Hashed Alpha Testing

Chris Wyman*
NVIDIA

Morgan McGuire
NVIDIA & Williams College

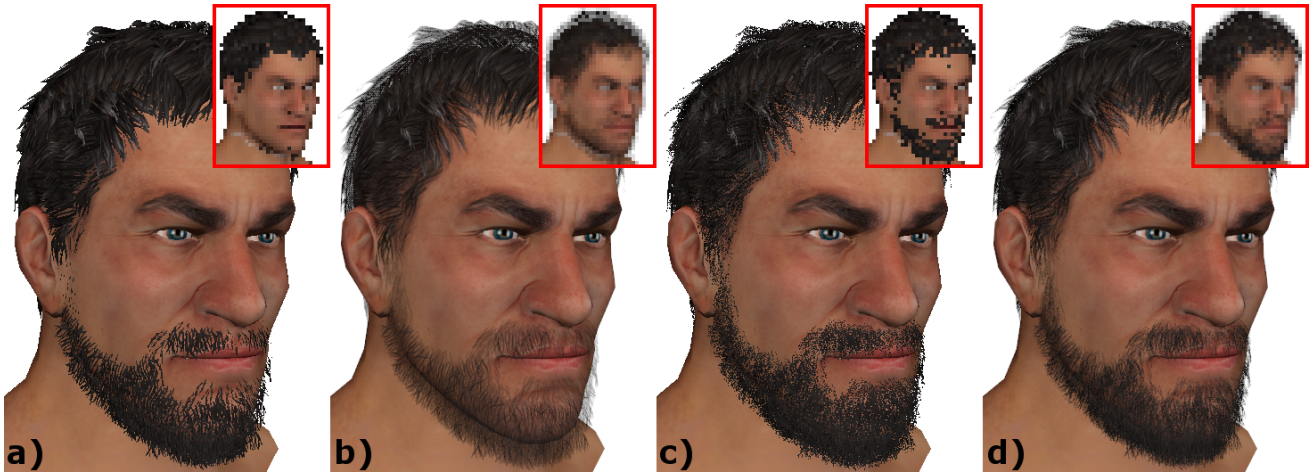

Figure 1: Hashed alpha testing *avoids alpha-mapped geometry disappearing in the distance and reduces correlations in multisample alpha-to-coverage. We show a bearded man with (a) traditional alpha testing, (b) traditional, hardware-accelerated alpha-to-coverage, (c) hashed alpha testing, and (d) hashed alpha-to-coverage. Insets show the same geometry from further away, enlarged to better depict variations.*

Abstract

Renderers apply *alpha testing* to mask out complex silhouettes using alpha textures on simple proxy geometry. While widely used, alpha testing has a long-standing problem that is underreported in the literature, but observable in commercial games: geometry can entirely disappear as alpha mapped polygons recede with distance. As foveated rendering for virtual reality spreads this problem worsens, as peripheral minification and prefiltering also cause this problem for *nearby* objects.

We introduce two algorithms, *stochastic alpha testing* and *hashed alpha testing*, that avoid this issue but add some noise. Instead of using a fixed alpha threshold, α_τ, stochastic alpha testing discards fragments with alpha below randomly chosen $\alpha_\tau \in (0..1]$. Hashed alpha testing uses a hash function to choose α_τ procedurally, producing stable noise that reduces temporal flicker.

With a good hash function and inputs, hashed alpha testing maintains distant geometry without introducing more temporal flicker than traditional alpha testing. We describe how hashed and stochastic alpha testing apply to alpha-to-coverage and screen-door transparency, and how they simplify stochastic transparency.

Keywords: hashed, stochastic, alpha map, alpha test
Concepts: •**Computing methodologies** → **Visibility;**

*e-mail:chris.wyman@acm.org

I3D '17, February 25-27, 2017, San Francisco, CA, USA
ISBN: 978-1-4503-4886-7/17/03...$15.00
DOI: http://dx.doi.org/10.1145/3023368.3023370

1 Introduction

For decades interactive renderers have used *alpha testing*, discarding fragments whose alpha falls below a specified threshold α_τ. While not suitable for transparent surfaces, which require alpha compositing [Porter and Duff 1984] and order-independent transparency [Wyman 2016a], alpha testing provides a cheap way to render binary visibility stored in an alpha map. Alpha testing is particularly common in engines using deferred rendering [Saito and Takahashi 1990] or complex post-processing, as it provides a correct and consistent depth buffer for subsequent passes. Today, games widely alpha test for foliage, fences, decals, and other small-scale details.

Alpha testing introduces artifacts. Its binary queries alias on alpha boundaries ($\alpha \approx \alpha_\tau$) occurring in texture space, making geometric antialiasing ineffective. Only postprocess antialiasing addresses this problem (e.g., [Lottes 2009; Karis 2014]). Texture prefiltering fails since a post-filter alpha test still gives binary results.

Less well-known, alpha mapped geometry can disappear in the distance, as shown in Figure 1. Largely ignored in academic contexts, game developers frequently encounter this problem (e.g., [Castano 2010]). Some scene-specific tuning and restrictions on content creation reduce the problem, but none completely solve it.

