3

III

Alpha Blending as a Post-Process

Benjamin Hathaway

3.1 Introduction

In this article we will present a novel alpha-blending technique that was developed for the off-road racing game *Pure* (see Figure 3.1). *Pure* was released in the summer of 2008 for the Xbox360, PS3, and PC platforms and featured races that subjected the player to extreme elevations, revealing detailed vistas stretching out to a distance of over 30 kilometers. With the art direction set on a photo-realistic look and locations taken from around the globe—some would require a high degree of foliage cover to be at all believable, or even recognizable.

Figure 3.1. A typical scene from *Pure* (post tone mapping & bloom effects).

During development it became apparent that we were going to need alpha blending, and lots of it! Unfortunately, alpha blending is one aspect of computer graphics that is difficult to get right, and trade-offs between performance and visual quality are often made; indeed, reluctance to risk performance has led to some game titles avoiding the use of alpha blending altogether. For a thorough introduction to the issues posed by alpha blended rendering, the reader is referred to the [Thibieroz 08] paper on advanced rendering techniques and the seminal work [Porter and Duff 84].

3.2 The Alternatives

Pure was destined to run on several platforms, each being equipped with at least one Dx9 class GPU (supporting shader model 3). This immediately presented us with several (hardware assisted) options for rendering our foliage geometry.

Alpha blending. Alpha blending uses a scalar value output by the pixel shader (alpha-value) to blend a rendered fragment with the destination pixel data.

When rendering layers of foliage with alpha blending, z-buffering artifacts are common. This can largely be resolved if the rendered primitives are sorted to draw furthest from the camera first. Sorting primitives before rendering is usually a prohibitive CPU cost for game rendering, and in the case of intersecting primitives there may indeed be no single correct draw order.

Alpha testing. Alpha testing uses a binary value output by the pixel shader to determine if the output fragment is visible or not. Alpha testing is usually combined with z-buffering techniques, which can either negate the need for geometric depth sorting or provide fill-rate optimizations (by way of z-rejection) when the scene is sorted in a front-to-back order.

Alpha testing is one of the most commonly used solutions to date; however the technique is prone to aliasing at the alpha edges.

Alpha-to-coverage. Alpha-to-coverage converts the alpha value output by the pixel shader into a coverage mask. This coverage mask is combined with the standard multisample coverage mask to determine which samples should be updated.

When alpha-to-coverage is combined with alpha testing, softer edges can be rendered whilst maintaining all the technical benefits afforded by alpha test rendering, i.e., sort independence and z-rejection opportunities. Although this is an improvement on simple alpha testing, the resulting alpha gradients can be of a poor quality compared to those obtained in alpha blending. This is particularly true when using a low number of samples or on hardware that does not support flexible coverage masks.

3.3 The Source Artwork

To try and emulate the richness of natural foliage, each tree and bush was constructed from a multitude of polygonal planes. The planes were oriented as randomly as possible, thus increasing the perceived density of foliage.

As can be seen in Figure 3.2, this leads to a great deal of primitive intersection, which raises two issues:

