpha-to-coverage is especially useful when alpha is used to define geometric primitives, as often occurs with foliage. In this case, alpha is equal to 1 wherever the primitive is defined, and 0 elsewhere. Only at the edges is alpha < 1 and alpha > 0. It is here that we get interesting coverage. As is shown in Figure 2.2.3, as alpha tapers off to 0, more and more of the background will fill the samples not occupied by the blades of grass. When the multisample resolve occurs, the leaf edge will be blended with the background to create a soft transition without aliasing. The limitations cited before are not of great concern in this case because the region where alpha-to-coverage is occurring is very small (only at the edges), thus reducing the chance for multiple layers to intersect.

This works great for vegetation, but what about smoke or other particle systems? In Figure 2.2.9 we see the result of alpha-blending several quads to create a smoke cloud. Figure 2.2.10 shows what happens when alpha-to-coverage is used instead of blending. Notice the mid-section of the cloud. A couple of quads are completely missing because the final quads killed off the quads in the back. One way to work around the limitation is to use commutative blending in addition to alpha-to-coverage. Instead of doing typical src alpha/inv src alpha blending, we do simple additive blending where destination color = current dest color + source for color. Using multiple render targets which work with AA in DirectX 10, we maintain a pixel count per pixel

by incrementing a single-channel texture. This allows us to, in the final pass, renormalize color to avoid saturating color. The result is Figure 2.2.11, which more closely mimics the alpha-blended reference image. This renormalization process was done to an FP16 render target. To mitigate the dithering artifacts, in Figure 2.2.12 we applied a filter that takes the current pixel and the four neighbors, blending them together if they contain smoke.

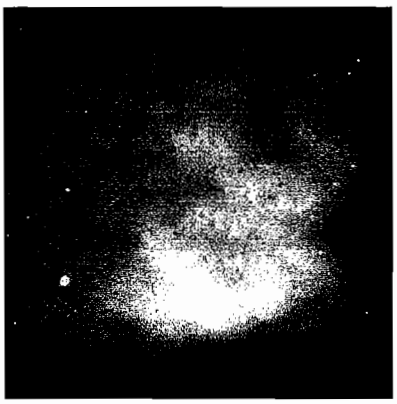

FIGURE 2.2.9 Alpha blending to create a smoke cloud.

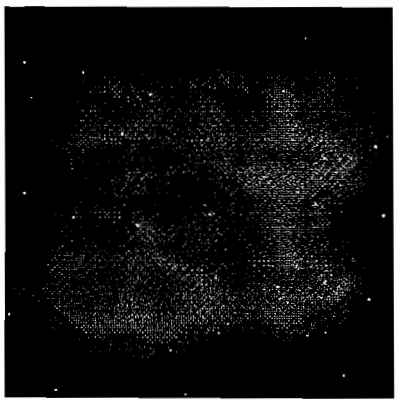

FIGURE 2.2.10 Alpha to coverage to create a smoke cloud.

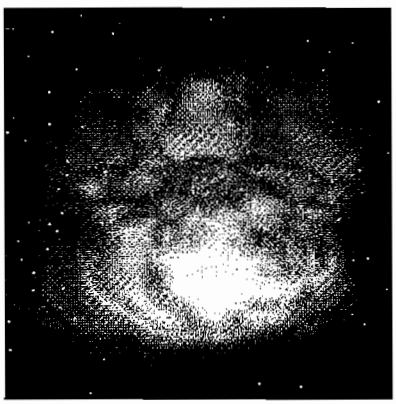

FIGURE 2.2.11 Alpha to coverage and commutative blending.

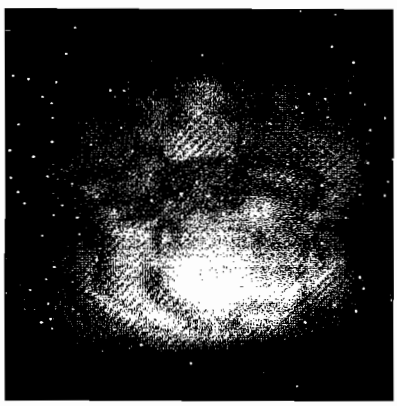

FIGURE 2.2.12 Alpha to coverage, commutative blending and filtering the result.

Another common artifact with alpha-to-coverage is shimmering with very far away or very small primitives. This will vary from implementation to implementation but is the result of dithering being used to achieve a greater variance in coverage. Since small triangles are inherently under-sampled (they represent a small portion of the screen), alpha values tend to jump rather quickly at the edges of such primitives. This

in turn causes rapid variations in coverage as the alpha value assumes vastly different coverage values. One way to mitigate this problem is to supersample small, distant primitives. One can also adjust the mip-chain such that the smaller mip levels are more blurred. The important thing to remember is that alpha directly relates to coverage and coverage directly relates to the blended pixel.

References

Mulder, J. D., F. C. A. Groen, and J. J. van Wijk, *Pixel Masks for Screen-Door Transparency,* IEEE Computer Society Press, 1998.

OpenGL 2.0 Spec. Available online at *http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf#search=%22OpenGL%202.0%20Spec%20%22.*

Alpha-to-coverage whitepaper–NVIDIA. Available at *http://download.developer.nvidia.com/developer/SDK/Individual_Samples/DEMOS/Direct3D9/src/AntiAliasingWithTransparency/docs/AntiAliasingWithTransparency.pdf.*