To solve this problem, we propose replacing the fixed alpha threshold, α_τ, with a stochastic threshold chosen uniformly in $(0..1]$, i.e., we replace:

if (color.a $<$ α_τ) **discard;**

with a stochastic test:

if (color.a $<$ *drand48*()) **discard;**

This adds high-frequency spatial and temporal noise, so we show a hashed alpha test has similar but stable behavior, i.e.,

if (color.a $<$ *hash*(. . .)) **discard;**

This paper makes the following contributions:

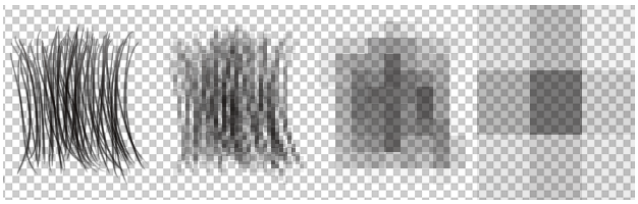

Figure 2: *Coarse mipmap texels average alpha over corresponding finer texels in the mipmap chain. The hair texture from Figure 1 has an average alpha $\alpha_{avg} = 0.32$, partly due to artist-added padding. Thus, accessing coarser mip levels for alpha thresholding, unsurprisingly, causes most fragments to fail the alpha test.*

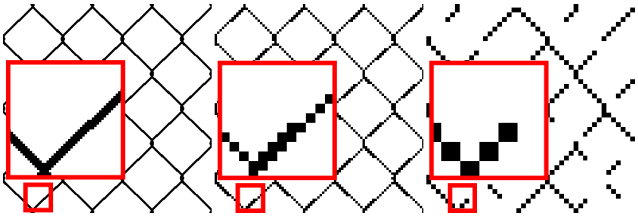

Figure 3: *When nearby, each metal wire in this fence is 5 pixels wide (left). With distance, alpha testing discretely erodes pixel coverage along edges so that the wire thins to roughly single-pixel width (center). Further away wires no longer cover even one pixel, and can disappear entirely (right).*

- shows *stochastic alpha testing* maintains consistent coverage even for distant geometry;
- shows *hashed alpha testing* provides similar benefits and, with well selected hash inputs, provides spatial and temporal stability comparable to traditional alpha testing;
- demonstrates how to integrate these algorithms into renderers without making magnified surfaces noisy, as traditional alpha testing already works acceptably there;
- shows stochastic alpha testing converges to stochastic transparency [Enderton et al. 2010] and order-independent transparency with enough stochastic samples of α_τ; and
- relates our new alpha tests to alpha-to-coverage and screen-door transparency, showing how those algorithms can be improved using stochasm and hashing.

2 Why Does Geometry Disappear?

Disappearing alpha-tested geometry is poorly covered in academic literature. However, game developers repeatedly encounter this issue. Castano [2010] investigates the causes and describes prior ad hoc solutions. Three issues contribute to loss of coverage in alpha-tested geometry:

Reduced alpha variance. Mipmap construction iteratively box filters the alpha channel, reducing alpha variance in coarser mipmap levels. At the coarsest level of detail (lod), the single texel averages alpha, α_{avg}, over the base texture. Many textures contain $\alpha_{avg} \ll \alpha_\tau$, causing more failed alpha tests in coarser lods, especially if the texture contains padding (see Figure 2). Essentially, a decreasing percentage of texels pass the alpha test in coarser mips.

Discrete visibility test. Unlike alpha blending, alpha testing gives binary visibility. Geometry is either visible or not in each pixel. As geometry approaches or recedes from the viewer, pixels suddenly transition between covered and not covered. This discretely erodes geometry with distance (see Figure 3). At some point 1-pixel wide geometry erodes to 0-pixel wide geometry, disappearing.

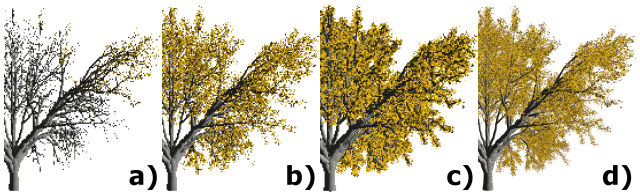

Figure 4: *Hashed alpha testing (b) maintains coverage better than regular alpha testing (a), but it still loses some coverage due to sub-pixel leaf billboards, compared to ground truth (d). Conservative raster (c) ensures full coverage, but alpha threshold α_τ must be modified based on sub-pixel coverage to avoid the oversampled billboards shown here.*

Coarse raster grid. With enough distance, even billboarded proxy geometry becomes sub-pixel. In this case, a relatively coarse rasterization grid poorly samples the geometry. Billboards may not appear on screen at all, causing an apparent loss of coverage (see Figure 4). Our algorithms do not address this issue, though conservative rasterization and multisampling reduce the problem.

3 State of the Art in Alpha Testing

Game developers have dealt with disappearing alpha-mapped geometry for years, as games often display foliage, fences, and hair from afar. Various techniques can help manage the problem.

Adjusting α_τ per texture mipmap level. Since mipmapping filters alpha, adjusting α_τ per mip-level can correct for reduced variance. Consider a screen-aligned, alpha-tested billboard covering A_0 pixels where a_0 pixels pass the alpha test at mip level 0. Seen at a distance, this screen-aligned billboard might use mip level i. You expect a consistent percentage of pixels passing the alpha test, i.e.:

$$a_0/A_0 \approx a_i/A_i.$$

One can precompute test thresholds $\alpha_\tau(i)$ that maintain this ratio for each mipmap [Castano 2010]. But as content affects this threshold, it differs between textures and even within a mip level (i.e., you want $\alpha_\tau(i, u, v)$). Even perfect per-level thresholds do not prevent geometry disappearing with distance; the fence in Figure 3 has nearly constant $\alpha_\tau(i)$ but still quickly disappears with distance.