1. How are we to correctly sort all the intersecting primitives?

2. How are we going to deal with the high degree of depth complexity present within a single foliage model?

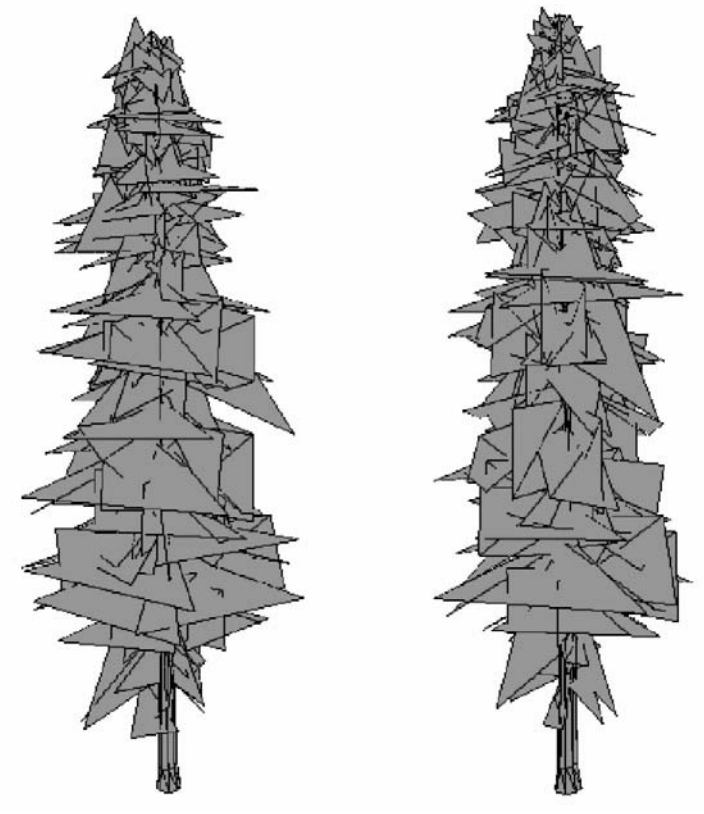

Figure 3.2. Geometric structure of a typical tree model rendered in *Pure* (shown from the front and side).

To correctly depth sort the primitives would require that we split the primitives along all the intersections. However, this would have increased the vertex count substantially and put a heavy burden on our memory budget, while also increasing memory bandwidth usage.

These issues are further exacerbated when the foliage is allowed to react to dynamic influences such as the wind, or physical collisions. Such influences may even cause neighboring foliage models to overlap rather unpredictably, and would therefore be impossible to optimize off-line, instead requiring a more costly, real-time solution.

3.4 Initial Attempts

Due to the high levels of primitive interpenetration within the artwork, we initially implemented our foliage renderer in the simplest manner possible, by using a combination of z-buffering and alpha-test techniques. After auditioning a number of variations of the alpha-reference value, we managed to achieve some reasonable results, although the overall look of the foliage tended to appear a little harsh, or synthetic, at times.

The most objectionable effect occurred when the camera performed slow translational motions from side-to-side (for example, during the pre-race camera sequences). As the camera moved, the alpha-channeled holes & edges within the foliage would begin to *sparkle* (due to the binary nature of alpha testing) and would often be exaggerated by the high degree of depth complexity present within each foliage model.

Next, we turned our attention towards the alpha-to-coverage feature. Alpha-to-coverage rendering integrates alpha-testing techniques with multi-sample rendering; it produces softer edges while maintaining all the technical benefits of alpha-test rendering. While initial results were favorable and the sparkling artifacts were indeed attenuated, we struggled to reproduce consistent results across all of our platforms. We also suffered from the increased fill-rate and bandwidth costs incurred by rendering the foliage at MSAA resolutions.

Neither approach seemed to deliver a high enough visual quality and it seemed a shame to quantize all those beautiful alpha-textures so heavily. Something new was needed—and we felt the answer was hiding in the silhouette of the foliage.

3.5 The Screen-Space Alpha Mask

The solution we devised—screen-space alpha masking (SSAM)—is a multi-pass approach (see the overview in Figure 3.3) implemented with rendering techniques that negate the need for any depth sorting or geometry splitting. Our solution can