Always sampling α from finest mip lod. Because filtering reduces alpha variance and changes threshold $\alpha_\tau(i)$, always sampling α from mip level 0 trivially avoids the problem. But this either costs two texel fetches per pixel (one for RGB, one for α) or eliminates color prefiltering. An alternative limits the maximum mipmap lod to a manually specified $i_{\max}$ and uses level $i_{\max}$ when $i > i_{\max}$. But both approaches can thrash the texture cache and reduce the temporal stability of fine texture details.

Rendering first with α-test, and then with α-blend. Alpha testing's popularity stems from order-independent blending's difficulty. Naive blending causes halos if the z-buffer gets polluted by transparent fragments. By first rendering with alpha testing and then rerendering with alpha blending, z-buffer pollution is reduced [Moore and Jefferies 2009]. This guarantees transparent fragments never occlude opaque ones, but requires rendering alpha-mapped geometry twice and does not work with deferred shading.

Supersampling. When storing binary visibility, a base texture's alpha channel contains only 0s or 1s. With sufficiently dense supersampling, you can always access the full resolution texture, providing accurate visibility. But the required density can be arbitrarily high, due to geometric transformations and parameterizations.

Alpha-to-coverage. With n-sample multisampling, alpha is discretized and outputs $\lfloor n\alpha \rfloor$ dithered binary coverage samples [Khar-

Figure 5: *A minified cedar tree using (a) alpha testing, stochastic alpha testing with (b) 1, (c) 4, (d) 16, and (e) 64 samples per pixel, and (f) a supersampled ground truth using sorted alpha blending.*

lamov et al. 2008]. Usually these dither patterns are fixed, causing correlations between layers and preventing correct multi-layer compositing. Enderton et al. [2010] proposes selecting random pattern permutations to address this problem.

Screen-door transparency. While uncommon today, screen-door transparency behaves similar to alpha-to-coverage except dithering occurs over multiple pixels. Various mask selection techniques exist for screen-door transparency [Mulder et al. 1998], but even random mask patterns lead to correlation between composited layers and repeating dither patterns visible over the screen.

4 Stochastic Alpha Testing

Stochastic alpha testing's key idea is simple: replace the fixed alpha threshold ($\alpha_\tau = 0.5$) with a stochastic threshold ($\alpha_\tau = drand48()$). Essentially, this replaces one sample from a regular grid pattern (i.e., sampling $[0..1]$ at 0.5) with one uniform random sample.

This simplifies stochastic transparency [Enderton et al. 2010] to use one sample per pixel. While this seems trivial, we observe that replacing a fixed alpha threshold with a random one solves the disappearing coverage problem: alpha mapped surfaces no longer disappear with distance (see Figure 5). Unlike with a fixed alpha threshold, with stochastic sampling the visibility, $V = (\alpha < \alpha_\tau)\ ?\ 0 : 1$, has the correct expected value $\mathbf{E}[V]$ of α.

But a single random sample is insufficient, as it introduces significant noise that causes continuous twinkle (see video). Reusing random seeds between frames and using stratification, e.g., [Laine and Karras 2011; Wyman 2016b], help stabilize noise for static geometry. But whenever object or camera moves the high-frequency noise reappears. Supersampling helps, but using one sample per pixel is a major appeal of alpha testing, since it works in forward and deferred shading, without MSAA, and even on low-end hardware.

5 Hashed Alpha Testing

In *hashed alpha testing* we aim for quality equivalent to stochastic alpha testing while simultaneously achieving stability equivalent to traditional alpha testing.

Instead of stochastic sampling, we propose using a hash function to generate alpha thresholds. Appropriate hash functions include those that generate outputs uniformly distributed in $[0..1)$, allowing direct substitution for uniform random number generators while allowing finer control by adjusting their inputs.

To obtain well distributed noise with spatial and temporal stability, we sought the following hash properties:

- noise anchored to surface, to avoid appearance of swimming;
- no correlations between overlapping alpha-mapped layers;
- and output α_τ discretized at roughly pixel scale, so sub-pixel translations return the same hashed value.

5.1 Hash Function

Our hash function is less important than its properties. We use the following hash function $f : \mathbb{R}^2 \to [0..1)$ from McGuire [2016]:

```
float hash( vec2 in ) {
   return fract( 1.0e4 * sin( 17.0*in.x + 0.1*in.y ) *
                 ( 0.1 + abs( sin( 13.0*in.y + in.x ))))
               );
}
```

We tried other hash functions, which gave largely equivalent results after (potentially) scaling inputs by different multipliers to obtain similar frequencies in the output.

To hash 3D coordinates, a hash $f : \mathbb{R}^3 \to [0..1)$ may provide more control. Repeatedly applying our 2D hash worked well for us:

```
float hash3D( vec3 in ) {
   return hash( vec2( hash( in.xy ), in.z ) );
}
```

5.2 Anchoring Hashed Noise to Geometry

To avoid noise swimming over surfaces, `hash()` inputs must stay fixed under camera and object motion. Candidates for stable inputs include those based on texture, world-space, and object-space coordinates.

In scenes with a unique texture parameterization, texture coordinates work well. But many scenes lack such parameterizations.

Hashing world-space coordinates provides stable noise for static geometry, and our early tests used world-space coordinates. However, this fails on dynamic geometry. Object-space coordinates give stable hashes for skinned and rigid transforms and dynamic cameras.