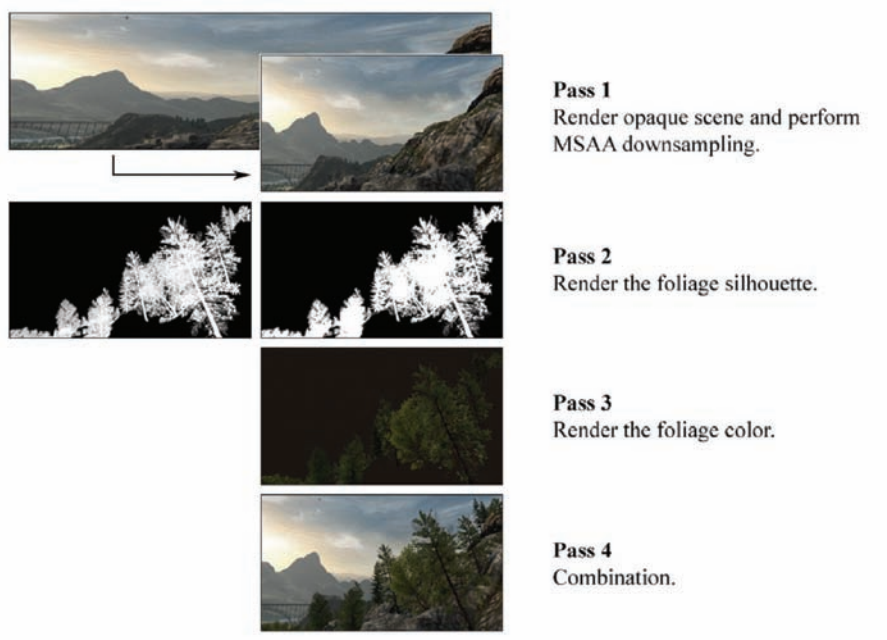

Figure 3.3. Diagram showing an overview of screen-space alpha masking.

yield results on par with alpha blending while correctly resolving internal overlaps (and depth intersections) on a per-pixel basis, using alpha-testing techniques.

We effectively performed *deferred alpha blending* using a full-screen post-process that resembles frame-buffer blending with the blend-operation set to ADD; the source and destination arguments are set to `SRCALPHA` and `INVSRCALPHA`, respectively. The inputs to the *blend* are rendered into three separate render targets and are then bound to texture samplers, referenced by the *final combination* post-process pixel shader (see Listing 3.2 in Section 3.11).

In terms of memory resources, we need at least three screen-resolution render-targets, two having at least three color channels (`rtOpaque` & `rtFoliage`), one with a minimum of two channels (`rtMask`), and a single depth buffer (`rtDepth`).

Note: this is *in addition* to any MSAA render-target memory requirements.

3.5.1 The Opaque Pass

During the first pass we rendered all our opaque scene elements into the color render target: `rtOpaque` (see Figure 3.4); the depth information generated was also kept and stored in the depth buffer: `rtDepth` (see Figure 3.5).

Figure 3.4. The opaque scene (color) written to `rtOpaque`.

For *Pure*, we rendered the opaque scene at 2× MSAA resolution and both the color and depth buffers were down sampled into screen-resolution render targets. Care had to be taken when down sampling the depth information, as incorrect samples were obtained when filtering was applied.

Figure 3.5. The opaque scene (depth) written to `rtDepth`.

Instead, we read a number of samples from the depth buffer (two, in the case of 2× MSAA), compared them, and simply retained the sample that was closest to the observer. For a more complete description of the solution adopted for *Pure*, the reader is referred to the work of [Iain Cantlay 04].

From this point onwards, we continued to render at non-MSAA resolutions, as it was observed that MSAA contributed little or nothing towards the quality of alpha generated edges—only those of a geometric nature. At this point, and depending on platform architecture, it may be necessary to copy the down sampled images back in to VRAM (repopulation) before further rendering can continue.

Additionally, at this point steps may need to be taken to update any hierarchical-z (hi-z) information that might be associated with `rtDepth`, and potentially optimize any subsequent draw calls that are depth occluded.

Note: Detailed information regarding the repopulation of depth information and restoration of Hi-Z can be obtained from the platform vendors and is unfortunately outside the scope of this article.

3.5.2 The Mask Generation Pass

In the second pass we generated a *silhouette*, or *mask*, of the foliage we wanted to render (see Figure 3.6). The silhouette image was rendered into our second render-target, `rtFoliage`, and we used the previously generated depth buffer, `rtDepth`, to correctly resolve any depth occlusions caused by the opaque scene.

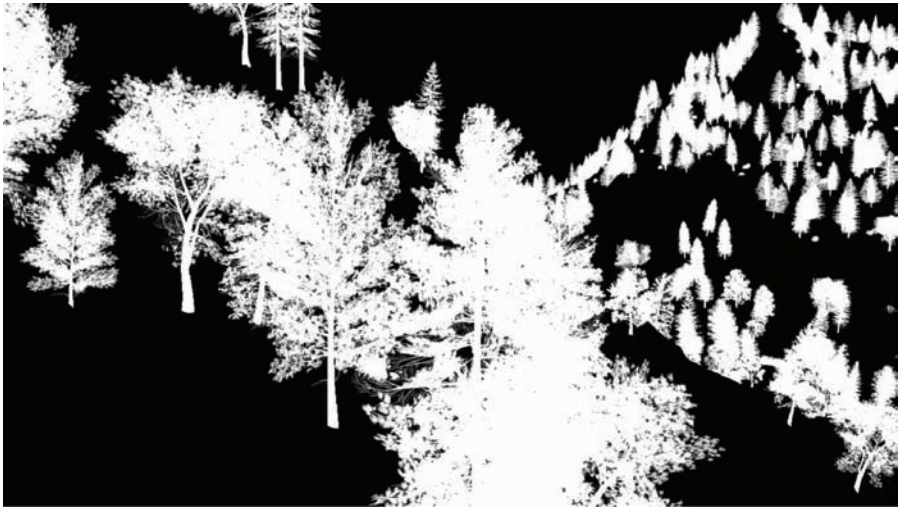

Figure 3.6. Additively accumulated foliage alpha mask.

The mask is a monochromatic image of the alpha-channel values that would normally be used during regular alpha blending. The alpha-values are additively blended onto a black background, during which, we enabled depth testing, and disabled both back-face culling and depth writes.

As the additive blending of two primitives produces the same result regardless of the draw order, it seemed to be the ideal option for efficient mask generation. However, some artifacts were evident when the mask was used: *as-is*, (as the blend factor) due to the limited bit depth of the render-targets being used, during the *final combination* pass, the additively accumulated values would quickly saturate towards white—even at a modest depth complexity. The saturation was most evident when foliage close to the observer was expected to possess low-order opacity, but is rendered in an area of high foliage depth complexity.

To generate as high quality a mask as possible, we needed to obtain as much detail from our silhouette as we could; we therefore set the alpha-reference render-state to zero during this pass, to avoid the rejection low opacity alpha-values.

A refinement on additive blending was the use of the max-blending mode. In this case we built up an image of the single highest alpha-value to have been written to each pixel, in effect acting as an *alpha-z-buffer*. As with additive blending, we set the source and destination blend arguments to `D3DBLEND_ONE`, but change the blend operation to `D3DBLENDOP_MAX`.

As can be seen in Figure 3.7, the max-blended mask contains a higher degree of structural information (than the additively blended mask) while still retaining the subtle alpha gradients located towards the outer edges of the foliage.

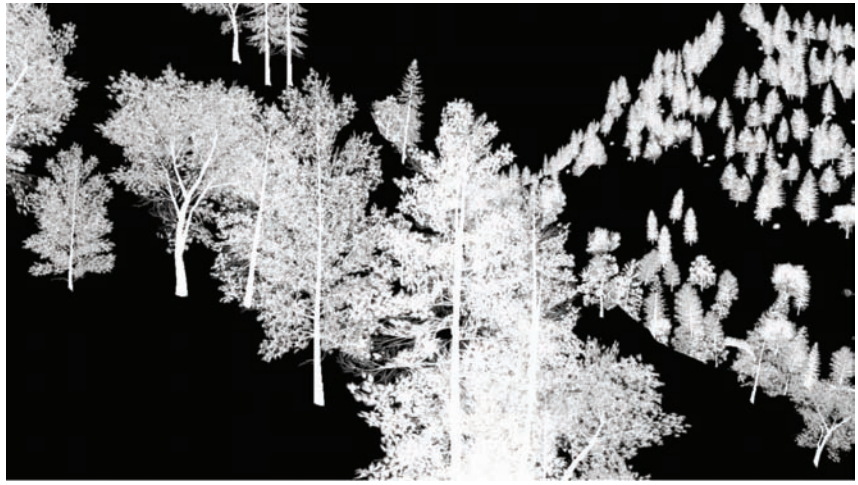

Figure 3.7. MAX blended foliage alpha mask.