All our stability improvements disappear if coordinate frames become sub-pixel, as each pixel then uses different coordinates to compute its hashed threshold. So it is vital to use coordinates consistent over an entire aggregate surface (e.g., a tree) rather than a portion of the object (e.g., each leaf).

5.3 Avoiding Correlations Between Layers

For overlapping alpha-mapped surfaces, using similar α_τ thresholds between layers introduces undesirable correlations similar to those observed in hardware alpha-to-coverage. These correlations are most noticeable when hashing window or eye-space coordinates, but they can also arise for texture-space inputs.

Including the z-coordinate in the hash trivially removes these correlations. We recommend always hashing with 3D coordinates to avoid potential correlations.

5.4 Achieving Stable Pixel-Scale Noise

Under slow movement, we do not want new thresholds α_τ between frames, as that causes severe temporal noise. But we still expect α_τ to vary between pixels, allowing dithering of opacity over adjacent pixels. This suggests using pixel-scale noise and thus reusing α_τ for sub-pixel movements. Since noise is anchored to the surface, for larger motions α_τ will *also* be reused, just in different pixels.

5.4.1 Stability Under Screen-Space Translations in X and Y

For stable pixel-scale noise, we normalize our object-space coordinates by their screen-space derivatives and then clamp. This causes all values on a pixel-scale to generate the same hashed value:

```
// Find the derivatives of our coordinates
float pixDeriv = max( length(dFdx(objCoord.xyz)),
                      length(dFdy(objCoord.xyz)) );

// Scale of noise in pixels (w/ user param g_HashScale)
float pixScale = 1.0/(g_HashScale*pixDeriv);

// Compute our alpha threshold
float α_τ = hash3D( floor(pixScale*objCoord.xyz) );
```

Here, `pixScale` scales `objCoord` so that we discretize our hash input (via the `floor()`) at roughly pixel scale, allowing all inputs within a pixel-sized region to return the same hashed value. User parameter `g_HashScale` controls the target noise scale in pixels (default 1.0). Changing `g_HashScale` is useful if the chosen hash outputs noise at another frequency. Also, when temporal antialiasing, using noise below pixel scale (e.g., 0.3–0.5) allows for temporal averaging.

5.4.2 Stability Under Screen-Space Translations in Z

That approach gives stable noise under small vertical and horizontal translations. But moving along the camera's z-axis changes derivatives `dFdx()` and `dFdy()`, thus changing hash inputs continuously. This gives noisy results, comparable to stochastic alpha testing, as α_τ thresholds are effectively randomized by the hash each frame.

For stability under z-translations, we need to discretize changes induced by such motion. In this case, only `pixDeriv` changes, so discretizing it adds the needed stability:

```
// To discretize noise under z-translations:
float pixDeriv = floor( max(length(dFdx(objCoord.xyz)),
                            length(dFdy(objCoord.xyz))) );
```

But this still exhibits discontinuities if `pixDeriv` simultaneously changes between discrete values in many pixels, e.g., when drawing large, view-aligned billboards. Ideally, we would change our noise slowly and continuously by interpolating between hashes based on two discrete values of `pixDeriv`, as below:

```
// Find the discretized derivatives of our coordinates
float maxDeriv = max( length(dFdx(objCoord.xyz)),
                      length(dFdy(objCoord.xyz)) );
vec2 pixDeriv = vec2( floor(maxDeriv), ceil(maxDeriv) );

// Two closest noise scales
vec2 pixScales = vec2( 1.0/(g_HashScale*pixDeriv.x),
                       1.0/(g_HashScale*pixDeriv.y) );

// Compute alpha thresholds at our two noise scales
vec2 alpha = vec2(hash3D(floor(pixScales.x*objCoord.xyz)),
                  hash3D(floor(pixScales.y*objCoord.xyz)));

// Factor to interpolate lerp with
float lerpFactor = fract( maxDeriv );

// Interpolate alpha threshold from noise at two scales
float α_τ = (1-lerpFacor)*alpha.x + lerpFactor*alpha.y;
```

This *almost* achieves our goal, but has two problems. First, it fails for $0 \leq$ `maxDeriv` < 1. To solve this we discretize `pixScale` on a logarithmic scale instead of discretizing `pixDeriv` on a linear scale:

```
// Scale of noise in pixels (w/ user param g_HashScale)
float pixScale = 1.0/(g_HashScale*maxDeriv);

// Discretize pixScales on a logarithmic scale
vec2 pixScales = vec2( exp2(floor(log2(pixScale))),
                       exp2(ceil(log2(pixScale))) );

// Factor to interpolate lerp with
float lerpFactor = fract( log2(pixScale) );
```

```
// Find the discretized derivatives of our coordinates
float maxDeriv = max( length(dFdx(objCoord.xyz)),
                      length(dFdy(objCoord.xyz)) );
float pixScale = 1.0/(g_HashScale*maxDeriv);

// Find two nearest log-discretized noise scales
vec2 pixScales = vec2( exp2(floor(log2(pixScale))),
                       exp2(ceil(log2(pixScale))) );

// Compute alpha thresholds at our two noise scales
vec2 alpha=vec2(hash3D(floor(pixScales.x*objCoord.xyz)),
                hash3D(floor(pixScales.y*objCoord.xyz)));

// Factor to interpolate lerp with
float lerpFactor = fract( log2(pixScale) );

// Interpolate alpha threshold from noise at two scales
float x = (1-lerpFactor)*alpha.x + lerpFactor*alpha.y;

// Pass into CDF to compute uniformly distrib threshold
float a = min( lerpFactor, 1-lerpFactor );
vec3 cases = vec3( x*x/(2*a*(1-a)),
                   (x-0.5*a)/(1-a),
                   1.0-((1-x)*(1-x)/(2*a*(1-a))) );

// Find our final, uniformly distributed alpha threshold
float α_τ = (x < (1-a)) ?
                ((x < a) ? cases.x : cases.y) :
                cases.z;

// Avoids α_τ == 0.  Could also do α_τ=1-α_τ
α_τ = clamp( α_τ, 1.0e-6, 1.0 );
```