Despite these attractive qualities, when the max-blended mask was used as the blend factor during the final combination pass, the foliage took on a wholly transparent look. We were left feeling that perhaps the solution lay somewhere in between the two approaches. And it did—quite literally.

The thought was that perhaps some linear combination of both masks would be the answer. Perhaps the saturation of the additive blending would compensate for the transparency of the max blending? And correspondingly, would the subtle alpha-gradients of the max-blended image reduce the saturation evident in the low-opacity, high-depth-complexity areas of the additive mask?

Fortunately the answer proved to be yes in both cases, and luckily for us all the platforms we were working with provided a method that could generate both of our masks in a single pass!

Separate alpha blend enable. Most DX9 class hardware supports the ability to specify separate blend operations and arguments for both the color *and* alpha channels, independently.

By enabling `D3DRS_SEPARATEALPHABLENDENABLE` and setting the correct series of operation and arguments, it is possible to simultaneously write the max-blended mask to the rgb channels (as a monochromatic image), and the additively-blended mask to the alpha channel (see Table 3.1).

Note: Both masks are combined at a later stage, prior to their use as the blend-factor in the final composition pass.

In order to send the alpha values to both blending units, we needed to replicate the alpha values across to all the color channels. This required a small modification to the end of the foliage shaders that resembled

```
Out.Color.xyzw = Out.Color.wwww;
```

Render State	**Value**
D3DRS_ALPHABLENDENABLE	TRUE
D3DRS_SEPERATEALPHABLENDENABLE	TRUE
D3DRS_BLENDOP	D3DBLENDOP_MAX
D3DRS_BLENDOPALPHA	D3DBLENDOP_ADD
D3DRS_SRCBLEND	D3DBLEND_ONE
D3DRS_SRCBLENDALPHA	D3DBLEND_ONE
D3DRS_DESTBLEND	D3DBLEND_ONE
D3DRS_DESTBLENDALPHA	D3DBLEND_ONE

Table 3.1. The blend-related render states used during mask generation.

The modification not only performed the replication, but also had the added benefit of optimizing the foliage-alpha rendering shader without us needing any prior knowledge of how the alpha value was generated. For example: to what sampler stage the alpha-texture-source was bound, etc.

All of the optimizing shader compilers we tested performed some form of *dead-code stripping*, see: ["Dead Code" 09]. This optimization removed any code that did not directly contribute to the output value, substantially increasing fill-rate efficiency, in this case, removing all of the color-related lighting equations and texture-fetch instructions that were not common to the generation of the alpha value.

HLSL source code for a typical mask rendering shader is provided in Listing 3.1 in Section 3.11.

3.5.3 The Color Pass

For the third rendering pass, we rendered an image of the foliage color into our final render-target: `rtFoliage` (see Figure 3.8), and again we use the depth buffer obtained during the opaque pass, stored in `rtDepth` (see Figure 3.9).

In order to maintain correct depth ordering (as is necessary in the case of the color image), we disabled both back-face culling and alpha blending, while enabling alpha test rendering, depth testing, and depth writes. Enabling depth writes during this pass also ensured that any subsequently rendered transparencies would be correctly depth sorted with the foliage.

When rendering the foliage color with alpha testing enabled, a suitable alpha reference value had to be chosen and we exposed the color-pass alpha reference value to the artists for tweaking.

The final value ended up being a trade-off between two values. First was a value high enough to produce visually pleasing alpha-test edges—for *Pure*, a value of ≈128. Second was a value low enough to minimize certain blending artifacts (that will be covered in Section 3.6), which for *Pure*, ended up being a value of ≈64.

3.5.4 The Final Composition Pass

In the fourth and final pass, we rendered a full screen post-process that essentially performed a linear interpolation of our opaque and foliage-color images, using the mask image as the blend-factor (see Figure 3.10).