Listing 1: *Final code to compute our hashed alpha threshold.*

A trickier problem arises during interpolation. A well-designed hash function $f : \mathbb{R}^2 \rightarrow [0..1)$ produces uniformly distributed output values in $[0..1)$. Interpolating between two uniformly distributed values in $[0..1)$ does *not* yield a new uniformly distributed value in $[0..1)$. This introduces strobing because the variance of our hashed noise changes during motion.

Fortunately, we can transform our output back into an uniform distribution by computing the cumulative distribution function (of two interpolated uniform random values) and substituting in our interpolated threshold. The cumulative distribution function is:

$$\mathrm{cdf}(x) = \begin{cases} \frac{x^2}{2a(1-a)} & 0 \leq x < a \\ \frac{x-a/2}{1-a} & a \leq x < 1-a \\ 1 - \frac{(1-x)^2}{2a(1-a)} & 1-a \leq x < 1 \end{cases} \quad (1)$$

for $a =$ `min(lerpFactor, 1-lerpFactor)`.

Combining these improvements gives the final computation for α_τ shown in Listing 1.

5.5 Implementation Considerations

When adding hashed alpha testing into a renderer, the goal is likely avoiding fadeout of distant alpha-mapped geometry. Traditional alpha testing works fine for nearby geometry, and having even stable noise along nearby edges may be undesirable.

5.5.1 Fading in Noise with Distance

Fortunately, we can fade in hashed noise by considering the following formulation of our alpha threshold:

$$\alpha_\tau = 0.5 + \delta,$$

where $\delta = 0$ for traditional alpha testing and $\delta \in (-0.5...0.5]$ for hashed and stochastic variants. We suggest modifying this as:

$$\alpha_\tau = 0.5 + \delta \cdot b(\text{lod}), \tag{2}$$

where $b(\text{lod})$ slowly blends in the noise, i.e., $b(0) = 0$ and $b(n) = 1$ for $\text{lod} = n$ coarse enough to rely entirely on hashed alpha tests. Values for n depends on your texture sizes and tolerance for noise; we found $n = 6$ worked well in all our experiments.

We found that linearly ramping b still introduced noise too close to the camera. The following quadratic ramp gave a good transition:

$$b(x) = \begin{cases} 0 & x \leq 0 \\ (x/n)^2 & 0 < x < n \\ 1 & x \geq n. \end{cases} \tag{3}$$

Anisotropic filtering repeatedly accesses finer mip levels, causing alpha geometry to fade out even using relatively low mip levels. Scaling x based on anisotropy before computing $b(x)$ fixes this:

```
// Find degree of anisotropy from texture coords
vec2 dTex = vec2( length(dFdx(texCoord.xy)),
                  length(dFdy(texCoord.xy)) );
float aniso = max( dTex.x/dTex.y, dTex.y/dTex.x );

// Modify inputs to b(x) based on degree of aniso
x = aniso * x;
```

Higher anisotropy increases x, varying α_τ more in Equation 2, and fixes these disappearing alpha maps at grazing angles.

5.5.2 Using Premultiplied Alpha

As hashed alpha testing accesses diffuse texture samples from a somewhat larger region than standard alpha tests, care is required to avoid sampling in-painted diffuse colors added by artists in transparent regions, especially at higher mip levels when we filter from large regions of the texture.

We encourage using premultiplied alpha, which mipmaps correctly and avoids this problem (e.g., see Glassner's [2015] work for further discussion). However, since premultipled alpha texture stores $(\alpha R, \alpha G, \alpha B, \alpha)$, we need to divide by alpha before returning our alpha tested color (R, G, B) in order to maintain convergence to ground truth when increasing sample counts.

While hashed alpha testing works with non-premultiplied diffuse textures, we frequently found that when hashed $\alpha_\tau < 0.5$, colors bled from transparent texels and introduced arbitrarily colored halos at alpha boundaries.

6 Applications to Alpha-to-Coverage

As noted in Section 3, alpha-to-coverage discretizes fragment alpha and outputs $\lfloor n\alpha \rfloor$ coverage bits dithered over an n-sample buffer. Generating $\lfloor n\alpha \rfloor$ coverage bits is equivalent to supersampling the alpha threshold, i.e., performing n alpha tests with thresholds:

$$\alpha_\tau = \frac{0.5}{n}, \frac{1.5}{n}, ..., \frac{n - 0.5}{n}. \tag{4}$$

This observation reveals that traditional alpha testing is a special case, where $n = 1$.