The final composition blend equation resembled

```
finalColor = rtOpaque + (rtFoliage - rtOpaque) * rtMask;
```

Application of the post-process consisted of the rendering of an orthographically projected, quadrilateral polygon mapped over the entire screen onto which we

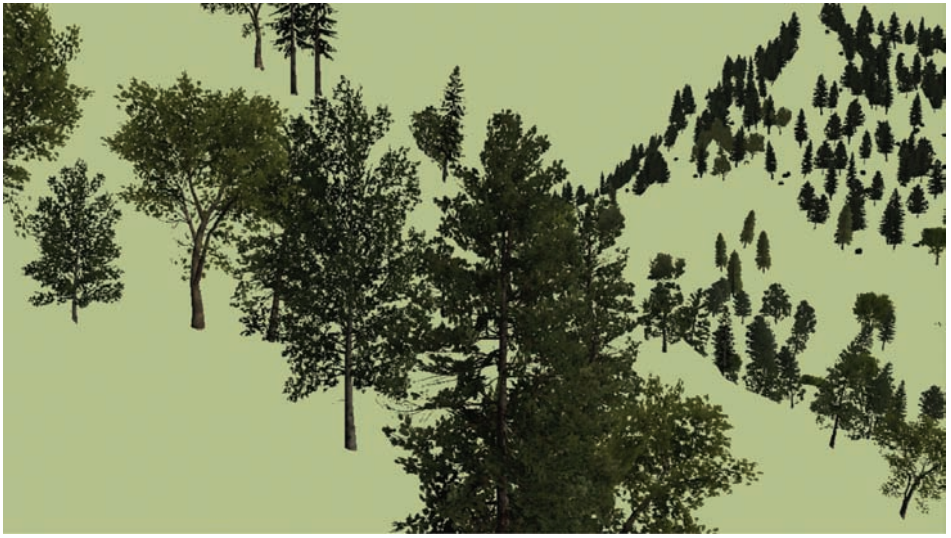

Figure 3.8. The foliage-color image written to `rtFoliage`.

Figure 3.9. The foliage-depth (alpha-tested) written into `rtDepth`.

Figure 3.10. The image produced by the final-combination pixel shader.

applied a pixel shader to actually perform the blending work. To ensure that we only sampled one texel per screen pixel, we took the platform's texture-sampling center into account and adjusted texture coordinates accordingly.

We now had two mask images to process, the max and additively blended masks, which needed to be combined in some way into a scalar blend-factor. For our implementation we chose to combine the masks using linear interpolation (or in HLSL, the *lerp* operation).

The final-composition blend equation, with linearly blended masks resembled

```
mask = rtMask.a + (rtMask.r - rtMask.a) * maskLerp;

rtResult = rtOpaque + (rtFoliage - rtOpaque) * mask;
```

The interpolation of the two masks introduced the value `maskLerp`, for which a value must be selected. Like the alpha-reference value, this is chosen on purely artistic grounds and was also exposed to the art team for experimentation. The final value for *Pure* was 0.85 (which produces a blend-factor composed of: 85% additive mask and 15% max-blended mask).

With `maskLerp` equal to 0.85, just enough max-blended mask is brought in to reduce the saturation artifacts without making the foliage too transparent.

In fact, it should be noted that some degree of transparency was found to be desirable. The slight transparency of the max contribution revealed distant

structure (such as trunks and branches) belonging to foliage that would have otherwise have been completely occluded by near foliage (adding a certain richness to the forest scenes).

The full HLSL source for the final composition pixel shader is given in Listing 3.2 in Section 3.11.

3.6 Alpha Reference Issues

As alpha test rendering was employed during the foliage color pass, an alpha-reference value was chosen—one that was high enough to stop overlapping edges from appearing too chunky (as mentioned, for *Pure* a value was chosen somewhere between ≈64 and ≈128). As a consequence, halo-like blending artifacts are sometimes visible where the foliage blended into the opaque image (see Figure 3.11).

3.6.1 The Clear-Screen Color Fix

Due to the alpha-reference value being set to a higher value during the color pass than that set during the mask-generation pass (for which the alpha-reference value was actually zero), moving outwards along an alpha gradient (from a value of one to zero), you can actually run out of foliage-color pixels before the mask-intensity reaches zero. This would reveal a proportion of the color-pass background color in

Figure 3.11. Image showing blending artifacts caused by a non-zero alpha reference value during the foliage-color pass.

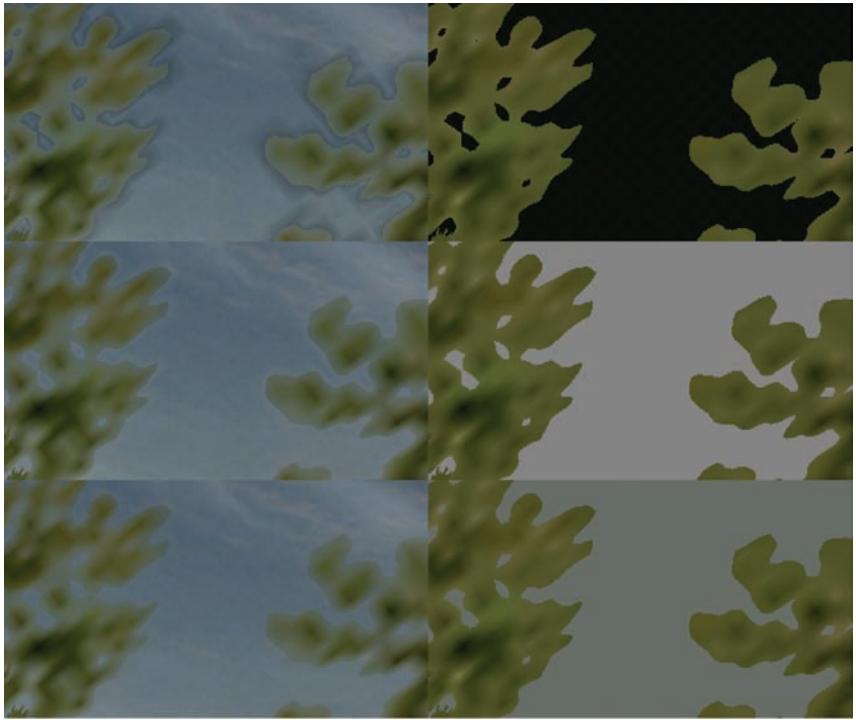

Figure 3.12. The effect of different clear screen colors (from top to bottom): too dark, too light, and just right.

pixels whose mask intensity fell below the color-pass alpha reference value. The solution employed for *Pure* was to expose the foliage color passes' clear screen color to the artists, the idea being that by adjusting the color, you could lighten the artifact until it was hardly visible (see Figure 3.12).

The technique worked well but felt less than optimum, especially as the artists could only choose one color per level. The color also tended to affect the overall color balance of the scene and would have to work for foliage rendered in both the lightest and darkest of conditions—very much a compromise.

3.6.2 The Squared Alpha Trick

A small modification made to the last line of the final composition pixel shader substantially improved the quality of the blending, almost entirely compensating for the aforementioned alpha-reference artifacts (see Figure 3.13). If the final

Figure 3.13. A close-up of the just-right clear screen color fix and the squared-alpha modification applied together.

mask value is numerically squared, the foliage alpha will roll off to black a little quicker while correctly maintaining areas of solid opacity.

```
return lerp(opaquePixel, foliagePixel, mask * mask);
```

It should be noted that while squaring the alpha channel does contract the foliage silhouette a little, a slight reduction in the foliage-color pass alpha-reference value should compensate for this.

3.7 Rendering Pipeline Integration

Foliage rendering is by no means the final step in rendering a game. There are many other alpha-blended elements to be integrated into the scene: grass, light shafts, and particle effects, to name but a few. Integration with these other stages is actually pretty straightforward, largely due to the fact that depth writing was enabled during the foliage-color pass.

This ensured that any subsequent depth testing would correctly resolve any depth-wise occlusions caused by the foliage (and/or opaque) scene elements.