Applying hashed or stochastic alpha testing to alpha-to-coverage is equivalent to jittered sampling of the thresholds:

$$\alpha_\tau = \frac{\chi_1}{n}, \frac{1 + \chi_2}{n}, ..., \frac{n - 1 + \chi_n}{n}, \tag{5}$$

Scene	# tris	Traditional Alpha Test	Hashed Alpha Test	Stochastic Alpha Test
Single fence	2	0.06 ms	0.08 ms	0.20 ms
Bearded Man	7.6 k	0.14 ms	0.16 ms	0.58 ms
Potted Palm	68 k	0.10 ms	0.12 ms	0.42 ms
Bishop Pine	158 k	0.22 ms	0.30 ms	0.75 ms
Japanese Walnut	227 k	0.28 ms	0.36 ms	0.99 ms
European Beech	386 k	0.39 ms	0.50 ms	1.69 ms
Sponza with Trees	900 k	1.05 ms	1.15 ms	4.04 ms
QG Tree	2,400 k	1.08 ms	1.22 ms	3.01 ms
UE3 FoliageMap	3,000 k	2.52 ms	2.86 ms	11.42 ms
San Miguel	10,500 k	5.19 ms	5.28 ms	7.30 ms

Table 1: *Cost comparisons for traditional, hashed, and stochastic alpha tests at* 1920×1080 *on a GeForce GTX 1080.*

for hashed samples $\chi_i \in (0..1]$. χ_i can be chosen independently per sample or once per fragment (i.e., $\chi_i = \chi_j$).

Traditional alpha-to-coverage uses fixed dither patterns for all fragments with the same alpha. This introduces correlations between overlapping transparent fragments, causing aggregate geometry to lose opacity (see Figure 1).

To avoid this, we compute a per-fragment offset α_o, increment α_+, and apply per-sample alpha thresholds:

$$\alpha_\tau = \frac{\chi_i + ((\alpha_o + i\alpha_+) \bmod n)}{n}, \quad \forall\, i \in [0...n-1]$$

Given hashed $\xi_1, \xi_2 \in [0..1)$ and limiting n to powers of two, $\alpha_o = \lfloor n\xi_1 \rfloor$ is an integer between 0 and $n-1$ and $\alpha_+ = 2\lfloor 0.5\, n\xi_2 \rfloor + 1$ is an odd integer between 1 and $n-1$. This decorrelates the jittered thresholds if ξ_1 or ξ_2 varies with distance to the camera.

6.1 Applications to Screen Door Transparency

Screen-door transparency simply dithers coverage bits over multiple pixels rather than multiple sub-pixel samples. So similar randomization of the interleaved α_τ thresholds and decorrelation between layers can occur between pixels rather than within a pixel.

7 Comparison to Stochastic Transparency

Stochastic alpha testing essentially simplifies stochastic transparency to one sample per pixel. Hence, when using more tests per pixel stochastic alpha testing converges to ground truth, just as stochastic transparency does.

In this light, based on the thresholds in Equation 4, alpha-to-coverage is stochastic transparency with regular instead of random samples. Using randomized thresholds, as in Equation 5, corresponds to stratified stochastic transparency.

But a key difference is that alpha testing is designed and frequently *expected* to work with a single sample per pixel. Avoiding temporal and spatial noise is key for adoption, hence our stable hashed alpha testing, which we believe provides an appealing alternative to traditional alpha testing.

8 Results

We prototyped our hashed and stochastic alpha test in an OpenGL-based renderer using the Falcor prototyping library [Benty 2016]. We did not optimize performance, particularly for stochastic alpha testing, as we sought stable noise rather than optimal performance. Timings include logic to explore variations to hashes, fade-in functions, and other normalization factors.

Figure 6: *Four plant models whose alpha-mapped polygons disappear with distance. This also happens when rendering at lower resolution (left four columns), which allows for better comparisons to our supersampled ground truth (right column). Notice how alpha testing loses alpha-mapped details and alpha-to-coverage introduces correlations that under represent final opacity where transparent polygons overlap. Both hashed alpha testing and hashed alpha-to-coverage largely retain appropriate coverage, but both introduce some noise.*

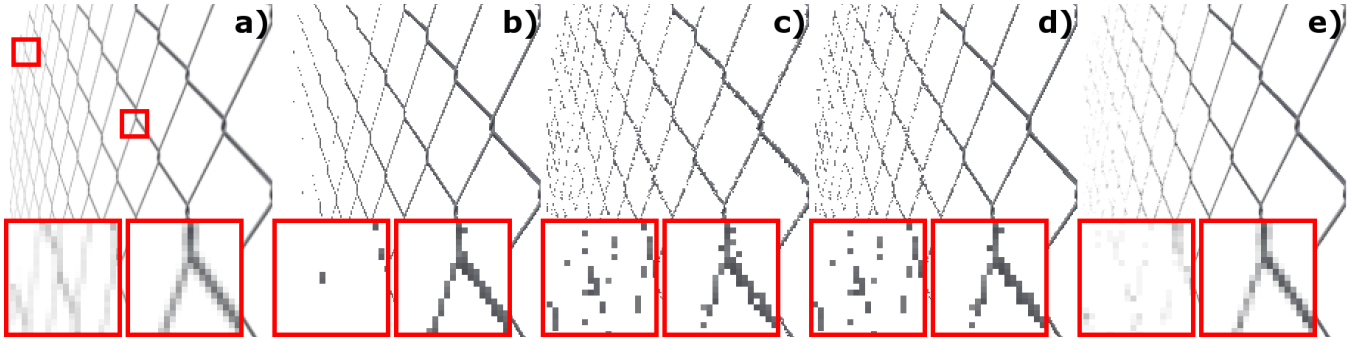

Figure 7: *An alpha-mapped chain link fence, as seen from an oblique angle using (a) alpha blending, (b) traditional alpha testing, (c) hashed alpha testing, (d) hashed alpha with noise faded in as per Section 5.5.1, and (e) hashed alpha testing with temporal antialiasing. Note how hashed alpha testing shows (noisy) geometry much further in the distance and better maintains the overall opacity (visible in the temporally antialised image). With the hash fade in, nearby chains appear much less noisy.*

Table 1 shows our performance relative to traditional alpha testing, using one shader for all surfaces, transparent and opaque, and rendered at 1920×1080. Our added overhead for hashed alpha testing is all computation, without additional texture or global memory accesses. Our stochastic alpha test prototype uses a random seed texture, requiring synchronization to avoid correlations from seed reuse. This causes a significant slowdown.