3.8 Conclusion

In this article we have presented a novel (cross-platform) solution to the alpha blending of foliage, a solution that increases the quality of a wide range of alpha-test-class renderings, giving them the appearance of true alpha blending.

The use of SSAM within the game *Pure* had a profound effect on the overall perceived quality of the environments. The effect yielded a soft natural look without sacrificing any of the detail and contrast present in the source artwork. Below we list a few of the pros & cons to using SSAM:

Pros:

- Foliage edges are blended smoothly with the surrounding environment.
- Internally overlapping and interpenetrating primitives are sorted on a per-pixel basis using alpha testing techniques.
- The effect is implemented using simple, low-cost rendering techniques that do not require any geometric sorting or splitting (only consistency in primitive dispatch order is required).
- The final blending operations are performed at a linear cost (once per pixel) regardless of scene complexity and over-draw.
- The effect integrates well with other alpha-blending stages in the rendering pipeline (Particles, etc).
- When combined with other optimizations such as moving lighting to the vertex shader, and optimizing the shaders for each pass, overall performance can be higher than that of MSAA-based techniques.

Cons:

- The overhead of rendering the extra passes.
- Memory requirements are higher, as we need to store three images.
- The technique cannot be used to sort large collections of semi-transparent, glass-like surfaces (or soft alpha gradients that span large portions of the screen) without potentially exhibiting visual artifacts.[1]

3.9 Demo

A RenderMonkey scene, as well as several instructional .PSD files, are available at http://www.akpeters.com/gpupro.

[1] There are occasional opacity-related artifacts visible within overlapping alpha-gradients (when the alpha-foliage-mask is either: > 0 or, < 1). Fortunately, the foliage-color pass always yields the nearest, and therefore the most visually correct, surface color.

3.10 Acknowledgments

I would like to say a big thank you to everyone at Black Rock Studio who contributed to this article, particularly: Jeremy Moore and Tom Williams (for pushing me to write the article in the first place), and Damyan Pepper and James Callin for being there to bounce ideas off during the initial development.

An extra special thank you goes to Caroline Hathaway, Nicole Ancel, and Wessam Bahnassi for proofreading and editing the article.

3.11 Source Code

```
sampler2D foliageTexture : register(s0);

struct PS_INPUT
{
    half2 TexCoord : TEXCOORD0;
};

half4 main(PS_INPUT In) : COLOR
{
    return tex2D(foliageTexture, In.TexCoord).wwww;
}
```

Listing 3.1. HLSL source code for a typical mask rendering pixel shader.

```
sampler2D rtMask : register(s0);
sampler2D rtOpaque : register(s1);
sampler2D rtFoliage : register(s2);
half maskLerp : register(c0); // 0.85h

half4 main(float2 texCoord: TEXCOORD0) : COLOR
{
   half4 maskPixel = tex2D( rtMask, texCoord);
   half4 opaquePixel = tex2D( rtOpaque, texCoord);
   half4 foliagePixel = tex2D(rtFoliage, texCoord);
   half mask = lerp(maskPixel.x, maskPixel.w, maskLerp);

   return lerp(opaquePixel, foliagePixel, mask * mask);
}
```

Listing 3.2. HLSL source code for the final composition pixel shader.

Bibliography

["Dead Code" 09] "Dead Code Elimination." *Wikipedia*. Available at http://en.wikipedia.org/wiki/Dead_code_elimination, 2009.

[Iain Cantlay 04] Iain Cantlay. "High-Speed, Off-Screen Particles." In *GPU Gems 3*, edited by Hubert Nguyen. Reading, MA: Addison-Wesley Professional, 2007.

[Thibieroz 08] Nicolas Thibieroz. "Robust Order-Independent Transparency via Reverse Depth Peeling in DirectX10." In *ShaderX6: Advanced Rendering Techniques* (2008).

[Porter and Duff 84] Thomas Porter and Tom Duff. "Compositing Digital Images." *Computer Graphics* 18:3 (1984): 253–59.