Cost varies with number, depth complexity, and screen coverage of alpha-mapped surfaces. At 1920×1080 with one test per fragment, our hashed alpha test costs an additional 0.1 to 0.3 ms per frame for scenes with typical numbers of alpha-mapped fragments. For stochastic alpha testing, synchronization costs increase greatly in high depth complexity scenes.

Figure 1 shows a game-quality head model with alpha-mapped hair billboards. With distance the hair disappears. This is most visible in his beard, as the underlying diffuse texture has no hair painted on his chin. See the supplemental video for dynamic comparisons with this model.

Figure 6 shows similar comparisons on a number of artist-created tree models provided as samples by XFrog. These trees' alpha maps contain 50–75% transparent pixels, causing foliage to disappear quickly when rendered in the distance or at low resolution. Hardware accelerated alpha-to-coverage has high layer-to-layer correlation that causes leaves to appear as a single layer and overly transparent. Hashed alpha testing and hashed alpha-to-coverage fix these problems and appear much closer to the ground truth, despite rendering at lower resolution.

Figure 7 demonstrates the behavior of hashed alpha testing under anisotropic filtering. With traditional alpha testing, the chain link fence quickly disappears while hashed alpha testing still shows details in the distance. However, even nearby chains on the fence have noisy edges. The level-of-detail based transition from Section 5.5.1 maintains the noisy details in the distance while switching to alpha testing nearer the camera. See our video for behavior under motion.

Figure 8 shows a more complex example where hashed noise may be undesirable nearby and the fade-in from Section 5.5.1 maintains crisp edges near the viewer.

Figure 9 compares the temporal stability of traditional, hashed, and stochastic alpha testing under slight, sub-pixel motion. Note that under the same sub-pixel motion hashed alpha testing exhibits temporal stability roughly equivalent to traditional alpha testing. Stochastic alpha testing and methods that do not anchor noise or discretize it to pixel scale exhibit significantly more instability.

Beyond use for distant or low-resolution alpha-mapped geometry,

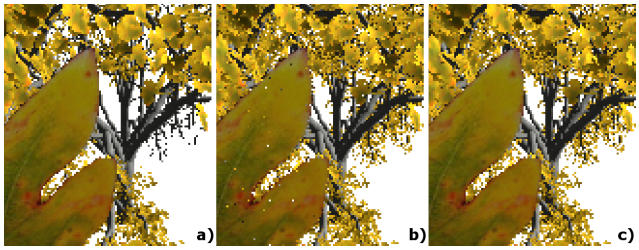

Figure 8: *With alpha testing (a), leaves in the tree disappear with distance. With hashed alpha testing these leaves are visible (b), but nearby leaves have noisy edges and, due to leaf alpha of 0.99, some internal noise. Fading in the hash contribution (c), as per Section 5.5.1, keeps distant leaves without nearby noise.*

other applications exist for hashed alpha testing. In head-mounted displays for virtual reality, rendering at full resolution in the user's periphery is computationally wasteful, especially as display resolutions increase. Instead, foveated rendering [Guenter et al. 2012] renders at lower resolution away from a user's gaze. Patney et al. [2016] suggest prefiltering all rendering terms, but they were not able to support alpha testing due to an inability to prefilter the results. Naive alpha testing in foveated rendering causes even nearby foliage to disappear in the periphery (see Figure 10). With temporal antialiasing, hashed alpha testing enables use of alpha mapped geometry in foveated renderers.

Figures 11 shows hashed alpha testing in a more complex environment. Results are more subtle as scene scale is small enough to minimize the pixels accessing very coarse mipmaps.

9 Conclusions

We introduced two new algorithms to solve the problem of alpha-mapped geometry disappearing with distance. *Stochastic alpha testing* uses a randomly chosen alpha test threshold α_τ rather than a fixed threshold. Because this introduces objectionable temporal noise, we developed *hashed alpha testing* to provide stable, procedurally generated noise using a hash function.

We obtained stable, pixel scale noise by hashing on discretized object-space coordinates at two scales. We showed how to ensure the interpolated hash value maintained a uniform distribution, and demonstrated temporal stability both in Figure 9 and the accompanying video. We provided inline code to replicate our hashed test.

Thinking about alpha-to-coverage and screen door transparency in

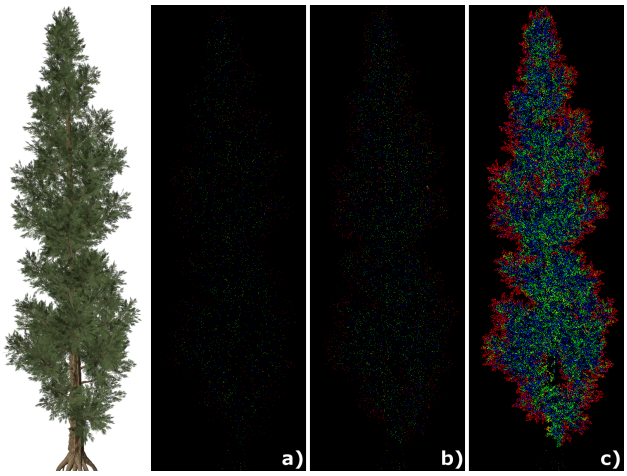

Figure 9: *We rendered a tree before and after a sub-pixel translation along the x-axis, and computed difference images for (a) traditional alpha testing, (b) hashed alpha testing, and (c) stochastic alpha testing. Red pixels correspond to major changes (with RGB differences between 0.5 and 1), usually where pixels toggle between background and pine needle. Yellow pixels represent moderate differences (0.25 to 0.5), green pixels smaller changes (0.13 to 0.25), and blue pixels minor changes (0.06 to 0.13).*

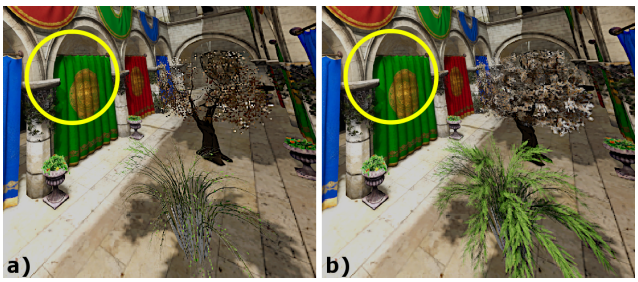

Figure 10: *Two images from Patney et al.'s [2016] foveated renderer. In both images, the viewer is gazing towards the green curtain (noted by the yellow circle) and regions outside the circle are rendered at progressively coarser resolution. Low resolution shading uses coarser mipmap levels, causing most alpha tests to fail using (a) traditional alpha testing. With (b) hashed alpha testing, the foliage maintains its aggregate appearance.*

the context of varying α_τ provides insights, showing them all to be different discrete sampling strategies for transparency: alpha test and alpha-to-coverage perform regular sampling, screen-door transparency interleaves samples, stochastic alpha testing randomly samples, and hashed alpha testing uses quasi-random sampling via a uniform hash function.

While our hashed test provides spatially and temporally stable noise without scene-dependent parameters, we did not explore the space of 2D and 3D hash functions to see which minimizes flicker between frames. Additionally, hash inputs more sophisticated than object-space coordinates may generalize over a larger variety of highly instanced scenes. Both areas seem fruitful for future work.

10 Acknowledgments

This idea evolved out of a larger project, meaning many researchers contributed indirectly. Special thanks to Pete Shirley for keeping our hash uniform by deriving the cdf in Equation 1, as well as Cyril Crassin for suggesting alpha maps as a simplified domain worth studying, Anton Kaplanyan for discussions on stable noise, Anjul Patney for adding hashed alpha testing to his foveated renderer, and Dave Luebke, Aaron Lefohn, and Marco Salvi for discussions on larger research directions.

Figure 11: *Disappearing geometry in San Miguel.*

References

BENTY, N., 2016. Falcor real-time rendering framework. https://github.com/NVIDIA/Falcor.

CASTANO, I., 2010. Computing alpha mipmaps. http://the-witness.net/news/2010/09/computing-alpha-mipmaps/.

ENDERTON, E., SINTORN, E., SHIRLEY, P., AND LUEBKE, D. 2010. Stochastic transparency. In *Symposium on Interactive 3D Graphics and Games*, 157–164.

GLASSNER, A. 2015. Interpreting alpha. *Journal of Computer Graphics Techniques 4*, 2, 30–44.

GUENTER, B., FINCH, M., DRUCKER, S., TAN, D., AND SNYDER, J. 2012. Foveated 3d graphics. *ACM Transactions on Graphics 31*, 6, 164:1–10.

KARIS, B. 2014. High-quality temporal supersampling. In *SIGGRAPH Course Notes: Advances in Real-Time Rendering in Games.*

KHARLAMOV, A., CANTLAY, I., AND STEPANENKO, Y. 2008. *GPU Gems 3*. Addison-Wesley, ch. Next-Generation SpeedTree Rendering, 69–92.

LAINE, S., AND KARRAS, T. 2011. Stratified sampling for stochastic transparency. *Computer Graphics Forum 30*, 4, 1197–1204.

LOTTES, T. 2009. FXAA. Tech. rep., NVIDIA, http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf.

MCGUIRE, M. 2016. *The Graphics Codex*, 2.13 ed. Casual Effects.

MOORE, J., AND JEFFERIES, D. 2009. Rendering technology at Black Rock Studios. In *SIGGRAPH Course Notes: Advances in Real-Time Rendering in Games.*

MULDER, J., GROEN, F., AND VAN WIJK, J. 1998. Pixel masks for screen-door transparency. In *Proceedings of Visualization*, 351–358.

PATNEY, A., SALVI, M., KIM, J., KAPLANYAN, A., WYMAN, C., BENTY, N., LUEBKE, D., AND LEFOHN, A. 2016. Towards foveated rendering for gaze-tracked virtual reality. *ACM Transactions on Graphics 35*, 6, 179:1–12.

PORTER, T., AND DUFF, T. 1984. Compositing digital images. In *Proceedings of SIGGRAPH*, 253–259.

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible rendering of 3-d shapes. *Proceedings of SIGGRAPH*, 197–206.

WYMAN, C. 2016. Exploring and expanding the continuum of OIT algorithms. In *High Performance Graphics*, 1–11.

WYMAN, C. 2016. Stochastic layered alpha blending. In *ACM SIGGRAPH 2016 Talks*.